May 1971

Preface

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 CDC 6400 and either the SCOPE operating system or CAL TSS. BCPL is a useful language but has few provisions for protection of the naive user.

May 1971

Acknowledgements

In the interest of making documention 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 the author's responsibility.

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

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

May 1971

Table of Contents

Preface
Acknowledgements
Table of Contents

Chapter 1 - General Information
1.1 Language Characteristics
1.2 Properties of BCPL
1.2.1 Program
1.2.2 Elements
1.2.3 Expressions
1.2.4 Constant Expression
1.2.5 Table Expressions
1.2.6 Operators
Arithmetic
Shift
Relational
Logical
Addressing
Conditional
1.2.7 Blocks
1.3 Processor Environment (to be supplied)
1.3.1 Hardware
1.3.2 Software
SCOPE
TSS

Chapter 2 - Structure of BCPL
2.1 Commands
2.1.1 Assignment Commands
2.1.2 Conditional Commands
2.1.3 Looping Commands
2.1.4 FOR Commands
2.1.5 RESULTIS Command
2.1.6 SWITCH Command
2.1.7 Transfer of Control
2.2 Declarations
2.2.1 GLOBAL
2.2.2 Manifest
2.2.3 EXTERNAL
2.2.4 Dynamic Cell
2.2.5 Dynamic Vector
2.2.6 Function
2.2.7 Routine
2.298 Label
2.2.9 Simultaneous Declarations
2.3 Preprocessor
2.3.1 Get command
2.3.2 Comments and spaces

2.3.3 Insertion of 'DO' or 'THEN'
2.3.4 Insertion of ';'
2.4 Additional Features
2.4.1 Assembly Language Coding (to be supplied)
2.4.2 Library of subprograms
HEAD (to be supplied)
BCPLIO (to be supplied
FREESPACE (to be supplied)
Special TSS Functions (to be supplied)
Input/Output Functions

Chapter 3 - Using BCPL
3.1 How to write a BCPL Program
3.2 How to compile BCPL
3.2.1 BATCH
3.2.2 TSS
3.3 Diagnostics

Appendices

A. BCPL Reserved Words and Tokens

B. ASCII Conversion Tables

C. Run-Time Environment

D. Syntax of BCPL

E. Programmers Checklist (to be supplied)

May 1971

CHAPTER 1 - GENERAL INFORMATION

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 either of the Computer Center's CDC 6400's.

1.1 LANGUAGE CHARACTERISTICS

The syntax is extremely rich, allowing a variety of ways to write conditional branches, loops, and subroutine definitions, and thereby permitting the construction of 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 feature is helpful for multi-programming or applications where recursion is useful (e.g., tree-processing).

1.2 PROPERTIES OF BCPL

1.2.1 Program

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

1.2.2 Elements

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

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

A number is a sequence of digits. An octal constant begins with octal digits followed by 'B'. The reserved word TRUE denotes -0=777...... 777B (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 '"' double quotes. The internal character set is ASCII. The actual character " can be represented in a string constant only by the pair *" 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.)
*Onnn	represents the octal character code nnn where nnn is three octal digits.

A string is represented as a sequence of 7 bit bytes. In the last word the characters are left justified, followed by a 140B.

56 49 42 35 28 21 14 7 0

	A		S	T	R	I	N	G
		I	N		M	E	M	O
	R	Y	0140					

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 single quotes. The actual 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'= 41B and '1C'=4243B.

1.2.3 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 be 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 t0, t1, t2, etc. represent table constants (constant expressions, string constants, or parenthesized table expressions).

Kind of Expression	Expression	Description
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 v 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 t0,t1,t2, ...	

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 2.1.5, after the construct block has been described.

1.2.4 Constant expression

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

1.2.5 Table expression

The value of a TABLE expression is the address of a static vector of cells initialized to the values of the TABLE constants t0, t1, ... Thus table is closely analogous to a string constant .

A TABLE constant is a string constant or a constant expression or a table expression enclosed in parentheses.

1.2.6 Operators

Arithmetic operators

There are two kinds of addition and subtraction: short and long. The long version is symbolized by suffixing a period to the symbol for the short version: i.e., the short operations are written

```
        e1 + e2             e1 - e2
while the long are written
        e1 +. e2            e1 -. e2
```

A short operation is undefined if the absolute value of either of its operands or its result is greated than 2^{17}. The long operations are defined for any 60-bit quantities.

Under addition and subtraction -0 (=7....7B) behaves like +0.

In general, multiplication, division and remainder are defined only when the operands and results are less than 2^{48} in absolute value[1]. The behavior of -0 is undefined.

The integer remainder (modulus) operator is defined as:

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

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.[2]

Relational operators

As with addition and subtraction, there are two kinds of relational operators, short and long, which are symbolized in the same manner. That is, the long version is obtained by suffixing a period to the short version (e.g., =., ≠., <., >., ≤., ≥.).

A relational expression of the form

```
e1 R1 e2 R2 e3 ... e(n-1) R(n-1) e(n)
```

is equivalent to

```
e1 R1 e2 ∧ e2 R2 e3 ∧ ... ∧ e(n-1) R(n-1) e(n)
```

The result of relations involving -0 is undefined.

Logical operators

The effect of a logical operator depends on its 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; any positive value means FALSE and any negative value means TRUE. Each subexpression is interpreted, from left to right, in truth-value context until the truth or falsehood of the expression is determined. Then evaluation stops. Thus

----

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

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

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:

		operator			
operand		∧	∨	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

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.

```
                  cell for ID
ID ---          ┌─────────────┐
                └─────────────┘
```

The contents 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 (or $). 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.

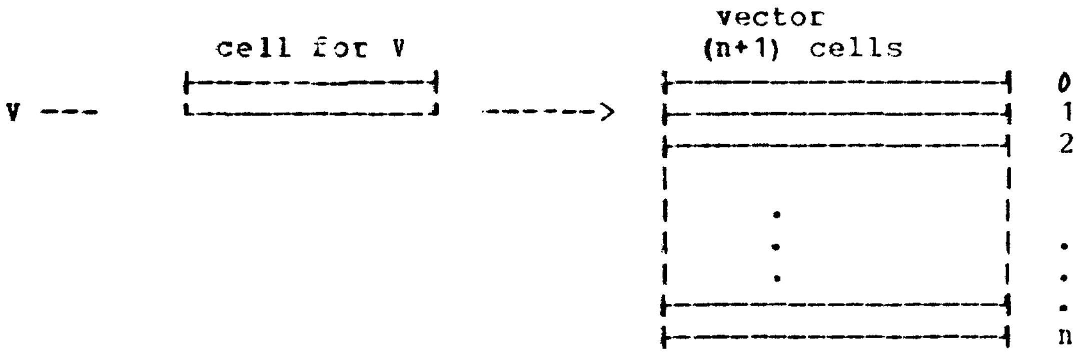
An address may be generated by means of the operator LV. The expression

LV e1

is valid only if e1 is

(1) an identifier, in which case the value of LV ID is the address of ID. (Note: e1 may not be an external name.)
(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 contents 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.

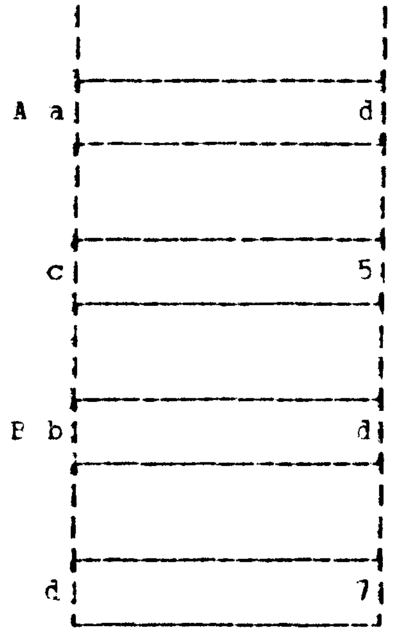
Thus, RV forces one more contents-taking than is normally demanded by the context.

As a summarizing example, consider the memory configuration depicted below.

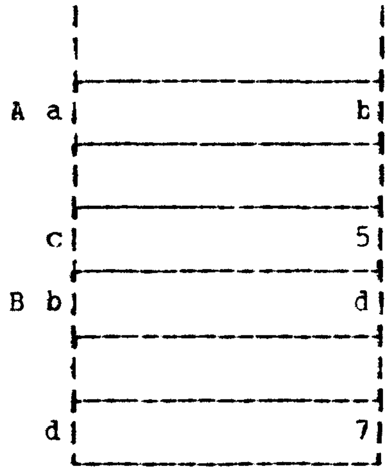
```
         |  .           |
         |  .           |
         |  .           |
         |--------------|
  A....a |             c|
         |--------------|
         |  .           |
         |  .           |
         |  .           |
         |--------------|
       c |             5|
         |--------------|
         |  .           |
         |  .           |
         |  .           |
         |--------------|
  B....b |             d|
         |--------------|
         |  .           |
         |  .           |
         |  .           |
         |--------------|
       d |             7|
         |--------------|
         |  .           |
         |  .           |
         |  .           |
```

a and b are the address of A and B respectively. Then each of the following assignments induces the memory configuration shown adjacent,

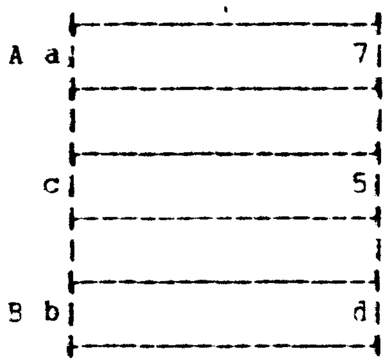
A := B

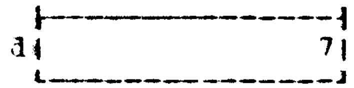


A := LV B

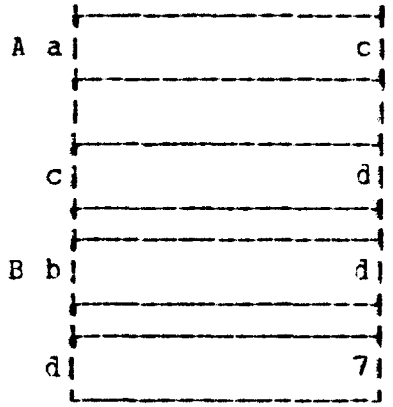


A := RV B





RV A := B



Note that

LV A:= B

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

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.

1.2.7 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 non-alphanumeric 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.

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.

CHAPTER 2 - STRUCTURE OF BCPL

2.1 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 2.3).

The pair of reserved words DO and THEN are synonymous.

The complete set of commands is shown here, with e, e1, e2 and e3 denoting the expressions and c, c1, and c2 denoting commands:

Type of Command	Command
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 c
switchon	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 2.2.7.

2.1.1 Assignment command

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
```

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
```

2.1.2 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.

2.1.3 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]
```

2.1.4 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
```

delcares a new cell with identifier ID (see Section 2.2.4).

Note that t is a new identifier not occurring in c.

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

2.1.5 RESULTIS command, and value blocks

The expression

```
VALCF [.....]
```

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

```
PESUITIS e
```

is encountered. The expresison 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.

2.1.6 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.

2.1.7 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.

2.2 DECLARATIONS

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

2.2.1 GLOBAL declaration

A BCPL program need not be compiled in one piece. The global vector provides a means of communication between separately compiled segments of a program. 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;... ]
```

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.

2.2.2 MANIFEST Declaration

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=777B;NMASK= ¬MASK]
```

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

```
MANIFEST [MASK=777B]
MANIFEST [NMASK= ¬MASK]
```

will declare NMASK as ¬777B.

A MANIFEST constant, like any constant, does not have an address. MANIFEST declarations may be combined exactly like GLOBAL declarations.

2.2.3 EXTERNAL Declaration

An identifier may be associated with ENTRYs declared in other (independently compiled) programs by the declaration:

```
EXTERNAL [ name ]                 or
EXTERNAL [ name = string constant ].
```

In the first case the string-constant is assumed to be the same as the *name*. The scope of the *name* is the same as for a global declaration. EXTERNAL declarations may be combined in the same manner as MANIFEST or GLOBAL declarations. Every routine and function declared in a BCPL program is declared an ENTRY (first seven characters only).

2.2.4 Dynamic cell declaration

The declaration

```
LET n1,n2,..., = e1,e2,...
```

creates n dynamic cells (words) and associates them with the identifiers n1,n2,... . These names are known in the remainder of the block containing the LET delcaration. They are also known in the expressions e1,e2,... . 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;... ]

A is known in e1,e2,c1,c2,...
B is known in e2,c1,c2, ...
F is known in e3,c1,c2,...
```

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.

2.2.5 Vector declaration

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. 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.

2.2.6 Function declaration

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.

The first 7 characters of N will be declared as an ENTRY and are thus accessible to other programs using appropriate EXTERNAL declarations.

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)
```

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.

2.2.7 Routine declaration

The declaration

```
LET N(p1,p2,...,pn) BE block
```

is identifical 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 the 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.

2.2.8 Label declaration

A label is declared by

<u>name</u>:

A label declaration may precede any command or label declaration, but may not precede any other form of declaration.

The scope of a label identifier is different from any other declaration, because it includes all of the smallest enclosing routine body, function body, or LET block (whichever is smallest), 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. Transferring to a label after the block in which it was declared has been left will produce chaotic (undefined) results.

2.2.9 Simultaneous declarations

Any declaration of the form

```
LET-----
```

may be followed by one or more delcaraticns 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.

2.3 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 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.

2.3.1 GET command

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 TSS the string is interpreted as a directory entry. Under SCOPE, the first seven characters of the string are used as the file name.

2.3.2 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.

2.3.3 DO or THEN

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 [
```

2.3.4 Semicolon

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 [
```

As an 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
```

2.4 ADDITIONAL FEATURES

2.4.1 Assembly Language Coding (to be supplied)

2.4.2 Library of Subprograms

Input/Ouput Routines

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

INITIALIZEIO(Y,SIZE) is a routine that sets up a free-space 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).

FINDINPUT(LFN) is a function taking a string constant file name (LFN) and returning a *stream-pointer* to be used by the input routines. FINDINPUT initializes an input buffer and attempts to read a buffer-load of the named file. If no information is found, an error occurs.

CREATEOUTPUT(LFN) is a function taking a 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 (= 144B) 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) write N words from V.0,...,V.(N-1) onto STREAM. Mixing calls of WRITEVEC and WRITCH on the same stream produces undefined results.

ENDREAD(STREAM) closes the file 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 it was 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 makes a standard exit.

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.

WRITES(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) write the number N (in octal) onto the output stream OUTPUT.

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 READCH and ASCII.

C7TO6: points to an output stream; used by WRITEN and WRITEO.

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

CHAPTER 3 - USING BCPL

3.1 HOW TO WRITE A BCPL PROGRAM (TO BE SUPPLIED)

3.2 HOW TO COMPILE BCPL

A field length of 45,000B 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.

3.2.1 Using BCPL under SCOPE

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.)

Compiling

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,CR.
```

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

	Default	
Parameter	Value	Use
I	INPUT	Designates the fileset containing the source code to be compiled. If the fileset appears to be empty, it is rewound and tried again. The source deck is terminated by an end of record.
L	OUTPUT	Designates the fileset 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 fileset on which the relocatable binary will be written. B=0 suppresses the output of binary.
C	0	Designates the fileset 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 fileset to be used for transmitting an intermediate object code between passes of the compiler. This fileset 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 fileset 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 be double-spaced.

CR Check reentrant.

3.2.2 Using BCPL under TSS

The compiler is invoked by typing BCPL to the Command Processor. The compiler then waits for lines of the form:

I=input;B=binary;C=compass;O=ocode;N=name;T;CR

All parameters are optional and may appear in any order. If the parameters input or compass are TTY, the input is taken from the teletype or the COMPASS program printed on the teletype, respectively. Except for T, the meanings of the parameters are the same as above but the default values are as follows:

I	TTY	
B	'B'	(i.e., BINPUT is I=INPUT or BBCPL if I=TTY)
C	O	
O	OCODE	
N	Same as I	BCPL if I=TTY

T causes the compilation times to be printed.

After each compilation BCPL waits for another line and exits when FIN is typed.

3.3 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 up arrow is printed under the last character read in before the error became apparent. A brief description of the error is printed. Only one error (the first) on any given line is reported. Errors reported on lines following the one containing 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.

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.

TTY	Standard	Multiple Punch	Alternate
	AND		
SHIFT N	↑	11-5-8	ASHIFT
	BE		
	BREAK		
	CASE		
	DO		THEN
	DEFAULT		
	END	7-8-9	
	=		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
	LSHIFT		
	LV		
	MANIFEST		
#	≠	(apostrophe)	NE
#.	≠.		LNE
	NEQV		
\	¬	12-6-8	NOT
	OR		ELSE
	REM		MOD
	REPEAT		
	REPEATUNTIL		
	REPEATWHILE		

TTY	Standard	Multiple_Punch	Alternate
	RESULTIS		
	RETURN		
	SHIFT		
$	RV		
	SWITCHON		
	TABLE		
	TEST		
	TO		
	TRUE		
	UNLESS		
	UNTIL		
	VEC		
	VALOF		
	WHILE		
	+		
	+.		
	-		
	S.		
	*		
	/		
	,		
	.		
	;	12-8-7	
	:	2-8	
	(		
	)		
SHIFT K	]	8-7	
SHIFT M	[	0-8-2	
←	:=		
	->	0-8-5	->
	.....B		

A string constant is delimited by double quotes (0-6-8) and a character constant by single quotes (11-6-8).

APPENDIX B. Graphic TTY Character Representation

ASCII Char	Printer Graphic	TSS ASCII Code	ASCII Char	Printer Graphic	TSS ASCII Code	ASCII Char	Printer Graphic	TSS ASCII Code
blank	blank	0	@	¬	40	'	≠	100
!	≤	1	A	A	41	a	A	101
"	↓	2	B	B	42	b	B	102
#	≡	3	C	C	43	c	C	103
$	$	4	D	D	44	d	D	104
%	%	5	E	E	45	e	E	105
&	∧	6	F	F	46	f	F	106
'	≠	7	G	G	47	g	G	107
(	(	10	H	H	50	h	H	110
)	)	11	I	I	51	i	I	111
*	*	12	J	J	52	j	J	112
+	+	13	K	K	53	k	K	113
,	,	14	L	L	54	l	L	114
-	-	15	M	M	55	m	M	115
.	.	16	N	N	56	n	N	116
/	/	17	O	O	57	o	O	117
0	0	20	P	P	60	p	P	120
1	1	21	Q	Q	61	q	Q	121
2	2	22	R	R	62	r	R	122
3	3	23	S	S	63	s	S	123
4	4	24	T	T	64	t	T	124
5	5	25	U	U	65	u	U	125
6	6	26	V	V	66	v	V	126
7	7	27	W	W	67	w	W	127
8	8	30	X	X	70	x	X	130
9	9	31	Y	Y	71	y	Y	131
;	;	32	Z	Z	72	z	Z	132
:	:	33	[	[	73	{	(	133
<	<	34	\	∨	74	|	blank	134
=	=	35	]	]	75	}	)	135
>	>	36	^	↑	76	~	blank	136
?	≥	37	_	->	77	rubout	blank	137

Table 2

Non-Graphic TTY Character Representation

Character	Internal ASCII Representation	Key Combination Systext Representation	Function
NUL	140	%@	
SOH	141	%A	
STX	142	%B	
ETX	143	%C	
EOT	144	%D	
EN	145	%E	
ACK	146	%F	
BEL	147	%G	Bell
BS	150	%H	Backspace
HT '*T'	151	%I	Horizontal Tab
LF	152	%J	Line Feed
VT	153	%K	Vertical Tab
FF	154	%L	Page Eject
CR '*N'	155	%M	
SO	156	%N	
SI	157	%O	
DLE	160	%P	
DC1	161	%Q	
DC2	162	%R	
DC3	163	%S	
DC4	164	%T	
NAK	165	%U	
SYN	166	%V	
ETB	167	%W	
CAN	170	%X	
EM	171	%Y	
SUB	172	%Z	
ESC	173	%]	
FS	174	%	
GS	175	%[	
RS	176	%	
US	177	%←	

May 1971

APPENDIX C The Run-Time Environemnt

Storage Allocation and Register Usage

Unlike any programs loaded by the SCOPE loader, BCPL object programs begin at 100B. The last program is followed by the first word of blank common. Under normal conditions a special program HEAD should be loaded before any BCPL programs. The following registers are used by BCPL programs:

B1	always contains 1
B2	always contains -1
B3	scratch register, used for all calls
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:

```
      0  ┌──────────────────────────────┐
         │    Communication area        │
         ├──────────────────────────────┤
         │    HEAD                      │
         ├──────────────────────────────┤
         │    BCPL                      │
         │                              │
         │    PROGRAMS                  │
         │                              │
         ├──────────────────────────────┤
   B5 -> │                              │ <- GLOBALO
         ├──────────────────────────────┤
         │    GLOBAL AREA               │
         ├──────────────────────────────┤
   B6 -> │    STACK AREA                │
         └──────────────────────────────┘
```

APPENDIX D. Syntax of BCPL

The notation used below in defining the syntax of BCPL is defined as follows:

1. A class of elements is represented by a notation variable, consisting of underlined lower case letters.
2. Literal characters are represented by upper case letters or special characters.
3. The vertical bar denotes an alternative.
4. Braces { } denote a repeatable group.
5. Three dots ... denote optional repetition of the immediately preceding syntactic unit.
6. The following are not represented:
   - a. comments,
   - b. block delimited tags,
   - c. graphic escape sequences allowable in strings,
   - d. allowable dropping of ; and DO according to preprocessor rules,
   - e. synonyms for certain system words and operators (see Appendix A),
   - f. restriction on usage of certain constructions in certain contexts,
   - g. required blanks.

1. Identifiers, Strings, Numbers

```
null::=
letter::=A|B|...|Z
octal digit::= 0|1|...7
digit::= octal digit |8|9
string constant[1]::= "characters ≤ 128"
character constant::= 'character ≤ 8'
octal number::= octal digit...B
number::= octal number | digit ,,,
identifier[2] ::= letter {letter|digit}...
```

2. Operators

```
addressop ::= LV | RV
multop ::= * | / | REM
addop ::= + | - | +. | -.
shiftop ::= LSHIFT | RSHIFT
```

May 1971

```
relop ::= = | ≠ | < | > | ≤ | ≥ | =. | ≠ | <. | >. | ≤. | ≥.
equop ::= EQV | NEQV
```

3. Expression

```
element    ::=    character___constant|string___constant|number|
               identifier|TRUE|FALSE
primary__expr  ::=  (expression)|VALOF  block|element|primary_expr
                   (expression list)
vector_expr ::= primary_expr ...
address_expr[3] ::= vector_expr|addressop address_expr
mult_expr ::= address_expr|mult_expr multop address_expr
add_expr ::= mult_expr|add_expr addop mult_expr
shift_expr ::= add_expr|shift_expr shiftop add_expr
rel_expr ::= shift_expr {relop shift_expr} ...
not_expr ::= rel_expr| ¬not_expr
and_expr ::= not_expr { not_expr} ...
or_expr ::= and_expr {and_expr} ...
eqv_expr ::= or_expr {eqv_op or_expr} ...
conditional ::= eqv_expr|eqv_expr -> conditional,conditional
expression[4] ::=  conditional|TABLE  constant__expression{,constant
                 expression} ...
```

4. Lists of Expression and Identifiers

```
exp -list ::= expression|expression, exp-list
expression-list ::= null|exp-list
n-list ::= identifier|identifier, n-list
name-list ::= mull|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]
external-item ::= identifier|identifier = string constant
external-list ::= external-item|external-item ; external-list
external-declaration ::= EXTERNAL[external-list]
simple-definition ::= 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

```
LHSE ::= identifier|vector-application|RV address expr
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 string-constant
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
```

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 1.2.2 for further restrictions on strings 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 1.2.6 Addressing__Operators.

[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.

[5] The lengths of the two lists must be equal.