Paul McJones

BCPL Users Guide

J. H. Morris
Computer Science

December 1969

Computer Center
University of California
Berkeley

TABLE OF CONTENTS

BCPL is a programming language for non-numeric applications such as compiler-writing and general systems programming. It has been used successfully to implement compilers, interpreters, text editors and a batch-processing operating system. The BCPL compiler is written in BCPL and runs on the Computer Center's CDC 6400.

Some of the distinguishing features of BCPL are:

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

The basic data object is a word (60 bits on the 6400) with no particular disposition as to type. A word may be treated as a bit-pattern, a number, a subroutine entry or a label. Neither the compiler nor the run-time system makes any attempt to enforce type restrictions. In this respect BCPL has both the flexibility and pitfalls of machine language.

Manipulation of pointers and vectors is simple and straightforward.

All subroutines are re-entrant and recursive since all data are kept in a stack. This is useful for multi-programming or applications where recursion is useful (e.g., tree-processing).

This manual is not intended as a primer; the constructs of the language are presented with scant motivation and few examples. To use BCPL on the 6400 effectively one must have a good understanding of how a computer works and be familiar with the operation of the 6400 and the SCOPE operating system. It is a useful language but has few provisions for protection of the naive user.

Acknowledgements

In the interest of making documentation of BCPL available quickly, large portions of this manual were taken from a very well-written memorandum by R. H. Canady and D. M. Ritchie of Bell Telephone Laboratories. Naturally any errors or omissions in this manual are my responsibility.

The initial design and implementation of BCPL were done by Martin Richards of Cambridge University, England.

The implementation for the 6400 was done with the assistance of Richard Aronoff.

LANGUAGE DEFINITION 2

2.1 Program

On the outermost level, a BCPL program consists of declarations: 'function', 'global', 'manifest', and 'label' declarations. But rather than starting from the outside and working in, the constructs of a BCPL program will be described from the inside out. The most basic construct is the 'element'.

2.2 Elements

```
<element>::= <identifier> | <integer constant> |
             <octal constant> |
             <string constant> |
             <character constant> | TRUE | FALSE
```

An <identifier> consists of up to 20 alphanumeric characters, the first of which must be a letter.

An <integer constant> is a sequence of digits. An <octal constant> begins with $8 followed by octal digits. The reserved word TRUE denotes -0 = $8777.....777 (i.e., a word of 1 bits) and FALSE denotes +0 . However, in any context where a truth value is expected, any negative value is interpreted as true.

A <string constant> consists of up to 128 characters enclosed by Ξ's. The internal character set is ASCII. The character Ξ can be represented in a string constant only by the pair *Ξ and the character * can be represented only by the pair **. Other characters may be represented as follows:

*n	is newline
*t	is horizontal tab (space up to column 11,21,31, etc.)
*Ønnn	represents the octal character code nnn where nnn is three octal digits.

A string is represented as a sequence of 60 bits words with eight 7-bit characters packed in the low-order 56 bits of each word. In the last word the characters are left justified, followed by ~~at least~~ one ~~zero~~ byte.

140B .

	A		S	T	R	I	N	G
		I	N		M	E	M	O
	R	Y	*Ø000	*Ø000	*Ø000	*Ø000	*Ø000	*Ø000

Each appearance of a string constant generates a new static vector of cells to contain the string. The value of the string constant is the address of this vector.

A character constant consists of up to 8 characters enclosed by ↓ characters. The character ↓ can be represented in a character constant only by the pair *↓ . The other escape conventions are the same as for a string constant.

A character constant is right justified in a word. Thus

↓A↓ = $8141

2.3 Expressions

The next construct in BCPL is the expression. Because an identifier has no type information associated with it, the type of an element is assumed to match the type required by its context.

All expressions are listed below. E1, E2 and E3 represent arbitrary expressions except as noted in the descriptions which follow the list, and C0, C1, etc., represent constant expressions (whose value is known at compile time - see Section 2.3.7).

primary	element	
	(E1)	
result	VALOF block	
function	E1 (E2,E3,...)	
addressing	E1.E2	subscripting
	LV E1	address generation
	RV E1	indirection
arithmetic	+E1	
	-E1	
	E1*E2	
	E1/E2	
	E1 REM E2	Integer remainder (modulus)
	E1+E2	
	E1-E2	
relational	E1 = E2	
	E1 ≠ E2	not equal
	E1 < E2	
	E1 > E2	
	E1 ≤ E2	
	E1 ≥ E2	
shift	E1 LSHIFT E2	left shift by E2 ≥ 0 bits
	E1 RSHIFT E2	right shift by E2 ≥ 0 bits
	E1 ↑ E2	arithmetic shift or $E1*2^{E2}$
logical	¬ E1	not (complement) E1
	E1 ∨ E2	inclusive or
	E1 ∧ E2	and
	E1 EQV E2	bitwise equivalence
	E1 NEQV E2	bitwise not-equivalence (exclusive or)
conditional	E1→E2,E3	
table	TABLE C0,C1,C2,...	

The relative binding power of the operators is as follows:

(highest)	VALOF
	function
	. (subscripting)
	LV RV
	* / REM
	+ -
	LSHIFT RSHIFT ↑
	relationals
	¬
	∨
	∧
	EQV NEQV
	→
(lowest)	TABLE

The VALOF expression will be described in 2.5.5, after the construct <block> has been described.

The value of a TABLE expression is the address of a static vector of cells initialized to the values of the constant expressions C0,C1,... . A table is thus closely analogous to a string constant.

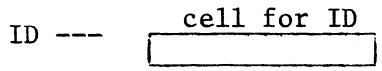
2.3.1 Addressing operators

The most interesting operators in BCPL are those which allow one to generate and use addresses. An address may be manipulated with integer arithmetic and is indistinguishable from an integer until it is used in a context which requires an address. For example, if X contains the address of a word in storage, then

X + 1

is the address of the next word.

If ID is an identifier, then associated with ID is a single word of memory, which is called a cell.



The content of this cell is called the value of ID. The address of the cell is called the address of ID.

An address may be used by applying the operator RV. The expression

RV E1

has as value the contents of the cell whose address is the value of the expression E1. Only the low-order 18 bits of E1 are used.

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

LV E1

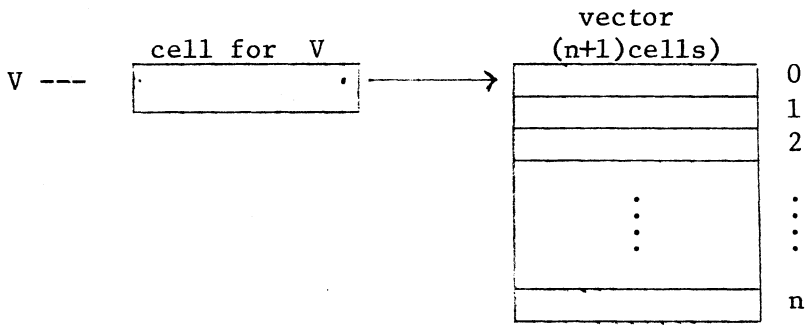
is valid only if E1 is

(1) an identifier, in which case the value of LV ID is the address of ID.

(2) a vector expression, in which case the value of LV E1.E2 is E1 + E2.

(3) an RV expression, in which case the value of LV RV E1 is E1.

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



The value of the expression

V.E1

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

address of this cell is the value of

V + E1

hence

LV V.E1 = V + E1

This relation is true whether or not the expression

V.E1

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

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

The interpretation of

RV E1

depends on context as follows:

(1) If it appears as the left-hand side of an assignment statement, e.g.,

RV E1 := E2

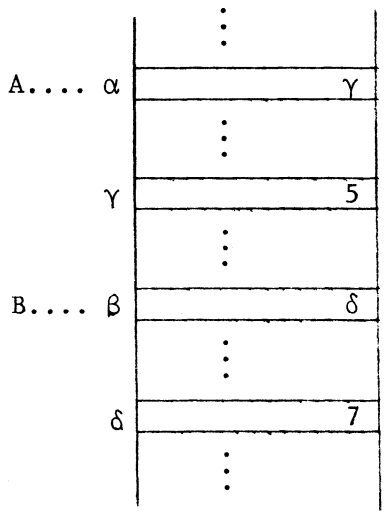
E1 is evaluated to produce an address and E2 is stored there.

(2) LV (RV E1) = E1 as noted above.

(3) In any other context E1 is evaluated and the contents of that value, treated as an address, are taken.

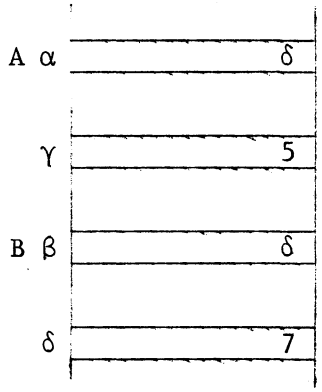
Thus, RV forces one more contents-taking than is normally demanded by the context.

As a summarizing example, consider the memory configuration depicted below

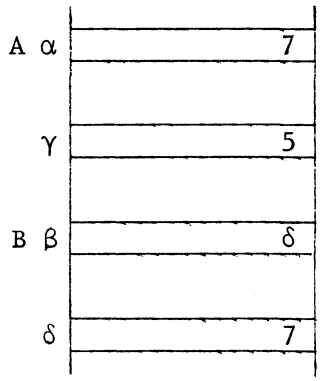


I.e., α and β are the addresses of A and B respectively. Then each of the following assignments induces the memory configuration shown adjacent,

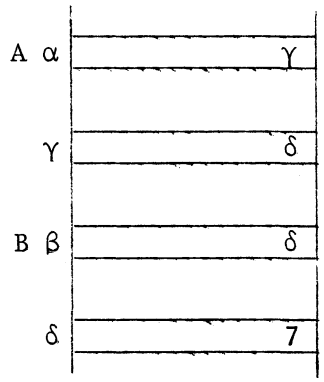
A := B



A := RV B



RV A := B



Note that

LV A := B

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

2.3.2 Arithmetic operators

There are two kinds of addition and subtraction, short and long. A short operation is undefined if either of its operands or its result is greater than 2^{17} in absolute value[1]. The long operations are defined for any 60-bit quantities. The short operations are written

E1 + E2 and E1 - E2

while the long are written

E1 +. E2 and E1 -. E2

-0 = \$87....7 behaves like +0 under addition and subtraction.

In general, multiplication, division, and remainder are defined only when the operands and results are less than 2^{48} in absolute value[2]. The behavior of -0 is undefined.

A REM B = A - (A/B) * B

2.3.3 Relations

As with addition and subtraction, there are two kinds of relational operators, short and long. The long version is obtained by suffixing a "." to the short version. (E.g., =., $\neq$., <., >., $\leq$., $\geq$.)

A relational expression of the form

$$E_1 \; R_1 \; E_2 \; R_2 \; E_3 \cdots E_{n-1} \; R_{n-1} \; E_n$$

is equivalent to

$$E_1 \; R_1 \; E_2 \wedge E_2 \; R_2 \; E_3 \wedge \cdots \wedge E_{n-1} \; R_{n-1} \; E_n$$

The result of relations involving -0 is undefined.

---

1 In the current implementation only operations involving small constants (i.e., $|x|<2^{17}$) are performed in 18-bit mode.

2 In the current implementation multiplication by constants having less than 7-bits in their absolute value is accomplished by shifts and adds so 60-bit operands are possible. Division or remainder by a constant power of 2 is done by shifting or masking respectively.

2.3.4 Shift operators

In the expression E1 LSHIFT E2 (E1 RSHIFT E2) E2 must evaluate to a non-negative number. The value is E1, taken as bit-pattern, shifted left (right) by E2 bits. Vacated positions are filled with 0 bits. The expression E1↑E2 calls for an arithmetic shift of E1 by E2 places. If E2 is positive E1 is shifted left circular; if E2 is negative E1 is shifted right with sign extension[1].

2.3.5 Logical operators

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

E1 ∨ E2 ∧ ¬ E3

will be true if

E1 is true (negative), in which case E2 and E3 are not evaluated

or if

E2 is true (negative) and E3 is false (positive).

In 'bit' context, the ¬ operator causes bit-by-bit complementation of its operand. The other operators combine their operands bit-by-bit according to the following table

operands		operator ∧	∨	EQV	NEQV
0	0	0	0	1	0
0	1	0	1	0	1
1	0	0	1	0	1
1	1	1	1	1	0

[1] On the 6400 arithmetic shifts are slightly faster than logical shifts.

2.3.6 Conditional operator

The expression

E1 → E2,E3

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

2.3.7 Constant expression

A constant expression is any expression involving only constants and operators other than LV, RV, VALOF, vector application (.), and TABLE.

2.4 Blocks

A block consists of one or more commands and/or declarations, enclosed by the symbols [called 'sectbra', at the beginning and], called 'sectket', at the end.

A sectbra or sectket may be "tagged" with up to 8 alphanumeric characters, terminated by the first nonalphanumeric character following the sectbra or sectket. A sectbra or sectket immediately followed by a space is in effect tagged with null.

A sectbra can be matched only by an identically tagged sectket. When the compiler finds a sectket, if the nearest sectbra (smallest currently open block) does not match, that block is closed and the process repeats until the matching sectbra is encountered.

Thus it is impossible to write blocks which are overlapping (not nested).

A block may be used wherever a command is allowed, and in addition is required in a few contexts where a command is not permitted. A block may be used for two purposes: to group a set of commands which are to be treated as a unit, and to delimit the scope of declarations.

2.5 Commands

Commands are separated by semicolons (;). However, in most cases the compiler automatically inserts a semicolon at the end of each line if it is syntactically correct there (see Section 2.7.3).

The pair of reserved words DO and THEN are synonymous. Note that AND and OR are not operators.

The complete set of commands is shown here, with E, E1, E2 and E3 denoting expressions and C, C1, and C2 denoting commands:

routine	E1(E2,E3,...)
assignment	<expression list> := <expression list>
conditional	IF E DO C
	UNLESS E DO C
	TEST E DO C1 OR C2
looping	WHILE E DO C
	C REPEAT
	C REPEATUNTIL E
	C REPEATWHILE E
for	FOR N=E1 TO E2 DO C
result	RESULTIS E
swtichon	SWITCHON E INTO [...]
transfer	GOTO E
	FINISH
	RETURN
	BREAK
block	[...]

Discussion of the 'routine' command

E1(E2,...)

which calls the routine whose address is E1 will be deferred to Section 2.6.6.

2.5.1 Assignment

The command

E1 := E2

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

(1)	an identifier	ID
(2)	a vector expression	E3.E4
(3)	a value-as-address expression	RV E3
(4)	a conditional	E3 → E4,E5

Case (1) is obvious. Cases (2) and (3) have been described in Section 2.3.1.

E3 → E4, E5 := E6

has the same effect as

```
[LET t = E6
 TEST E3
   DO E4 := t
   OR E5 := t]
```

where t is a new identifier.

A list of assignments may be written thus:

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

where Ei and Fi are expressions. This is equivalent to

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

2.5.2 Conditional commands

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

Expression E is evaluated in truth-value context. Command C1 is executed if E is true (negative), otherwise command C2 is executed.

2.5.3 Looping commands

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

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

```
C REPEAT
```

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

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

```
IF  E  DO  C  REPEAT
```

is the same as

```
IF  E  DO  [C REPEAT]
```

2.5.4 For command

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

N must be an identifier. This command will be described by showing an equivalent block.

```
[LET N,t = E1,E2
 UNTIL N>t DO
[C
 N := N+1]]
```

Note: The declaration

```
LET  ID = E
```

declares a new cell with identifier ID (see Section 2.6.3).

Note that t is a new identifier not occuring in C.

The most unusual feature of this command is that the identifier N is not available outside the scope of the command.

2.5.5 Resultis command, and value blocks

The expression

VALOF [.....]

defines a 'value block'. It is evaluated by executing the commands (and declarations) in the block, until a RESULTIS command

RESULTIS E

is encountered. The expression E is evaluated and its value becomes the value of the value block. Execution of commands within the value block ceases.

A value block must contain one or more RESULTIS commands and one must be executed.

In the case of nested value blocks, the RESULTIS command terminates only the innermost VALOF block containing it.

2.5.6 Switchon command

SWITCHON E INTO <block>

where the block contains labels of the form:

CASE <constant expression> : or
DEFAULT:

The expression E is first evaluated and if a case exists which has a constant with the same value then execution is resumed at that label; otherwise, if there is a default label then execution is continued from there, and if there is not, execution is resumed just after the end of the SWITCHON command.

The switch is implemented as a direct switch, a sequential search or binary search depending on the number and range of the case constants.

2.5.7 Transfer of Control

```
GOTO E
FINISH
RETURN
BREAK
```

The command GOTO E interprets the value of E as an address, and transfers control to that address. The command FINISH causes an implementation-dependent termination of the entire program. RETURN causes control to return to the caller of the routine. BREAK causes execution to be resumed at the point just after the smallest textually enclosing looping command. The looping commands are those with the following key words:

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

2.6 Declarations

There are eight distinct declarations in BCPL: GLOBAL, MANIFEST, dynamic cell, dynamic vector, function, formal parameter, routine, and label.

2.6.1 Global

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

GLOBAL [Name : constant-expression]

associates the identifier Name with the specified location in the global vector. Thus Name identifies a static cell which may be accessed by Name or by any other identifier associated with the same global vector location. Global declarations may be combined:

GLOBAL [N1:C1;N2:C2;...;Nn:Cn]

Note the absence of a final ; .

The scope of a global declaration, i.e., the region of program where the identifier is known, is the region immediately following the global declaration up to the end of the smallest textually enclosing block, except where the identifier is redeclared within that scope.

2.6.2 Manifest

An identifier may be associated with a constant by the declaration

```
MANIFEST [Name = constant-expression]
```

The scope of this declaration is the same as for a global declaration. Within the scope of this identifier, use of the identifier is exactly equivalent to using the constant expression.

The constant expressions in a multiple manifest declaration are all evaluated before the declarations take effect. Thus

```
MANIFEST [MASK = $8777; NMASK = ¬ MASK]
```

is illegal (unless MASK has been declared in a previous MANIFEST declaration). However

```
MANIFEST [MASK = $8777]
MANIFEST [NMASK = ¬ MASK]
```

will declare NMASK as ¬ $8777.

A manifest constant, like any constant, does not have an address. Manifest declarations may be combined exactly like global declarations.

2.6.3 Dynamic Cell

The declaration

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

creates n dynamic cells (words) and associate them with the identifiers N1,N2,...,Nn. These names are known in the remainder of the block containing the LET declaration. They are also known in the expressions E1,E2,...,En.[1] They are not known within the body of any function or routine declared subsequently in the block.

Example

```
[LET A = E1
 LET B = E2
 LET F(x) = E3
 C1;C2;...;Cm]
```

---

[1] This convention is not particularly useful.

A is known in E1,E2,C1,...,Cm
B is known in E2,C1,...,Cm
F is known in E3,C1,...,Cm

The words reserved by a dynamic cell declaration are released when the block in which the declaration appears is left.

Example

```
[LET  A = 1
   B := LV A]
[LET  X = 7
   C := RV B]
```

The effect of this program segment is not defined. In the current implementation, it is likely that 7, not 1 will be assigned to C.

2.6.4 Vector

The declaration

LET N = VEC m

where m is a constant expression, creates a dynamic vector of m+1 cells by reserving m+1 cells of contiguous storage in the stack, plus one cell which is associated with the identifier N. The scope of N is the same as for a dynamic cell declaration (2.6.3). Execution of the declaration causes the value of N to become the address of the block of m+1 cells. The storage created is released when the block is left.

2.6.5 Function

The declaration

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

declares a function named N with m parameters. The parentheses are required even if m=0 . The scope of the parameter names is the expression E. A parameter name has the same syntax as an identifier.

If the declaration is within the scope of a global declaration for N, then the global cell will be initialized to the address of the function before execution of the program. Thus the function N may be accessed from anywhere. Otherwise a static cell is created, is associated with the identifier N, and is initialized to the address of the function. In this case the identifier has the same scope as a global cell declaration.

The function is invoked by the expression

E0(E1,E2,..,Em)

where expression E0 evaluates to the address of the function. In particular, within the scope of identifier N the function may be invoked by the expression

N(E1,E2,..,Em)

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

Each value passed as a parameter is copied into the argument list, even if the expression for the parameter is a simple identifier. Thus arguments are always passed by value. The value passed may, of course, be an address.

2.6.6 Routine

The declaration

LET N(P1,P2,...,Pm) BE <block>

is identical in effect to a function declaration except that

(1) the body is a block rather than an expression
(2) no value is returned to the caller.

The scope of the parameter identifiers is the block.

The routine is called by the command

E0(E1,...,Em)

where expression E0 evaluates to the address of the routine. As in the case of a function, the routine N may be invoked by the command

N(E1,...,Em)

within the scope of identifier N.

Any function may be called as if it were a routine, but if a routine is called as a function, the value returned is undefined.

2.6.7 Label

A label is declared by

Name:

A label declaration may precede any command or label declaration, but may not precede any other form of declaration. Exactly as in the case of a function or routine, the label declaration creates a static cell if it is not within the scope of a global declaration of the same identifier. The local or global cell is initialized before execution with the address of the first command following the label declaration, so that the command

GOTO Name

has the expected effect.

The scope of a label identifier is different from any other declaration, because it includes all of the largest enclosing routine or function including the portion before the declaration itself.

Labels may be assigned to variables and passed as parameters. In general they should not be declared global, but can be assigned to global variables. (see 3.2.2. for an exception). Transferring to a label after the block in which it was declared has been left will produce chaotic (undefined) results.

2.6.8 Simultaneous declarations

Any declaration of the form

LET ------

may be followed by one or more declarations of the form

AND ------

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

2.7 Miscellaneous Features

2.7.1 GET

The command

GET ≡string≡

causes the file identified by ≡string≡ to be included in the source text in place of the 'get' command. The translation of the string into a file name, and the internal format of the file, are implementation dependent.

Under SCOPE, the first seven characters of the string are used as the file Name. The corresponding file is expected to be Hollerith card images.

2.7.2 Comments and spaces

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

Blank lines (lines including only the characters 'space', 'tab', and/or 'newline') are ignored also.

Space and tab characters may be freely inserted except inside an element, inside a system reserved word (e.g. VALOF), or inside an operator (e.g. :=). Space or tab characters are required to separate identifiers or system reserved words from adjoining identifiers or system reserved words.

2.7.3 Preprocessor

In order to make BCPL programs easier to read and to write, the compiler allows the syntax rules to be relaxed in certain cases. Source text input to the compiler is scanned by a preprocessor which is capable of inserting semicolons, and the reserved word DO (or THEN), where appropriate.

Thus the programmer normally can write BCPL programs without using the command terminator (semicolon) and with fewer DOs than the strict syntax requires.

The preprocessor inserts a semicolon between adjacent items if they appear on different lines and if the first is from the set of symbols which may end a command, namely:

```
BREAK  RETURN  FINISH  REPEAT
)  <element>  ]
```

and the second is from the set of items which may start a command, namely:

```
TEST  FOR  IF  UNLESS  UNTIL  WHILE  GOTO
SWITCHON  (  RV  <element>
RESULTIS  CASE  DEFAULT  BREAK  RETURN
FINISH  [
```

The symbol DO is inserted between pairs of items if they appear on the same line and if the first is from the set of items which may end an expression, namely:

```
)  <element>  ]
```

and the second is from the set of items which must start a command, namely:

```
TEST  FOR  IF  UNLESS  UNTIL  WHILE  GOTO
RESULTIS  CASE  DEFAULT  BREAK  RETURN
FINISH  SWITCHON  [
```

An as example, the following two program segments are equivalent:

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

2.8 The Run-time Library

2.8.1 Input-Output Routines

The input/output facilities of BCPL are quite primitive and simple.

INITIALIZEIO(Y,SIZE) is a routine that sets up a buffer area in the vector Y of length SIZE. It initializes a global pointer to the buffer area (IOBASE) and the character conversion tables (C6TO7 and C7TO6). If N is the maximum nubmer of simultaneously open files expected during the job, SIZE should be N*BUFFERSIZE (a manifest constant = 7 + the real buffer size, declared as 136 in BCPLGD).

FINDINPUT(LFN) is a function taking a display-coded file name (LFN) and returning a stream-pointer to be used by the input routines. FIND-INPUT initializes an input buffer and attempts to read a buffer-load of the named file. If no information is found, a warning message is printed, the file rewound, and a second read attempted. If this read fails the job is aborted. If the file has already been opened, the job is aborted.

CREATEOUTPUT(LFN) is a function taking a display-coded file name (LFN) and returning a stream-pointer to be used by output routines. No testing of the external file environment occurs, but a file may be opened any number of times.

READCH(STREAM,CH) is a routine which reads the next character from an input stream and stores it (indirect) in CH. Thus to get the character into a variable, A, one executes READCH(S,LV A). If the stream is at an end of the record the character ENDOFSTREAMCH (= $8255) is stored.

WRITECH(STREAM,CH) is a routine which writes a character onto an output stream.

READVEC(STREAM,V,N,EORL,EORC) reads N words from STREAM into V.0,...,V.(N-1) . If less than N words remain in the STREAM the number of words actually read is stored (indirect) in EORC and a transfer to EORL is performed. Mixing calls of READVEC and READCH on the same stream produces undefined results.

WRITEVEC(STREAM,V,N) writes N words from V.0,...,V.(N-1) onto STREAM. Mixing calls of WRITEVEC and WRITECH on the same stream produces undefined results.

ENDREAD(STREAM) positions the file at the next end of record and releases the buffer space associated with STREAM.

ENDWRITE(STREAM) writes out anything remaining in the buffer, writes an end of record, and releases the buffer space. This action is not performed until the file has been closed as many times as opened.

ENDOFSTREAM(STREAM) returns TRUE if the stream is at an end of record, otherwise FALSE.

CLOSEALL ( ) performs ENDWRITEs and ENDREADs on all open streams until they are closed.

ABORT ( ) performs a CLOSEALL and aborts the job.

2.8.2 Other useful subroutines

PACKSTRING(V,S) packs characters V.1,V.2,...,V.(V.0) into the vector S (i.e. into S.0,S.1,...,S.(V.0/8 + 1)).

UNPACKSTRING(S,V) stores the characters of S in V.1,...,V.N and stores N in V.0.

BCDWORD(S) produces a left-justified, display-coded word from a (long) string S .

ASCII(D,A) packs the display-coded word D into vector A .

WRITE(S) writes the characters of S onto the output stream OUTPUT (a global variable).

WRITEN(N) writes the number N onto the output stream OUTPUT.

WRITEO(N) writes the number N (in octal) onto the output stream OUTPUT.

2.8.3 Global variables for I/O

The following global variables are used by the I/O routines. They are declared in BCPLGD (see section 3.1.3).

IOBASE: holds pointer to buffer area; initialized by INITIALIZEIO, used by FINDINPUT, CREATEOUTPUT, and CLOSEALL.

C6TO7: points to a display-code to ASCII conversion vector; initialized by INITIALIZEIO, used by REACH and ASCII.

C7TO6: points to an ASCII to display code conversion vector; initialized by INITIALIZEIO, used by WRITECH and BCDWORD.

OUTPUT: points to an output stream; used by WRITES, WRITEN, and WRITEO.

MONITOR: points to an output stream for error messages; should be initialized before any I/O is attempted.

The four common files, BCPL, BCPL2, BCPLIO, and BCPLGD are public and may be accessed by any user in the normal way. (BCPL2 is the second pass of the compiler.)

3.1 Compiling

3.1.1 The Control Card

The BCPL compiler is directed to translate a source deck by the SCOPE control card:

```
LGO,BCPL,I=input,L=listing,B=binary,C=compass,O=ocode,N=name,T=tree,SA,D.
```

All parameters are optional and may appear in any order. Their interpretation is as follows:

Parameter	Default Value	Use
I	INPUT	Designates the file containing the source code to be compiled. If the file appears to be empty, it is rewound and tried again. The source deck is terminated by an end of record.
L	OUTPUT	Designates the file on which the source text, along with diagnostics and other information will be written. L=0 suppresses listing except for diagnostics which will appear on OUTPUT.
B	LGO	Designates the file on which the relocatable binary will be written. B=0 suppresses the output of binary.
C	0	Designates the file on which a COMPASS version of the program is written. This version may be assembled by COMPASS. C=0 suppresses COMPASS output.
O	OCODE	Designates the scratch file to be used for transmitting an intermediate object code between passes of the compiler. This file is always rewound at the start of compilation.
N	(same as B)	Gives a name to the binary and/or COMPASS program produced. I.e., N=name would cause "IDENT name" to be the first line of the COMPASS program.
T	0	Designates a file on which a representation of the parse tree will be written. T=0 suppresses the printing of the tree.
SA		If included as a parameter, suppresses abortion of the job if the compiler finds errors in the source program. (The compiler often produces an executable [but dangerous] program even when errors occur.)
D		If included as a parameter, the listing will he double-spaced.

3.1.2 Field Length During Compilation

A field length of $45{,}000_8$ should allow sufficient space for the compiler to translate most programs. If the stack space needed grows beyond the declared field length, an Arithmetic Error - Mode 1 will occur. There should never be an Arithmetic Error for any other reason, but there may be. The distinguishing characteristics of an Arithmetic Error caused by stack overflow are

1. B6 contains a number relatively close to the field length.
2. The offending instruction is either

```
        SAi B6 + K
or      SAi Xj + K
or      SAi Xj + Bk
```

   where i = 6 or 7 and the effective address is greater than the field length.

If these conditions are not satisfied, there is a bug in the compiler.

3.1.3 Library Declarations

If the program to be compiled references any routines from the library (see 2.8 for a description of these), it must include appropriate global declarations for the routines and global cells referenced. The simplest way to accomplish this is to prefix the command

```
GET =BCPLGD=
```

to the program text. This will insert the contents of common file BCPLGD in the program at that point. The information in BCPLGD is shown below. If the library BCPLIO is ever to be loaded with the program, then no other global identifiers should be given the numbers assigned by the file BCPLGD to the I/O routines.

```
GLOBAL                  //    SUBROUTINE NAMES

[INITIALIZEIO:2         //    Sets up buffers
FINDINPUT:3             //    Opens input file
CREATEOUTPUT:4          //    Opens output file
READCH:5                //    Reads a character
WRITECH:6               //    Writes a character
READVEC:7               //    Reads words
WRITEVEC:8              //    Writes words
ENDREAD:9               //    Closes input file
ENDWRITE:10             //    Closes output file
```

```
ENDOFSTREAM:11        //    Tests end-of-record
CLOSEALL:12           //    Closes all files
ABORT:13              //    Aborts job
PACKSTRING:14
UNPACKSTRING:15
BCDWORD:16            //    ASCII-display code converter
ASCII:17              //    Display code-ASCII converter
WRITES:18             //    Writes a string
WRITEN:19             //    Writes a number
WRITEO:20             //    Writes a number in octal]

GLOBAL                //    VARIABLES

[IOBASE:30            //    Pointer to buffer area
C6TO7:31              //    Display-code-ASCII vector
C7TO6:32              //    ASCII-display-code vector
OUTPUT:33             //    Output stream pointer
MONITOR:34            //    Error stream pointer]

MANIFEST

[BUFFERSIZE=136       //    A 129 word buffer + 7 word FET
ENDOFSTREAMCH = 255//       EOS signal]
```

3.1.4 Diagnostics

There are three types of diagnostics given during compilations: parse, translation and general.

A parse diagnostic occurs when a relatively simple syntactic error is detected during the early phases of compiling. An arrow (↑) is printed under the last character read in before the error became apparent. A brief description of the error is printed. No more than one error (the first) on any given line is reported. Errors reported on lines subsequent to the first error should be regarded with suspicion since the compiler does not recover very well.

A translation diagnostic occurs in the later phases of compilation and reports errors such as use of an undeclared identifier. Each error is briefly described and a representation of the relevant portion of the parse tree is printed.

A few general diagnostics may occur at any time. They include such mishaps as table overflows and missing input files.

3.2 Executing

3.2.1 Loading

The common file BCPLIO contains an initializing program, START, and the input-output library routines, IO1 and IO2. Under normal circumstances this file should be loaded before any compiled BCPL programs.

If any programs not produced by the BCPL compiler (or disguised to appear so) are to be loaded (e.g., the TRACE package) then the load sequence should be

```
BCPLIO
All  BCPL programs
TAIL     (shown below)
All other programs
```

The program for TAIL can be produced by the following COMPASS program

```
IDENT   TAIL
SA1     B5 + B1
SB4     X1
JP      B4
END
```

See Appendix C for details of the BCPL run-time conventions.

3.2.2 The Main Program

The first command executed in a core-load of BCPL programs is the one labelled by a global name (e.g. BEGIN) declared to be GLOBAL 1 in the source program. Before any input-output can be performed, INITIALIZEIO must be called. A minimal main program is shown below.

3.3 A Complete Job

The following deck constitutes a simple BCPL job.

```
J6000,30,45000.
COMMON,BCPL.
COMMON,BCPL2
COMMON,BCPLIO.
COMMON,BCPLGD.
LGO,BCPL,N=MAIN.
LGO,BCPL,N=BUS.
LOAD,BCPLIO.
LGO.
(7-8-9)
```

```
GET ≡BCPLGD≡                          // To declare IO routines
                                         and global cells
GLOBAL [BEGIN :1; BUSINESS: 100]      // Private global decs.
BEGIN:                                // Execution starts here
[LET BUFFER = VEC BUFFERSIZE          // One buffer
INITAILIZEOIO (BUFFER,BUFFERSIZE
OUTPUT := CREATEOUTPUT (BCDWORD(≡OUTPUT≡))
MONITOR := OUTPUT                     // Put errors on output
WRITES (≡NOW WE GET DOWN TO BUSINESS *N ≡)
BUSINESS ()
CLOSEALL()]                           // Close output file
(7-8-9)

//  THE SECOND PROGRAM

GLOBAL [BUSINESS: 100]
LET BUSINESS () BE [RETURN]
```

APPENDIX A : Reserved Words and Tokens

The following list of words and symbols are treated as atoms by the BCPL syntax analyzer. The alternate forms may be used to avoid multiple punching.

Standard	Multiple Punch	Alternate
AND		
↑	11-5-8	ASHIFT
BE		
BREAK		
CASE		
DO		THEN
DEFAULT		
=		EQ
=.		LEQ
FALSE		
FINISH		
FOR		
≥	12-5-8	GE
≥.		LGE
GET		
GLOBAL		
>	11-7-8	GR
>.		LGR
IF		
INTO		
≤	5-8	LE
≤.		LLE
LET		
∧	0-7-8	LOGAND
∨	11-0	LOGOR
<	12-0	LS
<.		LLS

Standard	Multiple Punch	Alternate
LSHIFT		
LV		
MANIFEST		
≠	(apostrophe)	NE
≠.		LNE
NEQV		
¬	12-6-8	NOT
OR		ELSE
REM		MOD
REPEAT		
REPEATUNTIL		
REPEATWHILE		
RESULTIS		
RETURN		
RSHIFT		
RV		
SWITCHON		
TABLE		
TEST		
TO		
TRUE		
UNLESS		
UNTIL		
VEC		
VALOF		
WHILE		
+		
+.		
-		
-.		
*		
/		
,		
.		
;	12-8-7	

Standard	Multiple Punch	Alternate
:	2-8	
(		
)		
[	8-7	$(
]	0-8-2	$)
:=		
→	0-8-5	→,-*
$8		

A string constant is delimited by ≡'s (0-6-8) and a character constant by ↓'s (11-6-8).

APPENDIX B : BNF of BCPL

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

1. Comments are not included, and the space character is not represented even where required. In fact this only occurs between an identifier and a system word, and between two system words; that is, where an obvious misinterpretation would occur if the space were left out.

2. Block delimited tags are not included, since they are impossible to represent in BNF.

3. The graphic escape sequences allowable in strings are not represented.

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

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

6. Certain constructions can be used only in specific contexts. Not all these restrictions are included: for example, CASE and DEFAULT can be used only in switches, and RESULTIS only in a VALOF block. Finally, there is the necessity of declaring all identifiers before they are used.

The brackets { } are used to group categories without inventing a name and a suffixed * means "an arbitrary number ($\geq$0) of".

1. Identifiers, Strings, Numbers

<null> ::=

<letter> ::= A|B|...|Z

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

<digit> ::= <octal digit> |8|9

<string constant>[1] ::= ≡<128 or fewer characters>≡

<character constant> ::= ↓<8 or fewer characters>↓

<octal-number> ::= $8 <octal digit>*

<number> ::= <octal-number> | <digit> <digit>*

<identifier>[2] ::= <letter> {<letter>|<digit>}*

2. Operators

<addressop> ::= LV | RV

<multop> ::= * | / | REM

<addop> ::= + | - | +. | -.

<shiftop> ::= LSHIFT | RSHIFT | ↑

<relop> ::= = | ≠ | < | > | ≤ | ≥ | =. | ≠. | <. | >. | ≤. | ≥.

<eqvop> ::= EQV | NEQV

3. Expressions

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

<primary E> ::= <primary E> (<expression list>) | (<expression>) |
VALOF <block> | <element>

<vector E> ::= <vector E> . <primary E> | <primary E>

<address E>[3] ::= <addressop> <address E> | <vector E>

<mult E> ::= <mult E> <multop> <address E> | ~~<mult E>~~ address E>

<add E> ::= <add E> <addop> <mult E> | <mult E>

<shift E> ::= <shift E> <shiftop> <add E> | <add E>

<rel E> ::= <shift E>{<relop> <shift E>}*

<not E>::= ¬ <not E> | <rel E>

<and E> ::= <not E> { ∧ not E>}*

<or E> ::= <and E> { ∨ <and E>}*

<eqv E> ::= <or E>{<eqv op> <or E>}*

<conditional> ::= <eqv E> → <conditional>,<conditional>|<eqv E>

<expression>[4] ::= <conditional> | TABLE <constant expression>{,<constant
expression>}*

4. Lists of Expressions and Identifiers

<exp-list> ::= <expression> | <expression> , <exp-list>

<expression-list> ::= <null> | <exp-list>

<n-list> ::= <identifier> | <identifier> , <n-list>

<name-list> ::= <null> | <n-list>

5. Declarations

<manifest-item>[4] ::= <identifier> = <constant-expression>

<manifest-list> ::= <manifest-item> | <manifest-item> ; <manifest-list>

<manifest-declaration> ::= MANIFEST [<manifest-list>]

<global-item> ::= <identifier> : <constant-expression>

<global-list> ::= <global-item> | <global-item> ; <global-list>

<global declaration> ::= GLOBAL [<global-list>]

<simple-definition>[5] ::= <n-list> = <exp-list>

<vector-definition> ::= <identifier> = VEC <constant-expression>

<function-definition> ::= <identifier> ( ) = <expression>

<routine-definition> ::= <identifier> ( ) BE <block>

<definition> ::= <simple-definition> | <vector-definition> |
<function-definition> | <routine-definition>

<simple-declaration> ::= LET <definition>

<decl-tail> ::= AND <definition> | AND <definition> <decl-tail>

<simultaneous-declaration> ::= <simple-declaration> <decl-tail>

<declaration> ::= <simple-declaration> | <simultaneous-declaration> |
<global-declaration> | <manifest-declaration>

<declaration-part> ::= <declaration> | <declaration> ; <declaration-part>

6. Left-hand-side Expressions

<simple-LHSE> ::= <identifier> | <vector-application> | RV<address E>

<LHSE> ::= <simple-LHSE> | ~~<expression> → <LHSE> , <LHSE>~~

<left-hand-side-list> ::= <LHSE> | <LHSE> , <left-hand-side-list>

7. Commands

<assignment>[5] ::= <left-hand-side-list> := <exp-list>

<simple-command> ::= BREAK | RETURN | FINISH

<goto-command> ::= GOTO <expression>

<routine-command> ::= <function-application>

<resultis-command> ::= RESULTIS <expression>

<switchon-command> ::= SWITCHON <expression> INTO <block>

<repeatable-command> ::= <assignment> | <simple-command>
| <goto-command> | <routine-command>
| <resultis-command> | <repeated-command>
| <switchon-command> | <block>

<repeat-command> ::= <repeatable-command> REPEAT

<repeatwhile-command> ::= <repeatable-command> REPEATWHILE <expression>

<repeatuntil-command> ::= <repeatable-command> REPEATUNTIL <expression>

<repeated-command> ::= <repeat-command> | <repeatwhile-command>
| <repeatuntil-command>

<test-command> ::= TEST <expression> DO <command> OR <command>

<get-command> ::= GET <stringconst>

<if-command> ::= IF <expression> DO <command>

<unless-command> ::= UNLESS <expression> DO <command>

<while-command> ::= WHILE <expression> DO <command>

<until-command> ::= UNTIL <expression> DO <command>

<for-command> ::= FOR <identifier> = <expression> TO <expression> DO <command>

<unlabelled-command> ::= <repeatable-command> | <repeated-command>
| <test-command> | <if-command> | <get-command>
| <unless-command> | <while-command> | <until-command> | <for-command>

8. Labels, Prefixes, and Labelled Commands

<label-prefix> ::= <identifier> :

<case-prefix> ::= CASE <constant-expression> :

<default-prefix> ::= DEFAULT :

<prefix> ::= <label-prefix> | <case-prefix> | <default-prefix>

<command> ::= <unlabelled-command> | <prefix> <command>

9. Blocks

<command-list> ::= <command> | <command> ; <command-list>

<body> ::= <command-list> | <declaration-part> | <declaration-part> ; <command-list>

<block> ::= [<body>]

<program> ::= <body>

---

1 See 2.2 for further restrictions on string and character constants.

2 An identifier may not be a reserved word. See Appendix A for the list of reserved words.

3 The operands of LV are restricted as per 2.3.1.

4 A <constant-expression> is a <conditional> evaluable at compile time. Specifically, it cannot contain identifiers which are not manifest or the

operators LV, RV, VALOF, vector application (.) and TABLE.
The lengths of the two lists must be equal.

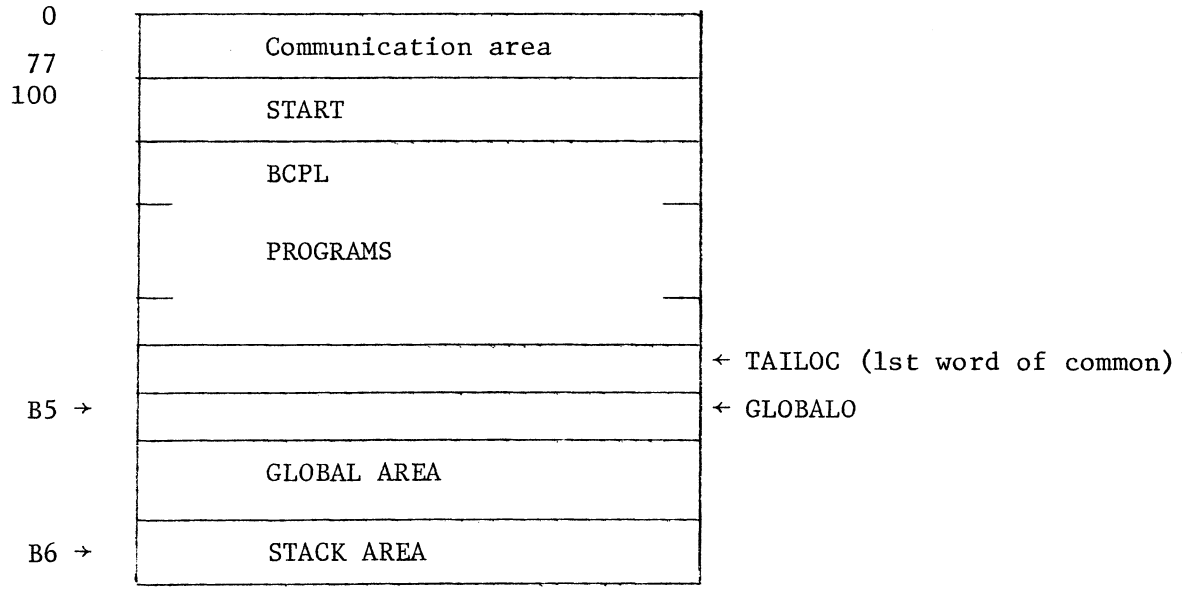
APPENDIX C : The Run-time Environment

1. Storage Allocation and Register Usage

Like any programs loaded by the SCOPE loader, BCPL object programs begin at 100_8. The last program is followed by the first word of blank common. Under normal conditions a special program START should be loaded before any BCPL programs. A sample START is shown in the next section. The following registers are used by BCPL programs:

B1	always contains 1
B2	always contains -1
B4	scratch register, used for all non-local transfers
B5	always contains the first common address + 1
B6	contains the dynamic stack pointer (>B5)
X0-X5	scratch registers
X6	always contains 0
X7	used for all non-zero stores

Depicted below is a core map with certain special locations noted:



2. Linkage

Communication between independently compiled programs is done through the Global Area rather than through the ENTRY - EXTERNAL mechanism of SCOPE (sorry). Each program has the responsibility of initializing the global cells for its entry points. This activity is accomplished as follows:

1. From the loader's point of view, START is the initial entry point to be transfered to after loading. After initializing various registers, START stores some transfer instructions in TAILOC (the first word of common) and simply falls through to the program loaded after it.

2. That program initializes the appropriate Global cells and falls through to the program following it.

3. The last program loaded falls into the ambush left in TAILOC by START. These instructions effect a transfer to the address stored in GLOBALO + 1 (i.e., GLOBAL 1). Hopefully, some program has stored a label there.

A sample START program:

```
          IDENT  START
RESET     MACRO                 This macro initializes the important registers
          SB5  =XGLOBALO
          SA2  B5
          SB6  X2               Reset stack pointer
          SB1  1
          SB2  -1
          BX6  X6-X6            Zero X6
          ENDM
          ENTRY  START,MASKTAB,GLOBALO
ONES      SET  60               These instructions set up a table of masks
MASKTAB   BSS  0                to be used
          DUP  61
ZEROS     SET  60-ONES          by compiled shift instructions
          VFD  ZEROS/0,ONES/-0
ONES      SET  ONES-1
          ENDD
TAIL      SA1  B5+B1            These will be stored in TAILOC
          SB4  X1
          JP   B4
GSIZE     EQU  400              400 is maximum global index
START     SX7  GLOBALO+GSIZE+1  First instruction executed
          SA7  GLOBALO          Leave stack pointer for reset
          RESET                 Initialize registers
          SA1  TAIL             Lay ambush and
          BX7  X1               fall through
          SA7  TAILOC
          USE  //               Put what follows in COMMON
```

```
TAILOC    BSS   1          Instructions stored here
GLOBALO   BSS   1          Holds B6 while non-BCPL code is running
          BSS   15000      Lots of space, can be truncated by RFL
          USE   0
          END   START      Signal Loader to start at START
```

3. CALL - SAVE - RETURN Conventions

By convention each subroutine takes its local storage from locations immediately above the address held in B6 at the time the subroutine is entered. Any parameters are placed in B6+2, B6+3, etc.

Between two BCPL programs, the rules are as follows:

1. The Caller
   a. Stores calling parameters in B6+N+2, B6+N+3,... where N is the number of words it is currently using for local storage.
   b. Places the entry point of the Callee in B4.
   c. Increments B6 by N.
   d. Places the return address in X7.
   e. Transfers (through B4) to the Callee.
2. The Callee
   a. Stores X7 (the return address) in the address in B6.
   b. Finds its parameters in B6+2, B6+3, etc.
   c. If returning a value, leaves it in X1.
   d. Returns by a transfer to the contents of the contents of B6 (again through B4).
3. The Caller
   a. Decrements B6 by N, restoring it to its original value.
   b. May expect a result in X1.

A typical function call, say X := F(X+1), might appear as follows:

```
          SA1   B6+30        X is in local location 30
          SX7   X1+B1        Add 1
          SA7   B6+47        Save 45 locations
          SA1   B5+10        F is GLOBAL 10
          SB4   X1
          SB6   B6+45        Increment Stack Pointer
          SX7   RLOC         Leave Return in X7
          JP    B4
RLOC      SB6   B6-45        Decrement Stack Pointer
          BX7   X1           Expect result in X1
          SA7   B6+30        Store X
```

The operations of the callee are quite simple. A skeleton for F would be

```
F       SA7     B6      SAVE RETURN
        :
        :
        SA1     B6+5    RESULT
        SA2     B6
        SB4     X2
        JP      B4      RETURN
```

4. A BCPL-COMPASS Interface

The following macros provide a fool-proof way of making a COMPASS program look like a BCPL program to the linkage mechanism and any calling program. The means for calling BCPL programs from COMPASS are analogous.

```
LINKAGE         MACROS
HEAD            MACRO
                LOCAL L
                EQ    L             Transfer around code
ENDC            MACRO               Define macro to be placed at end
                HERE
L               BSS   0
                ENDM
                ENDM
GLOBAL          MACRO   NAME,INDEX Initialize a global
                SX7     NAME
                SA7     B5+INDEX
                ENDM

SUBROUTINE      MACROS
SAVE            MACRO
                SA7   B6            Save Return
                SX7   B6            Hide B6 in GLOBAL0 for reset
                SA7   B5
                ENDM
RETURN          MACRO
                RESET               See START
                SA2   B6
                SB4   X2
                JP    B4
                ENDM
```

These MACROS might be used to write some floating-point functions as follows:

```
            IDENT       FLOAT
            HEAD
FMULT       SAVE
            SA1         B6+2
            SA2         B6+3
            FX1         X1*X2
            RETURN
FDIV        SAVE
            SA1         B6+2
            SA2         B6+3
            FX1         X1/X2
            RETURN
            ENDC
            GLOBAL      FMULT,201
            GLOBAL      FDIV,202
            END
```

APPENDIX D : Display Code ASCII Correspondence

Display Code Graphic	ASCII Graphic	ASCII Octal
A	a	141
B	b	142
C	c	143
D	d	144
E	e	145
F	f	146
G	g	147
H	h	150
I	i	151
J	j	152
K	k	153
L	l	154
M	m	155
N	n	156
O	o	157
P	p	160
Q	q	161
R	r	162
S	s	163
T	t	164
U	u	165
V	v	166
W	w	167
X	x	170
Y	y	171
Z	z	172
0	0	60
1	1	61
2	2	62
3	3	63
4	4	64
5	5	65
6	6	66
7	7	67
8	8	70
9	9	71
+	+	53
-	-	55
*	*	52
/	/	57
(	(	50
)	)	51
$	$	44
=	=	75
sp	sp	40

Display Code Graphic	ASCII Graphic	ASCII Octal
,	,	54
.	!	41
≡	%	45
[	[	133
]	]	135
:	:	72
≠	#	43
→	?	77
∨	|	174
∧	&	46
↑	\	130
↓	'	47
<	<	74
>	>	76
≤	{	173
≥	}	175
¬	^	136
;	;	63

The string constituents *N and *T map into the ASCII characters newline (12_8) and horizontal tab (11_8), respectively.