BCPL/360 REFERENCE MANUAL

John R. Kelly
Thomas R. Wilcox

February 1970

ABSTRACT

BCPL is a simple recursive programming language designed for compiler writing and system programming: it was derived from true CPL (Combined Programming Language) by removing those features of the full language which make compilation difficult namely, the type and mode matching rules and the variety of definition structures with their associated scope rules.

ACKNOWLEDGEMENTS

BCPL was originally developed by Martin Richards. The authors wish to thank Mr. Richards for making available his IBM 7094 implementation of BCPL and to thank the Office of Computer Services of Cornell University for supporting the development of our IBM S/360 implementation. Sections 1-7 of this manual are revisions of [1]. This manual was printed by a text editor programmed in BCPL/360 by J. Kelly.

TABLE OF CONTENTS

1.0 Introduction

BCPL is a general purpose recursive programming language which is particularly suitable for large non-numerical problems in which machine independence is an important factor. It was originally designed as a vehicle for compiler construction and has, so far, been used in four compilers. BCPL is currently implemented and running on an IBM S/360 under OS, a 7094 under Project MAC's CTSS, a GE 635 under GECOS, and a KDF 9 at Oxford. Other implementations are under construction for MULTICS, the ICT 1900 series, and Atlas.

Sections 1-7 of this manual describe the basic features of BCPL in an essentially machine independent manner. However the exact specifications of the preprocessing rules, and occasionally the syntax and semantics, apply only to the BCPL/360 implementation of the language. Sections 8-10 describe the various compiler and loader options and the run time support routines available in BCPL/360.

2.0 BCPL syntax

The syntactic notation used in this manual is basically BNF with the following extensions:

1) The symbols E, D, and C are used as shorthand for <expression>, <definition>, and <command>.

2) The metalinguistic brackets '<' and '>' may be nested and thus used to group together more than one constituent sequence (which may contain alternatives). An integer subscript may be attached to the metalinguistic bracket '>' and used to specify repetition; if it is the integer n, then the sequence within the brackets must be repeated at least n times; if the integer is followed by a minus sign, then the sequence may be repeated at most n times or it may be absent.

2.1 Hardware Syntax

The hardware syntax is the syntax of an actual implementation of the language and is therefore necessarily implementation dependent since it depends on the character set that is available. To simplify the description of any implementation of BCPL a canonical syntax has been defined and this is given in section 2.2. The canonical representation of a BCPL program consists of a sequence of symbols from the following set.

2.1.1 BCPL Canonical Symbols

Throughout the rest of this manual words composed entirely of underlined small letters will be used as the names of canonical symbols. The names of all these symbols are given below together with their BCPL/360 hardware representations.

CANONICAL SYMBOL	BCPL/360 EQUIVALENT NOTATIONS
number	a) A decimal integer between -2^{29} (-536870912) and $2^{29}-1$ (536870911). b) A hexadecimal integer between 0 and 3FFFFFFF inside double quotes, i.e. "0" to "3FFFFFFF". c) A character constant consisting of a single character enclosed in single quotes, e.g. 'A', '3', '$', '*N', and '*''.
name	A sequence of up to 20 characters from the set {A,...,Z,@,_,0,...,9}, the first of which is not a digit. However the identifiers listed below are reserved words and can not be used as names.
stringconst	Up to 255 characters enclosed in single quotes (any of the 256 EBCDIC characters may be used). The following two character combinations are translated into single characters as follows: *N -> new line, *T -> tab, *S -> space (blank), *B -> backspace, *a -> a where a ≠ N, T, S, or B. Note that string constants which are separated by blanks will be concatenated into one string. This allows typing long strings on several cards.
true	TRUE, -1, "3FFFFFFF"
false	FALSE, 0
pos	+
neg	-
plus	+
minus	-
mult	*
div	/
rem	REM
eq	=, EQ
ne	¬=, NE
ls	<, LT
gr	>, GT, GR
le	<=, =<, ¬>, LE
ge	>=, =>, ¬<, GE
lshift	LS, LSHIFT
rshift	RS, RSHIFT
srshift	SRS
not	¬, NOT
logand	&, LOGAND
logor	|, LOGOR

eqv	==, EQV
neqv	¬==, EXOR, NEQV
rv	RV
lv	LV
vecap	.
cond	->, -*
valof	VALOF
table	TABLE
assign	:=
goto	GOTO, GO TO
if	IF
unless	UNLESS
test	TEST
do	DO, THEN
or	OR
for	FOR
to	TO
by	BY
while	WHILE
until	UNTIL
repeat	REPEAT
repeatwhile	REPEATWHILE
repeatuntil	REPEATUNTIL
break	BREAK
resultis	RESULTIS
return	RETURN
finish	FINISH
switchon	SWITCHON
into	INTO
case	CASE
default	DEFAULT
colon	:
semicolon	;
comma	,
let	LET
and	AND
manifest	MANIFEST
global	GLOBAL
be	BE
vec	VEC
sectbra	$, $tag
sectket	#, #tag
rbra	(
rket	)
sbra	[
sket	]
csect	SECTION
include	INCLUDE
prog	PROGRAM
end	end of deck, ENDSECTION

2.1.2 BCPL/360 Conventions and Preprocessor Rules

The Preprocessor is the part of the BCPL compiler which transforms the raw source text of a program into canonical symbols. This section describes the conventions used in the BCPL/360 Preprocessor.

1) Comments may be included in a program between a double vertical line '||' and the end of the line. Example:

```
LET R() BE || THIS ROUTINE REFILLS THE VECTOR SYMB
    FOR I = 1 TO 200 DO SYMB.(I) := READCH(INPUT)
```

2) Section brackets may be tagged with a sequence of alphanumeric characters. Two section brackets are said to match if their tags are identical. More than one section may be closed by a single closing section bracket since, on encountering a closing section bracket, if the current opening section bracket is found not to match then the current section is automatically closed by the insertion of an extra closing bracket. The process is repeated until the matching open section bracket is found. For example,

```
$1  UNTIL I = 0 DO
          $2 R(I)
             I := I - 1  #1
```

The final section bracket #1 does not match $2 and is, therefore, equivalent to #2 #1.

3) The canonical symbol semicolon is inserted by the Preprocessor between pairs of items if they appear on different lines and if the first is from the set of items which may end a command or definition, namely:

break, return, finish, repeat, sket, rket, sectket, name, stringconst, number, true, false

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

test, for, if, unless, while, until, goto, resultis, case, default, break, return, finish, switchon, sectbra, rbra, valof, rv, name

4) The canonical symbol do is inserted by the Preprocessor 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:

sket, rket, sectket, name, number, stringconst, true, false

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

test, for, if, unless, while, until, goto, resultis, case, default, break, return, finish, switchon

5) The source text occupies columns 1 to 72 inclusive and is completely free form except that: 1) canonical symbols such as names, numbers, and reserved words must be separated from one another by one or more spaces or by some other delimiter such as an operator (+, *, <, etc.), comma, parenthesis, etc., and 2) canonical symbols may not be split across card boundaries.

Example:

The following is a complete program segment for separate compilation; it is written in the BCPL/360 representation and exhibits some of the preprocessor rules. Note that it was not necessary to write a single semicolon since they will all be inserted automatically.

```
SECTION EXAMPLE

GLOBAL $ CHECKDISTINCT:100
         REPORT:101
         DVEC:102  #

LET CHECKDISTINCT(E,S) BE || TEST FOR MULTIPLY DEFINED NAMES
   $1 UNTIL E = S DO
         $ LET P = E + 4
           AND N = DVEC.(E)
           WHILE P < S DO
               $ IF DVEC.(P) = N DO REPORT(142,N)
                 P := P + 4  #
           E := E + 4   #1
```

2.2 Canonical Syntax

The syntax given in this section defines all the legal constructions in BCPL without specifying either the relative

binding powers or association rules of the command and expression operators. These are given later in the manual in the descriptions of each construction.

To improve the readability of the syntax a mixture of the canonical and the BCPL/360 hardware representations is used in this manual.

```
E          ::= <name> | <stringconst> | <number> | true |
               false | ( E ) | valof <block> | lv E | rv E |
               E ( E ) | E () | E . E | E <diadic op> E |
               <monadic op> E | E -> E, E | E < , E >0 |
               table <constant list>

<diadic op> ::= * | / | rem | + | - | = | ¬= | LT | GT |
                LE | GE | LS | RS | SRS | & | LOGOR |
                == | ¬==

<monadic op> ::= + | - | ¬

<constant> ::= restricted E

D          ::= D < and D >0 | <name> ( <name list>1- ) = E |
               <name> ( <name list>1- ) be C |
               <name list> = E |
               <name list> = vec <constant list>

<name list> ::= <name> < , <name> >0

<constant list> ::= <constant> < , <constant> >0

C          ::= E := E | E ( E ) | E () | goto E |
               <name> : < C >1- | <block> |
               if E do C | unless E do C |
               test E then C or C |
               while E do C | until E do C | C repeat |
               C repeatwhile E | C repeatuntil E |
               for <name> = E < to E | to E by E |
                    by E to E > do C |
```

```
                    break | return | finish | resultis E |
                    switchon E into <block> |
                    case <constant> : < C >1-  | default : < C >1-

<block body> ::= C < ; C >0  | <declaration>1  < ; C >0

<block> ::= $ <block body> #

<declaration> ::= let D < ; >1-  |
                      < manifest | global > $ <name> < = | : >
                           <constant>   < ; <name> < = | : >
                           <constant> >0  # < ; >1-  |
                      prog $ <name list> # < ; >1-

<control section> ::= csect <name> < ; >1-  <block body>

<program> ::= < <control section> end >1
```

3.0 Data Items

3.1 Rvalues, Lvalues, and Data Items

An RVALUE is a binary bit pattern of a fixed length (which is 30 bits in the BCPL/360 implementation). Rvalues may be used to represent a variety of different kinds of objects such as integers, truth values, vectors, or functions. The actual kind of object represented is called the TYPE of the Rvalue.

A BCPL expression can be evaluated to yield an Rvalue but its type remains undefined until the Rvalue is used in some definitive context and it is then assumed to represent an object of the type required by the context. For example, in the following function application

```
(B.(I) -> F, G) (1,Z(I))
```

the expression (B.(I) -> F, G) is evaluated to yield an Rvalue which is then interpreted as the Rvalue of a function since the expression occurs in the operator position of a function application; whether F and G are in fact functions is not explicitly checked. Similarly the expression B.(I) (which is a vector application) occurs where boolean is expected and so its Rvalue is interpreted as a truth value.

There is no explicit check to ensure that there are no type mismatches.

An LVALUE is a bit pattern representing a storage location containing an Rvalue. An Lvalue is the same size as an Rvalue and is a type in BCPL. There is one context

where an Rvalue is interpreted as an Lvalue and that is as the operand of the monadic operator rv. For example, in the expression

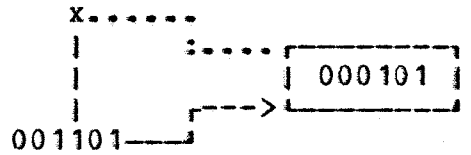
rv F(I)

the expression F(I) is evaluated to yield an Rvalue which is then interpreted as an Lvalue since it is the operand of rv. The application of rv on this Lvalue yields the Rvalue which is contained in the location represented (or referred to) by the Lvalue.

An Lvalue may be obtained by applying the operator lv to an identifier, a vector application, or an rv expression (see section 4.9).

A DATA ITEM is composed of an Lvalue, the referenced Rvalue, and possibly an associated identifier. The term is used loosely to mean the Lvalue, Rvalue, or identifier depending on context.

In BCPL a non-global data item may have at most one identifier. The following diagram shows a data item for a six bit machine.



The dotted line indicates the semantic association between an identifier x and the storage location which is represented by the box; the dashed arrow shows the correspondence between the Lvalue 001101 and the Rvalue

000101, and the dashed line shows the correspondence between the identifier x and the Lvalue 001101.

It is meaningful to say that the Lvalue of the data item x is the bit pattern 001101, that the Rvalue of the data item x is the bit pattern 000101 and that the Lvalue 001101 refers to the data item x.

3.2 Types and Representations

An Rvalue may represent an object of one of the following types:

integer, logical, Boolean, function, routine, label, string, vector, and Lvalue.

Although the bit pattern representations of each type is implementation dependent certain relations between types are not.

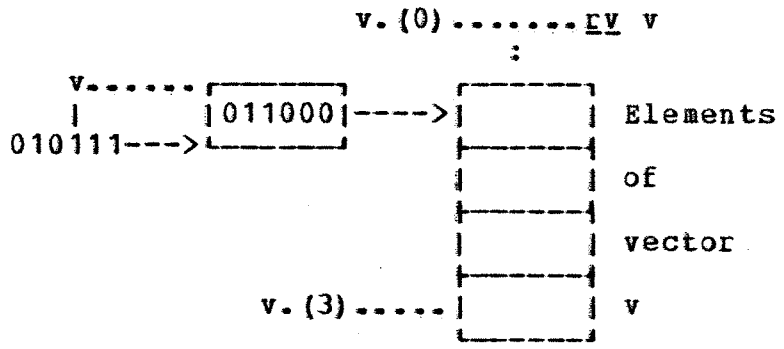
1) The Rvalue of a vector v, say, is identical to the Lvalue of its zeroth element:

v and lv v.(0) have the same Rvalue

and also

rv v and v.(0) have the same Rvalue

The following diagram illustrates a vector for a six bit machine:



2) The Lvalue of the nth element of a vector v may be obtained by adding the integer n to v; thus

   lv v.(n) is equal to v + n

3) If x, y, and t are the first, second, and nth parameters of a function or routine and if v = lv x, then

```
v.(0) = x,
v.(1) = y,
v.(n-1) = t.
```

This property may be used to define functions and routines with a variable number of actual parameters. In the definition of such a function or routine it is necessary to give a formal parameter list which is at least as long as the longest actual parameter list of any call for it.

Example:

The following definition

```
LET R(A,B,C,D,E,F) BE
    $ LET V = LV A
      - - - -
      - - - -  #
```

defines the routine R which may be called with six or fewer actual parameters. During the execution of the routine, the variable V may be used as a vector whose first n elements

are the first n actual parameters of the call; thus during the following call

R(126,36,18,99)

the initial Rvalues of

V.(0), V.(1), V.(2), V.(3)

are

126, 36, 18, 99.

The Rvalue of a label is a bit pattern representing the program position of the labelled command. Note that it does not contain information about the activation level of the function or routine in which the label occurred.

The Rvalue of a function or routine is a representation of the entry point of the function or routine.

4.0 Primary Expressions

A primary expression is any expression described in this section.

4.1 Names

Syntax: A name is a sequence of one or more characters from a restricted alphabet called the name character alphabet. The hardware representation of characters in this alphabet and the rules for recognizing the start and end of names are implementation dependent. In BCPL/360 the name character alphabet is the set {A,...,Z,@,_,0,...,9}; the first character of a name can not be a digit.

Semantics: Two names are equal if they have the same sequence of name alphabet characters. A name may always be evaluated to yield an Rvalue. If the name was declared to be a manifest constant (see section 7.4) then the Rvalue will be the same on every evaluation; if the name was declared in any other way then it is a variable and its Rvalue may be changed dynamically by an assignment command. If N is a variable then its Lvalue is the Rvalue of the expression:

lv N

4.2 String Constants

Syntax: '<string alphabet character>$_0$'

The hardware representation of characters in the string alphabet is implementation dependent. In BCPL/360 the entire EBCDIC character set may be used. However the characters ' and * must be represented by *' and ** respectively. In addition

*N represents new line,

*S represents space,

*B represents backspace,

*T represents tab,

*a represents a where a ≠ N, S, B, or T.

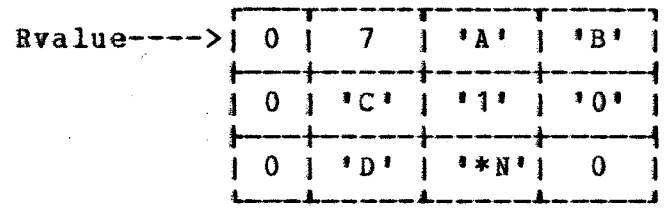
Semantics: A string constant of length one has an Rvalue which is the bit pattern representation of the character (8 bits for EBCDIC); this is right justified and filled with zeroes, and hence is a number (see section 4.3) in the range 0-255.

A string constant with length other than one is represented as a BCPL vector; the length and the string characters are packed in successive words of the vector (three characters per word in BCPL/360) with zero fill in all the unused bits. The Rvalue of a packed string constant is a

pointer to the zeroth word (which contains the length and first two characters) of the vector.

Example:

The string 'ABC10D*N' is represented as follows:



4.3 Numerical Constants

Syntax: <digit>$_1$ or "<hex digit>$_1$" or '<string alphabet character>'

Semantics: The sequence of digits is interpreted as a decimal integer in the first case, as a right justified hexadecimal integer in the second case, and as a character constant (see section 4.2) in the last case. In BCPL/360 the word size is 30 bits. Hence decimal numbers range from -2^{29} (-536870912) to $2^{29}-1$ (536870911) while hexadecimal numbers range from "0" to "3FFFFFFF" where A-F represent 10-15.

4.4 True and False

Syntax: true or false

Semantics: The Rvalue of true is a bit pattern entirely composed of ones; the Rvalue of false is zero. Note that true = ¬false.

4.5 Bracketted Expressions

Syntax: (E)

Semantics: Parentheses may enclose any expression; their sole purpose is to specify grouping.

4.6 Result Blocks

Syntax: valof <block>

Semantics: A result block is a form of BCPL expression; it is evaluated by executing the block until a resultis statement is encountered, which causes execution of the block to cease and returns the Rvalue of the expression in the resultis command (see section 6.16).

4.7 Vector Applications

Syntax: E1 . E2

The dot is necessary to distinguish a vector application from a function application. E1 is a primary expression. The vector application operator is less binding than the implied function application operator, but more binding than

all other dyadic operators. It associates to the left. (Note: other implementations may use E1*(E2) or E1*[E2] to denote vector applications.)

Semantics: A vector is represented by a pointer to a consecutive group of words which are the elements of the vector. The pointer points to the zeroth element. To obtain the Rvalue of a vector application, E1 and E2 are evaluated to yield two Rvalues. The first is interpreted as a vector pointer and the second as the subscript. The element is then accessed to yield the result.

The Lvalue of an element may be obtained by evaluating the expression

lv E1.(E2)

The representations of vectors, Lvalues, and integers is such that the following relations are true:

E1.(E2) = rv (E1 + E2)

lv E1.(E2) = E1 + E2

4.8 Function Applications

Syntax: E0 (E1, E2, ..., En)

E0 is a primary expression. Note that the parentheses are required even for an empty argument list. The implied function

application operator is the most binding dyadic operator and associates to the left. (Other implementations may use [and] instead of parentheses.)

Semantics: The function application is evaluated by evaluating the expressions E0, E1, ..., En and assigning the Rvalues of E1, ..., En to the first n formal parameters of the function whose Rvalue is the value of E0; this function is then entered. The result of the application is the Rvalue of the expression in the function definition (see section 7.7).

4.9 Lv Expressions

Syntax: lv E

E is a primary expression.

Semantics: The Lvalue of some expressions may be obtained by applying the operator lv; it is only meaningful to apply lv to a vector application, an rv expression, or an identifier which is not a manifest constant.

The result of the application depends on the leading operator of the operand as follows:

a) A vector application.

The result is the Lvalue of

the element referenced (see section 4.7).

b) An rv expression.

The result is the value of the operand of rv. The following relation is always true:

lv rv E = E

c) A name.

The result is the Lvalue of the data item with the given name (which must not be a manifest constant). If the name was declared explicitly as a function, routine, global, or label then its Lvalue is constant (but its Rvalue is not, see section 7.2).

4.10 Rv Expressions

Syntax: rv E

E is a primary expression.

Semantics: The value of an rv expression is obtained by evaluating its operand to yield an Rvalue which is then interpreted as the Lvalue of a data item. The result is the Rvalue of this data item.

5.0 Compound Expressions

5.1 Arithmetic Expressions

Syntax: E1 * E2 or E1 / E2 or E1 rem E2 or E1 + E2 or +E1 or E1 - E2 or -E1

The operators * / and rem are more binding than + and - and associate to the right. The operators + and - associate to the left.

Semantics: All these operators interpret the Rvalues of their operands as signed integers, and all yield integer results.

The operator * denotes integer multiplication.

The division operator / yields the correct result if E1 is divisible by E2; otherwise for BCPL/360 it truncates towards zero.

The operator rem yields the remainder of E1 divided by E2; for BCPL/360 the remainder has the same sign as the dividend E1.

The operators + and - denote addition and subtraction respectively.

5.2 Relational Expressions

Syntax: E1 <relop> E2 < <relop> En >$_0$

where <relop> ::= = | ¬= | < | > | <= | >=
and n ≥ 2 (see section 2.1.1 for equivalent symbols).

The relational operators are less binding than the arithmetic operators.

Semantics: The result of evaluating an extended relation is true if and only if all the individual relations are true. The order of evaluation is undefined. The Rvalues of the expressions E1,...,En are interpreted as signed integers and the relational operators have their usual mathematical meanings.

5.3 Shift Expressions

Syntax: E1 LS E2 or E1 RS E2 or E1 SRS E2

E2 is any primary or arithmetic expression and E1 is any shift, relational, arithmetic, or primary expression. Thus the shift operators are less binding than the relations on the left and more binding on the right.

Semantics: The Rvalue of E1 is interpreted as a logical bit pattern and that of E2 as an integer. The result of E1 lshift E2 is the bit pattern E1 shifted to the left by E2 places. E1 rshift E2 is as for lshift but shifts to the right. Vacated positions are

filled with zeros and the result is undefined if E2 is negative or greater than the data item size (30 bits).

The operator srshift is peculiar to BCPL/360. BCPL assumes word addressing, as illustrated by the rule

lv E1.(E2) = E1 + E2

(see section 4.7), whereas the S/360 hardware does byte addressing. Hence in BCPL/360 integers use only the high order 30 bits of a S/360 32 bit word. The low order 2 bits are normally forced to zeroes when necessary. srshift is a right shift for which the two low order bits are not reset to zeroes. Note that lshift may be used to shift the two low order bits into the 30 bit field.

5.4 Logical Expressions

Syntax: ¬E1 or E1 & E2 or E1 | E2 or

E1 == E2 or E1 ¬== E2

The operator ¬ is most binding; then, in decreasing order of binding power are:

& , | , == , ¬== .

All the logical operators are less binding than the shift operators.

Semantics: The operands of all the logical

operators are interpreted as binary bit patterns of ones and zeros.

The application of the operator ¬ yields the logical negation of its operand. The result of the application of any other logical operator is a bit pattern whose nth bit depends only on the nth bit of the operands and can be determined by the following table.

The values of the nth bits	Operator &	|	==	¬==
both ones	1	1	1	0
both zeros	0	0	1	0
otherwise	0	1	0	1

5.5 Conditional Expressions

Syntax: E1 -> E2, E3

E1, E2, and E3 may be any logical expressions or expressions of greater binding power. E2 and E3 may, in addition, be conditional expressions.

Semantics: The value of the conditional expression E1 -> E2, E3 is the Rvalue of E2 or E3 depending on whether the value of E1 represents true or false respectively. In either case only one alternative is evaluated. If the value of E1 does not

represent either true or false then the result of the conditional expression is undefined in general. However for BCPL/360 it will be assumed to be true.

5.6 Tables

Syntax: table E0, E1, ..., En

where all the expressions are more binding than comma; the expressions may be composed of constants and any operator except srshift, rv, and lv.

Semantics: A table is a static vector whose elements are initialized prior to execution to the values of the expressions E0 to En. The Rvalue of a table is a pointer to its zeroth element. Each of the expressions must evaluate to an integer, string constant, label, or table, and the item placed in the table is respectively the integer, the Rvalue of the string constant, the Lvalue of the label variable, or the Rvalue of the table.

6.0 Commands

6.1 Simple Assignment Commands

Syntax: E1 := E2

Semantics: E1 may either be an identifier, a vector application, or an rv expression, and its effect is as follows:

a) If E1 is an identifier:

The identifier must refer to a data item which has an Lvalue (i.e., it must not be declared as a manifest named constant). The assignment replaces the Rvalue of this data item by the Rvalue of E2.

b) If E1 is a vector application:

The element referenced by E1 is updated with the Rvalue of E2.

c) If E1 is an rv expression:

The operand of rv is evaluated to yield a value which is then interpreted as an Lvalue; the Rvalue of E2 then replaces the Rvalue of the data item referred to by the Lvalue.

6.2 Assignment Commands

Syntax: L1, L2, ..., Ln := R1, R2, ..., Rn

Semantics: The semantics of the assignment command is defined in terms of the simple assignment command; the command given above is semantically equivalent to the following sequence:

```
L1  :=  R1
L2  :=  R2
.   .   .   .
Ln  :=  Rn
```

Note that the individual assignments are executed from left to right and not simultaneously.

6.3 Routine Commands

Syntax: E0 (E1, E2, ..., En)

where E0 is a primary expression. Note that the parentheses are required even for an empty argument list. The implied routine (function) application operator is the most binding dyadic operator and associates to the left. (Other implementations may use [and] instead of parentheses.)

Semantics: The above command is executed by assigning the Rvalues of E1, E2, ..., En to the first n formal parameters of the routine whose Rvalue is the value of E0; this routine is then entered. The execution of

this command is complete when the execution of the routine body is complete.

6.4 Labelled Commands

Syntax: N: C where N is a name.

Semantics: This declares a data item with name N; its scope is the smallest textually enclosing routine or function body and its initial Rvalue is a bit pattern representing the program position of the command C. Its Lvalue is constant, and refers to a position in the global vector (see section 7.3) if and only if the labelled command occurs within the scope of a global with the same name as the label. The Rvalue of a label is initialized prior to execution of the program.

6.5 Goto Commands

Syntax: goto E

Semantics: E is evaluated to yield an Rvalue, then execution is resumed at the statement whose label had the same initial Rvalue.

6.6 If Commands

Syntax: if E do C

Semantics: E is evaluated to yield an Rvalue which

is then interpreted as a truth value. (See section 4.4 for the representation of Boolean values.) If the value of E represents neither true nor false then the effect is implementation dependent. For BCPL/360, a nonzero value is considered to be true.

6.7 Unless Commands

Syntax: unless E do C

Semantics: This statement is equivalent to the following:

```
if ¬(E) do C
```

6.8 While Commands

Syntax: while E do C

Semantics: This is equivalent to the following sequence:

```
    goto L
M:  C
L:  if E goto M
```

where L and M are identifiers which do not occur elsewhere in the program.

6.9 Until Commands

Syntax: until E do C

Semantics: This statement is equivalent to:

```
while ¬(E) do C
```

6.10 Test Commands

Syntax: test E do C1 or C2

Semantics: This statement is equivalent to the following sequence:

```
    unless E goto L
    C1
    goto M
L:  C2
M:
```

where L and M are identifiers which do not occur elsewhere in the program.

6.11 Repeated Commands

Syntax: C repeat or C repeatwhile E or

C repeatuntil E

where C is the textually shortest command to the left of the repeating symbol, i.e. where C is any command other than an if, unless, while, until, test, or for command.

Semantics: C repeat is equivalent to:

```
L: C
   goto L
```

C repeatwhile E is equivalent to:

```
L: C
   if E goto L
```

C repeatuntil E is equivalent to:

```
L: C
     unless E goto L
```

where L is an identifier which does not occur elsewhere in the program.

6.12 For Commands

Syntax:

```
for N = E1 to E2 do C  or
for N = E1 to E2 by E3 do C  or
for N = E1 by E3 to E2 do C
```

where N is a name. The last two forms may not be allowed in other implementations of BCPL.

Semantics: If the first form is used then E3 is assumed to be 1. If the step amount E3 is a negative constant then the for statement is equivalent to:

```
$ let N, T, B = E1, E2, E3
  until N < T do
      $ C
          N := N + B # #
```

Otherwise the step amount is assumed to be positive at run time, so that the for statement is equivalent to:

```
$ let N, T, B = E1, E2, E3
  until N > T do
      $ C
```

```
N := N + B # #
```

In either case T and B are identifiers which do not occur elsewhere in the program.

6.13 Break Commands

Syntax: break

Semantics: When this statement is executed it causes execution to be resumed at the point just after the smallest textually enclosing loop command. The loop commands are those with the following key words:

while, until, repeat, repeatwhile, repeatuntil, and for.

6.14 Finish Commands

Syntax: finish

Semantics: This causes the execution of the program to cease. A finish command is automatically appended to the end of each control section.

6.15 Return Commands

Syntax: return

Semantics: This causes a return from a routine body to the point just after the routine command which made the routine call. A

return command is automatically appended to the end of each routine.

6.16 Resultis Commands

Syntax: resultis E

Semantics: This causes execution of the smallest enclosing result block to cease and return the Rvalue of E.

6.17 Switchon Commands

Syntax: switchon E into <block>

where the block contains labels of the form:

case <constant> : or

default:

The constant may be any manifest expression (see section 7.4).

Semantics: 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.

In BCPL/360 the switch is implemented as a direct switch, a sequential search, or

a binary search depending on the number and range of the case constants.

6.18 Blocks

Syntax: $ <declaration>$_{1}$ < ; C >$_{0}$ # or

$ C < ; C >$_{0}$ #

Semantics: A block is executed by executing the declarations (if any) and then executing the commands of the block in sequence.

The scope of a declaration is the region of program consisting of the declaration itself, the succeeding declarations, and the command sequence.

7.0 Definitions and Declarations

7.1 Scope Rules

The SCOPE of a name N is the textual region of program throughout which N refers to the same data item. Every occurrence of a name must be in the scope of a declaration of the same name.

There are three kinds of declaration:

a) A formal parameter list of a function or routine: its scope is the function or routine body.

b) The set of labels set by colon in a routine or function body: its scope is the routine or function body.

c) Each declaration in the declaration sequence of a block: its scope is the region of program consisting of the declaration itself, the succeeding declarations, and the command sequence of the block.

Two data items are said to be declared at the same level of definition if they were declared in the same formal parameter list, as labels of the same routine or function body, or in the same declaration.

There are three semantic restrictions concerning scope rules, namely:

a) Two data items with the same name may not be declared in the same level of definition.

b) If a name N is used but not declared within the

body of a function or routine, then it must either be a manifest named constant or a data item with a manifest constant Lvalue, that is it must have been declared as a global, an explicit function or routine, or as a label. Thus the following program is illegal:

```
let x = 1
let f(y) = x + y
```

since x is a dynamic data item (see sections 7.2 and 7.5).

c) A label set by colon may not occur within the scope of a data item with the same name if that data item was declared within the scope of the label and was not a global. Thus the following program is illegal:

```
$1 let L = 0
L: <command> #1
```

7.2 Space Allocation and Extent of Data Items

The EXTENT of a data item is the time through which it exists and has an Lvalue. Throughout the extent of a data item, its Lvalue remains constant and its Rvalue is changed only by assignment.

In BCPL data items can be divided into two classes:

a) Static data items:

Those data items whose extents last as long as the program execution time; such data items have

constant Lvalues. Every static data item must have been declared either in a function or routine definition, in a global declaration, or as a label set by colon.

b) Dynamic data items:

Those data items whose extent is limited; the extent of a dynamic data item starts when its declaration is executed and continues until execution leaves the scope of the declaration. Every dynamic data item must be declared either by a simple definition, a vector definition, or as a formal parameter. The Lvalue of such a data item is not a constant.

7.3 Global Declarations

Syntax: global $ <name> : <constant>

< ; <name> : <constant> >$_0$ # <;>$_{1-}$

The constants may be any manifest expressions (see section 7.4).

Semantics: The global vector is the sole means of communication between separately compiled segments of a program. To call a function or routine which is declared in one segment from a position in another it is necessary to declare it as a global with the same value in each of the two segments.

The above declaration declares a set of

names to be global and allocates the positions in the global vector as defined by the manifest constants.

A global variable is a static data item and has an Lvalue which is constant.

7.4 Manifest Declarations

Syntax: manifest $ <name> = <constant>

< ; <name> = <constant> >$_{0}$ # <;>$_{1-}$

The constants are manifest expressions and may be composed of other manifest named constants, numerical constants (see section 4.3), and all operators except srshift, rv, and lv.

Semantics: This declaration declares each name to be a manifest constant with a value equal to the value of its associated constant expression. The meaning of a program would remain unchanged if all occurrences of manifest named constants were textually replaced by their corresponding values.

This facility has been provided to improve the readability of programs and to give the programmer greater flexibility in the choice of internal representations of data.

7.5 Simple Definitions

Syntax: N1, N2, ..., Nn = E1, E2, ..., En

Semantics: Data items with names N1,...,Nn are first declared, but not initialized, and then the following assignment command is executed:

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

A simple definition declares dynamic data items.

7.6 Vector Definitions

Syntax: N1, N2, ..., Nn = vec E1, E2, ..., En

where N1,...,Nn are names and E1,...,En are any manifest expressions (see section 7.4). (Other implementations may restrict n to 1.)

Semantics: The value of each constant expression Ei must be a manifest constant and it defines the maximum allowable subscript value of vector Ni. The minimum subscript value is always zero. The initial Rvalue of Ni is the Lvalue of the zeroth element of the vector; both Ni and the elements of the vector are dynamic data items.

The use of a vector is described in section 4.7.

7.7 Function Definitions

Syntax: N (<namelist>$_{1-}$) = E

where N is a name. (Other implementations may use [and] instead of parentheses.)

Semantics: This defines a function with name N; the data item defined is static and has its Rvalue initialized prior to execution of the program. The Lvalue of N is constant, and refers to a position in the global vector if and only if the function definition is in the scope of a global definition of the same name.

The names in the name list are called formal parameters and their scope is the body of the function E. The extent of a formal parameter lasts from the moment of its initialization in a call until the time when the evaluation of the body is complete.

All function and routines may be defined and used recursively.

Function applications are described in section 4.8.

7.8 Routine Definitions

Syntax: N (<namelist>$_{1-}$) be C

where N is a name. (Other implementations may use [and] instead of parentheses or may require C to be a block.)

Semantics: This defines a routine with name N. The semantics of a routine definition is exactly as for a function definition except that the body of a routine is a command and therefore its application yields no result. A routine should therefore only be called in the context of a command.

Routine commands are described in section 6.3.

7.9 Simultaneous Definitions

Syntax: D < and D >$_0$

Semantics: All the definitions are effectively executed simultaneously and all the defined data items have the same scope which, by the scope rules given in 7.1, includes the simultaneous definition itself; a set of mutually recursive functions and routines may thus be declared.

7.10 Definition Declarations

Syntax: let D < ; >$_{1-}$

Semantics: Definition D is declared and executed.

8.0 Program Sectioning and Linkage

BCPL/360 programs may consist of many control sections, each of which is separately compiled. The resulting object decks are combined by a linkage-editor or loader for execution. Normally one section is the main program body and contains the program entry point, while the remaining sections contain routine and function definitions.

8.1 Section Declaration

Syntax: csect N where N is a name.

This must be the first declaration in the control section. N is truncated to 7 characters.

Semantics: The current control section (i.e. the block body following the section declaration) is given the name $N. The global initialization entry point for this section is given the name N.

8.2 Program Declaration

Syntax: prog $ <name list> #

This declaration must occur exactly once in the total program. Usually it is placed with the global declarations in the main control section.

Semantics: Each name in the name list should occur in the section declaration of some control

section. A separate control section named #PROG is generated from this declaration. It contains the global initialization entry points of the named control sections (see section 10.3).

9.0 Compiler Features

This section describes the various compiler phases, options available, and error messages of the BCPL/360 Compiler.

9.1 Compiler Phases

Compilation of a control section consists of four phases. If more than one control section is being compiled by a single invocation of the compiler then the first two phases are done for all sections before the last two phases are done for the sections. Errors detected during a phase usually result in suppression of succeeding phases for the control section in error.

1) Syntax Analysis

   The input is preprocessed and syntax analyzed to form a syntax tree and a cross reference table.

2) Translation

   The syntax tree is translated into an intermediate object code for a stack machine.

3) Code Generation Pass 1

   The intermediate code is converted into equivalent S/360 machine code.

4) Code Generation Pass 2

   The S/360 machine code is output in the form of standard OS/360 object modules, one per control section.

9.2 Compiler Options

The following compiler output options are available. Each may be preceded by NO to suppress the option. Options are specified in the PARM field of the EXEC card. The default is

```
PARM='SOURCE,XREF,NOTREE,NOICODE,NOLIST,NODECK,LOAD'
```

1) SOURCE The input cards are numbered and printed. Statement numbers are also printed.

2) XREF Two name usage tables are printed, one for names used more than once and one for single usage names. The numbers following each name indicate the cards where the name was used.

3) TREE The syntax tree is printed.

4) ICODE The intermediate object code is output.

5) LIST An assembly listing of the S/360 machine code is printed. It is occasionally required for debugging. The statement numbers refer to those shown in the source listing.

6) DECK An object module is punched.

7) LOAD An object module is added to the loader input data set.

The only input option available is the number of control sections to be compiled in one invocation of the compiler. Control sections should be separated by a card containing the reserved word ENDSECTION. The last section should not be followed by ENDSECTION.

9.3 Error Messages

The following error messages are produced by the syntax analyzer. Each is printed near the card in error in the source listing.

```
SYNTAX TREE OVERFLOW
NAME MISSING OR IN ERROR
$ MISSING AFTER 'GLOBAL' OR 'MANIFEST'
= MISSING IN MANIFEST DEFINITION
# MISSING AT END OF DECLARATION
SECTION NAME MISSING
$ MISSING AT BEGINNING OF BLOCK
# MISSING AT END OF BLOCK
NON-NAME IN NAMELIST
NAME IS TOO LONG (>20 CHARACTERS)
'SECTION' MISSING AT BEGINNING OF SECTION
: MISSING IN GLOBAL DEFINITION
MISSING ) IN SUBEXPRESSION OR SUBSCRIPT
MISSING ) IN PARAMETER LIST
, MISSING IN CONDITIONAL EXPRESSION
ILLEGAL SYMBOL IN SUBEXPRESSION
INVALID FUNCTION OR ROUTINE NAME
INVALID USE OF ( IN 'LET' STATEMENT
INVALID SYMBOL IN 'LET', = ASSUMED
INVALID LABEL FIELD
INVALID COMMAND. POSSIBLY MISSING :=
'DO' (OR 'THEN') MISSING
MISSING 'OR' IN 'TEST' STATEMENT
= MISSING IN 'FOR' STATEMENT
'TO' MISSING IN 'FOR' STATEMENT
'DO' (OR 'THEN') MISSING IN 'FOR'
'INTO' MISSING IN 'SWITCHON' STATEMENT
: MISSING AFTER 'CASE' LABEL
: MISSING AFTER 'DEFAULT'
NAME EXPECTED TO LEFT OF = IN 'FOR' LOOP
'FIRST SYMBOL OF COMMAND OUT OF CONTEXT
ILLEGAL CHARACTER: char
STRING CONSTANT TOO LONG (>512 CHARS)
HEX CONSTANT IS TOO LONG (>8 DIGITS)
NAME TABLE OVERFLOW
ANALYSIS COMPLETED BEFORE END OF TEXT
```

The translator produces the following error messages. With each it prints the corresponding statement number and syntax subtree.

```
'DEFAULT' USED TWICE IN THE SAME SWITCH
COMPILER ERROR DETECTED IN DECLNAMES
COMPILER ERROR IN DECLVARS
```

```
'RESULTIS' OUTSIDE 'VALOF' BLOCK
A NAME, VECTOR APPLICATION OR 'RV' EXPRESSION EXPECTED
     ON THE LEFT SIDE OF A SIMPLE ASSIGNMENT
THE RIGHT HAND SIDE OF AN ASSIGNMENT OR SIMULTANEOUS
     DEFINITION HAS TOO MANY MEMBERS
EXPRESSION LIST TOO SHORT
STRUCTURE WRONG IN AN EXPRESSION
ERROR IN OPERAND OF 'LV'
NUMBER ON LEFT SIDE OF :=
THE FOLLOWING NAME HAS BEEN USED BUT NOT DECLARED
A FUNCTION OR ROUTINE HAS A DYNAMIC FREE VARIABLE
ERROR IN CONSTANT EXPRESSION
ILLEGAL OPERATOR IN CONSTANT EXPRESSION
NON-MANIFEST CONSTANT NAME IN CONSTANT EXPRESSION
NAME CLASH INVOLVING A LABEL
TOO MANY CASES IN A SWITCH
TWO DATA ITEMS WITH THE SAME NAME AND SAME SCOPE
TOO MANY NAMES DECLARED (>350)
TOO MANY INITIALIZED GLOBAL VARIABLE DEFINITIONS (>50)
EXPRESSION LIST OUT OF CONTEXT
UNKNOWN EXPRESSION OPERATOR
EXPRESSION MISSING
SUBEXPRESSION MISSING
TOO MANY CASES FOR A DIRECT SWITCH
TOO MANY CASES FOR A HASH SWITCH
A FEW TOO MANY CASES FOR A HASH SWITCH
NO CASES IN A SWITCH
```

The code generator produces only the following two warning messages.

```
nTH BASE REGISTER ALLOCATED AT LOC inst-addr
LOC data-addr MAY BE UNADDRESSABLE FROM LOC inst-addr
```

From an assembly listing (see section 9.2), one can determine whether these errors are fatal or ignorable.

The first message indicates that program base registers R10-R9 are being used as additional stack base registers (normally R14-R11, see section 10.4), and is fatal if the routine or function in which it occurs is more than 4096 bytes long. This error is usually caused by a routine containing a sequence of 5 or more nested let declarations, each equivalent to the following:

```
let v = vec 1022 || or larger
```

To program around this limitation, combine some of the declarations into one declaration.

The second message indicates that the data item at location data-addr is more than 4096 bytes from the start of the routine or function in which the message occurs, and is fatal if the routine is less than 4097 bytes long. In the fatal cases, the data item is usually the entry point of a local routine or function. To program around this limitation, declare the routine to be global.

9.4 Compiler JCL

The BCPL/360 Compiler is invoked as an OS/360 job step by the following job control language statements.

```
//BCPL     EXEC BCPL,PARM='options',TIME=1,REGION=150K
//SYSIN     DD  *
(BCPL/360 source decks)
/*
```

The cataloged procedure for BCPL is as follows.

```
//BCPL     EXEC PGM=BCPL
//STEPLIB  DD  DSNAME=SYS2.BCPL,DISP=SHR
//SYSPRINT DD  SYSOUT=A
//SYSPUNCH DD  SYSOUT=B
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  SYSOUT=B
//SYSGO    DD  DSNAME=&&LOADSET,DISP=(MOD,PASS),          X
//             UNIT=SYSDA,SPACE=(CYL,(1,1))
```

The compiler returns a return code of 0 if no errors are encountered. Otherwise the return code is 8.

The BCPL/360 Compiler is written in BCPL and hence uses the queued sequential input/output routines described in section 10.5. The default record format, block size, and logical record length values (see section 10.5.4) used by

the compiler for its I/O files are summarized in the following table.

DdName	RecFm	BlkSize	LRecL
SYSPRINT	FBA	1064	133
SYSPUNCH	F	80	80
SYSUT1	VB	3500	44
SYSUT2	F	80	80
SYSGO	FB	800	80
SYSIN	F	80	80

10.0 Run Time Support Features

This section describes the run time facilities available to the normal BCPL/360 user. However since BCPL interfaces easily with assembly code, almost all of the OS/360 facilities are available if desired.

10.1 General Considerations

A loadable BCPL/360 program consists of object modules produced by the BCPL/360 Compiler and/or OS Assembler. Most of these modules (control sections) are supplied by the user. However the control sections MAIN, OSPACK, and LIBRARY are maintained in a library since they form the run time support package described below. This package provides primarily for storage allocation, input/output, timing, and string manipulation.

Input/output in BCPL/360 uses the queued sequential access method (QSAM) of OS/360. Thus all OS record format, blocking, and buffering options are available. The routines described below provide for either character by character or record by record input/output. The special characters '*N', '*S', '*B', and '*T' are meaningful with character I/O (see section 4.2). Data sets intended for printing should use the first character of each record as a carriage control character instead of a data character.

Most BCPL/360 programs are reentrant. The run time support package is always reentrant. However the BCPL Compiler will produce non-reentrant code for any BCPL

section which modifies (by assignment statement or other means)

1) local label, routine, or function variables,

2) string or table constants.

The following standard declarations are used by the run time support package. These declarations should occur in every BCPL control section which uses the package.

```
GLOBAL || MAIN/OSPACK
$ PARM:0; RTNCODE:0; START:1
  GETMAIN:10; FREEMAIN:11; TIME:12; OPEN:13; CLOSE:14
  READCH:15; WRITECH:16; SETTABS:17; SETCCSW:18; STIMER:19
  TTIMER:20; GET:21; PUT:22; HASH:23; SVC:24; LOAD:25
  UNLOAD:26  #

MANIFEST || MAIN/OSPACK
$ EODCH=55; IN=0; OUT=1; F=128; FB=144; FA=132; FBA=148
  FS=136; FBS=152; FSA=140; FBSA=156; V=64; VB=80; VA=68
  VBA=84; VS=72; VBS=88; VSA=76; VBSA=92; U=192; UA=196
  TU=0; BIN=1; DEC=2; TOD=3; CANCEL=1; NOCANCEL=0
  TASK=0; WAIT=1; REAL=3; EU=0; EC=32; VU=192; VC=224  #

GLOBAL || LIBRARY
$ INPUT:30; OUTPUT:31; FORMNUMBER:40; FORMDIGIT:41
  PACKSTRING:42; UNPACKSTRING:43; PACKSTRING4:44
  UNPACKSTRING4:45; WRITES:46; WRITEN:47; WRITEX:48
  GETPARM:49; EQSTRING:50; GTSTRING:51; CAT:52; ACTIVATE:53
  RESUME:54; CURRENTP:55  #

MANIFEST || S/360 MACHINE SPECIFICATIONS
$ BYTEBITS=8; BYTEMAX=255; BYTESPERWORD=3
  BYTE1SHIFT=16; BYTE2SHIFT=8; BYTE3SHIFT=0  #
```

(Note: Global words 0 through 59 are reserved for use by the run time support package.)

10.2 Loader

The BCPL/360 Loader reads the users object modules and those in the library, forms the program in core, and transfers control to the program entry point. It is invoked by the following JCL.

```
//BCPLGO  EXEC BCPLGO,PARM='parm field',TIME=t,REGION=r
//ddname   DD  VOLUME=v,UNIT=u,DSNAME=n,DISP=d,                X
//             DCB=dcb,SPACE=s,LABEL=l
//SYSLIN   DD  *
(additional object modules)
//SYSIN    DD  *
(program's input data)
/*
```

The cataloged procedure for BCPLGO is as follows.

```
//BCPLGO  EXEC PGM=LOADER,COND=(4,LT)
//STEPLIB  DD  DSNAME=SYS2.BCPL,DISP=SHR
//SYSLOUT  DD  SYSOUT=A
//LOADSET  DD  DSNAME=&&LOADSET,DISP=(MOD,DELETE),             X
//             UNIT=SYSDA,SPACE=(TRK,(1,1))
//LIBRARY  DD  DSNAME=SYS2.BCPLLIB,DISP=SHR
//SYSPRINT DD  SYSOUT=A
//SYSPUNCH DD  SYSOUT=B
//SYSOUT   DD  SYSOUT=A
```

The loader requires 4K of core for its own code. It requests 30K to 450K for the program code area plus 5K for I/O buffers. The buffers and unused program space are released prior to program execution.

The loader reads object modules from data sets in the order LOADSET, SYSLIN, LIBRARY. Only the first occurance of a control section is loaded. The dataset &LOADSET is produced by the LOAD option of the BCPL Compiler or OS Assembler. The LOADSET DD card may be used even if &LOADSET has not been created. The SYSLIN DD card is either omitted or used to enter additional object modules. The library contains the MAIN, OSPACK, and LIBRARY object modules. The data sets are read using QSAM input.

The loader writes a program map on the data set SYSLOUT. Each line contains a control section name and starting address or an entry point name and address. When loading is completed, the program length and program entry

point address are printed. All addresses are the actual core addresses.

The loader accepts two control cards. Each is punched beginning in column 1. The card

STOP

causes the loader to stop reading the current input data set and procede to the next input set. The card

ENTRY name

makes name the new program entry point.

The program entry point is determined by the first ENTRY card encountered. If there are none then the first object module with an entry point address on its END card (normally the control section MAIN) determines the entry point. Such modules may be produced by the Assembler but never by the BCPL/360 Compiler. If there still is no entry point then the start of the first control section is used.

The loader has no options and hence does not use the parm field. Instead it passes the parm field to the loaded program according to standard OS conventions. The data sets defined by the SYSPRINT, SYSPUNCH, SYSOUT, ddname, and SYSIN DD cards are used only by the loaded program. In other words, the loader is completely transparent to the loaded program (except that 4K of core is unavailable). Thus loaded program return codes are passed directly to the job scheduler.

Since BCPL object modules conform to OS conventions the OS Linkage-Editor may be used to form load modules with the

following JCL.

```
//LKED     EXEC PGM=IEWL,PARM='MAP,LIST,NCAL',COND=(4,LT)
//SYSPRINT DD   SYSOUT=A
//SYSLIN   DD   DSNAME=&LOADSET,DISP=(OLD,DELETE)
//         DD   DDNAME=SYSIN
//         DD   DSNAME=SYS2.BCPLLIB,DISP=SHR
//SYSLMOD  DD   DSNAME=&GOSET(GO),DISP=(MOD,PASS),          X
//              UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//SYSUT1   DD   UNIT=SYSDA,SEP=SYSLMOD,SPACE=(CYL,(1,1))
//SYSIN    DD   *,DCB=BLKSIZE=800
(additional object modules)
/*
```

Note that the BLKSIZE of &LOADSET and SYS2.BCPLLIB is normally 800. The load module GO in &GOSET may be executed by:

```
//GO       EXEC PGM=*.LKED.SYSLMOD,PARM='parm field',       X
//              COND=(4,LT)
//SYSPRINT DD   SYSOUT=A
//SYSPUNCH DD   SYSOUT=B
//SYSOUT   DD   SYSOUT=A
//ddname   DD   appropriate parameters
//SYSIN    DD   *
(program's input data)
/*
```

10.3 MAIN Control Section

MAIN is the primary entry and exit point of a BCPL/360 program. It allocates space for global and stack variables, reserves space for I/O buffers etc., initializes global labels and entry points, transfers control to a BCPL section, receives control from a finish statement, closes I/O files, releases all allocated space, and passes a return code to its caller. MAIN and hence the total BCPL program satisfies the standard OS/360 subroutine linkage conventions when interfacing with OS. Within BCPL code, however, different conventions are used (see section 10.4).

MAIN is called (normally by the job scheduler) with one parameter, the parm field of the EXEC card. MAIN searches this parm field for one subparameter, namely

S=(Minstack,Maxstack,Global,Buffer)

The usual JCL conventions for omitting positional parameters apply. These four parameters determine the amount of global, stack, and buffer space allocated as follows. Minstack and Maxstack are given in units of thousands of words. As much space as possible within this range will be allocated for the run time stack. Global specifies the amount of space for global variables in hundreds of words. Buffer specifies the amount of space to be reserved for input/output buffers, GetMain storage requests, etc. in thousands of bytes. The default values are S=(10,50,2,10).

MAIN uses the prog declaration information to initialize the global labels, routine and function entry points. Normally all routines declared in one control section which are to be called by other control sections must be declared as global routines (in both sections). This applies particularly to the routines in the support package. Hence the names OSPACK and LIBRARY should always be included in the name list of the prog declaration.

MAIN sets PARM (global word 0) to point to a vector containing the following information.

```
PARM.(0) = stack size in words
PARM.(1) = global size in words
PARM.(2) = buffer reserve size in words
PARM.(3) = number of parm field characters (0 to 100)
PARM.(4) to PARM.(28) = the parm field of the EXEC card
                        packed four characters per word
                        beginning with PARM.(4)
```

After the global initialization, MAIN transfers control to the label START (global word 1). START should be declared in the block body of the main BCPL control section. On receiving control from a finish statement, MAIN closes any files still open, releases all storage which it has allocated, in particular the global and stack areas, and returns to the caller with a return code. If RTNCODE (global word 0) is 0 to 4095 then it becomes the return code. Otherwise the code is 0.

10.4 BCPL Error Dump

The following program errors transfer control to a dump routine in the MAIN control section.

Code	Meaning
1	operation (illegal machine instruction)
2	privileged operation
3	execute
4	protection (memory protection violation)
5	addressing (address larger than memory size)
6	specification (usually incorrect boundary alignment)
7	data

An attempt is made to print a specially formatted dump. Then the program is terminated via ABEND. The following DD statement must be included in the job step JCL to get a BCPL dump (see section 10.2).

```
//SYSOUT   DD  SYSOUT=A
```

(Note: RecFm, BlkSize, and LRecL are forced to FBA, 1064, and 133 respectively.)

Similarly a //SYSUDUMP DD SYSOUT=A card will give a standard OS ABEND dump. The ABEND user completion code will be the interruption code given above if a BCPL dump is printed. Otherwise it is 100 plus the code. On entry to ABEND all general registers are as they were at the interrupt except R1 which contains the user completion code and R6 which points to the instruction in error.

A BCPL dump contains the interruption type and location, the general register contents at the time of the interrupt, the MAIN work area, sub-global area, global area, stack area, the currently open I/O file control blocks and I/O buffer areas, and finally the program code sections. In the dump, columns labelled BCPL DECIMAL are 30 bit words, and columns labelled BCPL STRING are 3 characters per 30 bit word. Decimal addresses are word addresses, whereas hex addresses are byte addresses.

MAIN work area is used by the MAIN and OSPACK sections. Sub-global is a 6 word area which immediately precedes global word 0. The global area is immediately followed by the stack area. The first part of a file control block is a QSAM data control block (DCB). The character buffers are the records currently being read or written by ReadCh and WriteCh (see section 10.5). The buffer pools contain the next records to be read or the last records written.

BCPL/360 programs use general registers and the stack

area in the following manner. R1, R3, and R5 are used for expression evaluation. Occasionally they are used in even-odd pairs, i.e. R0-R1, R2-R3, and R4-R5. R7 points to the sub-global area. R8 is the program code base register. R9 and R10 are program base registers when the routine is over 4095 bytes. R14 is the current activation record (stack base) register. For large activation records, extra base registers from R13 to R9 are used. R15 is the return address and branch address register.

On entry to a BCPL routine or function an activation record is allocated at the current top of the stack. The first 8 words are a save area containing R8-R15 of the calling routine. The next words are the routine arguments. Above these are the stack variables and vectors (allocated by let declarations, see section 7). When a dump occurs, R7 locates the global area, R8 the currently executing routine, and R14 the current activation record. From this one can trace back through the sequence of routine calls by following the saved R14 values.

10.5 OSPACK Control Section

OSPACK is a collection of routines and functions which interface with OS/360. It includes routines for main storage allocation, input/output, and timing. In the descriptions which follow, all numbers and strings are assumed to be in the BCPL 30 bit word format unless explicitly stated otherwise.

10.5.1 GetMain

Syntax: CondCode := GETMAIN(Type,Hiarchy,SubPool,Loc,
Length,MaxLen)

This function is used to obtain core storage outside the stack and global areas, i.e. from the space reserved for I/O buffers etc. Type determines the type of request being made. An unconditional request ABENDS if there is insufficient storage, whereas a conditional request always returns control. A variable request specifies an upper and lower bound on the amount of core. As large a block as possible within this range will be allocated. A fixed request specifies an exact amount. Requests should be for an even number of words. Type should be one of the following:

EU fixed, unconditional
EC fixed, conditional
VU variable, unconditional
VC variable, conditional

Hiarchy determines which storage hierarchy is to be allocated from. Fast core is 0 and slow core is 1. SubPool determines the storage subpool (0 to 127). Subpool 0 is the primary storage pool. Hiarchy and SubPool should normally be 0. Length is the minimum or fixed length requested in words. MaxLen is the maximum length in words for variable requests. Loc should point to a pair of words. The first is set to the address of the allocated storage and the second to its length. The function returns true if the storage is allocated and false otherwise.

10.5.2 FreeMain

Syntax: FREEMAIN(SubPool,Address,Length)

This routine releases storage obtained via GetMain. SubPool is the storage subpool (normally 0). Address points to the area to be released. Length is the length of the area in words. A length of 0 means the whole subpool is to be released (except for subpool 0).

10.5.3 Time

Syntax: TIME(Unit,Loc)

This routine obtains the real time of day and date in a printable format. Unit is one of DEC, BIN, or TU. DEC returns the time of day in the string form 'hh:mm:ss.th'. BIN returns it as an integer giving the number of hundredths of seconds since midnight. TU returns a 32 bit unsigned integer giving the number of timer units since midnight. A timer unit is about 26.04 microseconds. However the timer on most S/360 machines is updated only 60 times a second.

Loc points to a 9 word result area. Loc.(0) becomes the integer year, i.e. 70 for 1970. Loc.(1) is the integer day (1 to 366). Loc.(2) to Loc.(4) contain the date in the string form 'mm/dd/yy'. Loc.(5) to Loc.(8) contain the DEC time for DEC requests. Loc.(5) contains the BIN or TU time for these requests.

10.5.4 Open

Syntax: OPEN(N,DdName1,IO1,RecFm1,BlkSize1,LRecL1,
FcbLoc1,DdName2,...)

This routine opens datasets for reading or writing. At most 10 files may be open simultaneously. All I/O uses the queued sequential access method of OS/360. N is the number of files to be opened. The rest of the argument list consists of six arguments for each file. DdName is a string giving the JCL statement ddname which defines the data set, e.g. 'SYSPRINT'. IO is IN for input files and OUT for output files. RecFm is the file record format, BlkSize its block size, and LRecL its logical record length. FbcLoc is the Lvalue of a word whose Rvalue becomes either the Lvalue of a file control block or 0 if the file fails to open.

The record format may be fixed length records (F), variable length (V), or undefined (U). The records may be blocked (B) or unblocked for F and V formats. Fixed standard and variable spanned formats may also be used (S). The first character of each record may be used as a carriage control character for printed files (A). The carriage control characters are summarized in the table below.

Char	Meaning
blank	skip one line before printing line
0	skip two lines before printing
-	skip three lines before printing
+	skip zero lines before printing
1	skip to channel 1 (top of page) before printing

The record format should be one of the following: F, FB, FA, FBA, FS, FBS, FSA, FBSA, V, VB, VA, VBA, VS, VBS, VSA, VBSA, U, UA.

For fixed format records the logical record length gives the length of the records in bytes. For variable

format records LRecL is the maximum length in bytes of a logical record. This maximum length must be 4 bytes greater than the actual maximum data length because a 4 byte control word is appended to the front of each data record for V format files. LRecL should be 0 for undefined format files since each physical block is interpreted as a logical record.

The block size is the length in bytes of a physical block. For fixed formats it must be a multiple of LRecL. For V and U formats it is the maximum length of a block. For V formats this maximum length must include a 4 byte block control word at the start of each block.

The actual record format value is determined in the following way. If RecFm < 0 then the format is -RecFm. Otherwise RecFm is a default value which is used only if no record format has been specified in the JCL DD statement or in the data set label. If RecFm > 0 then it is the default value. If RecFm = 0 then a standard default value depending on the ddname is used as summarized in the table below. The LRecL and BlkSize parameters are treated in a similar manner.

		Standard Defaults		
DdName	Use	RecFm	BlkSize	LRecL
SYSIN	input	F	80	80
SYSPRINT	print	FBA	1064	133
SYSPUNCH	punch	F	80	80
all else	any	FB	800	80

The following statement is normally used to open the standard input and print files.

```
OPEN(2,'SYSIN',IN,0,0,0,LV INPUT,
     'SYSPRINT',OUT,0,0,0,LV OUTPUT)
```

10.5.5 Close

Syntax: CLOSE(N,Fcb1,Fcb2,...)

This routine closes I/O files. N is the number of files to be closed. Fcb is a file control block. The files must be open. For output files if the current line buffer is non-empty then it is written out prior to closing. All storage space obtained by Open for buffers etc. is released.

10.5.6 ReadCh

Syntax: Char := READCH(Fcb)

This function reads and returns the next character from the input file designated by the file control block Fcb. At the end of each logical record, ReadCh returns the character '*N'. When the end of the data set is reached, ReadCh thereafter returns the character EODCH.

10.5.7 WriteCh

Syntax: WRITECH(Fcb,Char)

This routine writes the character Char onto the output file Fcb. Only the low order 8 bits of Char are used. If the current record is full then it is written out and Char begins a new record. The characters '*N', '*T', and '*B' are not written but cause certain control actions instead. The new line character '*N' terminates the current record. For fixed format files, blanks are appended to the end of

short records. The tab character '*T' inserts blanks up to the next tab stop or the end of the record. The backspace character '*B' backs up the character pointer one position so that the last character written is replaced by the next character to be written. '*B' is ignored if the current record is empty.

10.5.8 SetTabs

Syntax: SETTABS(Fcb,N,Tab1,Tab2,...)

This routine sets tab stops for an output file. Fcb is a file control block. N is the number of tabs to set. Tab1 etc. are the tabs in increasing order. A file may have at most 10 tab stops simultaneously. SetTabs clears all previous tabs before setting the N new stops. The first data character of a record is tab position 1. For A format files (see section 10.5.4) this first character is a carriage control character if the file is printed.

10.5.9 SetCcSw

Syntax: SETCCSW(Fcb,CcSwitch)

This routine sets a control character switch which determines the interpretation of I/O control characters. Fcb is a file control block and CcSwitch is true or false. The switch is true when a file is opened. For input files a true switch means that ReadCh returns '*N' for end of record and EODCH for end of file (see section 10.5.6), while a false switch means it returns 256+'*N' and 256+EODCH

respectively. For output files a true switch means that WriteCh takes control action for the characters '*N', '*T', and '*B' (see section 10.5.7), while a false switch means these characters are written like any other data characters.

10.5.10 STimer

Syntax: STIMER(Mode,Unit,Time)

This routine sets an interval timer. If a time interval was previously set, it is cancelled. Mode is one of TASK, WAIT, or REAL. A task time interval is decremented only when the CPU is executing the user's task. A wait interval places the user's task in wait status until the (real time) interval expires. A real time interval is continuously decremented regardless of the state of the CPU, user's task, etc. When an interval expires it is automatically cancelled.

Unit is one of TU, BIN, DEC, or TOD, while Time is the time interval or time of day to be set. For TU time, Time is a 32 bit unsigned integer giving the time interval in timer units (about 26.04 microseconds). For BIN time, Time is a BCPL integer giving the interval in hundredths of seconds. For DEC time, Time is a string of form 'hh:mm:ss.th' giving the interval in hundredths of seconds. For TOD time (useable only with WAIT or REAL modes), Time is a string of form 'hh:mm:ss.th' giving a real time of day at which the interval expires. (Note: BinTime = TuTime / 384 . The timer is normally updated only 60 times a

second.)

10.5.11 TTimer

Syntax: TimeLeft := TTIMER(Unit,Cancel)

This function returns the time remaining in a time interval set by STimer. Unit is either TU or BIN (see section 10.5.10). Cancel is CANCEL if the time interval is to be cancelled and NOCANCEL if the time interval is to remain in force.

10.5.12 Get

Syntax: Length := GET(Fcb,Record)

This function reads the next logical record from the input file designated by file control block Fcb. Record points to an area in which the record is to be placed. The record is packed 4 characters per word and begins in Record.(1). Record.(0) is used as a control word for variable format files and may be ignored. Get returns the length in bytes of the record as its value. When the end of the file is reached, Get returns 0 thereafter. (Note: The function ReadCh (see section 10.5.6) uses Get when it reads a new record.)

10.5.13 Put

Syntax: PUT(Fcb,Record,Length)

This routine writes a logical record on the output file designated by file control block Fcb. Record points to the

record to be written. The record must be packed 4 characters per word and begin in Record.(1). Record.(0) is used as a control word for variable format files. Length is the length of the record in bytes. For fixed format files if Length is less than the file logical record length then the appropriate number of blanks are appended to the end of the record in the area pointed to by Record. (Note: The routine WriteCh (see section 10.5.7) uses Put when it writes a record.)

10.5.14 Hash

Syntax: HASH(String,Codes,N,Base1,Base2,...)

This routine forms hash codes from a string. String is the string to be hashed. It must be in the unpacked format, i.e. String.(0) = length, String.(1) = first character, String.(2) = second character, etc. Codes points to a vector of length N in which the hash codes are to be placed, one per word. N is the number of codes to construct. For I = 1,...,N the Ith code is reduced modulo BaseI, i.e. 0 <= CodeI <= BaseI-1.

10.5.15 SVC

Syntax: R15 := SVC(N,Regs)

This function calls the system supervisor by executing a SVC N instruction where N is the number of the supervisor call. Regs points to a 4 word vector from which general registers R14, R15, R0, and R1 are loaded just prior to the

SVC. (Regs.(0)=R14, ..., Regs.(3)=R1.) The contents of R14-R1 just after the SVC is completed are returned in the Regs vector. The R15 contents are also returned as the function value. N is a BCPL integer (0-255), whereas all register quantities are 32-bit items.

10.5.16 Load

Syntax: LOAD(Name)

This routine loads a BCPL load module into core. Name is a BCPL string giving the member name of the load module. The load module data set must be a STEPLIB, JOBLIB, or LINKLIB data set. At least one control section in the load module must be a BCPL control section and must contain a prog declaration whose name list refers to exactly those BCPL control sections which are in the load module. When the load module is created by the OS/360 Linkage Editor, an ENTRY #PROG card must be included with the control section object decks. The Load routine uses this entry point to locate and enter the global initialization sequence of each BCPL control section in the load module. (Note: The base load module loaded by the job scheduler should contain the MAIN, OSPACK, and LIBRARY control sections, and also a BCPL main control section which defines the START label and whose prog declaration refers to exactly the BCPL sections in the base module. The entry point of the base module is MAIN.)

10.5.17 Unload

Syntax: UNLOAD(Name)

This routine deletes a load module loaded by Load. Name is a BCPL string giving the member name of the load module.

10.6 LIBRARY Control Section

LIBRARY is a collection of routines and functions for input/output and string manipulation. Instead of describing each routine, the actual source code for the LIBRARY section is given below. Note that the output routines use the global variable OUTPUT to determine which file to write on.

```
SECTION LIBRARY

|| (INSERT STANDARD DECLARATIONS OF SECTION 10.1 HERE.)

|| BCPL/360 LIBRARY

LET FORMNUMBER(X) = X >= '0' -> X - '0', X - 'A' + 10
|| RETURNS AS A BCPL INTEGER THE VALUE OF THE EBCDIC DIGIT X

AND FORMDIGIT(X) = X < 10 -> X + '0', X + 'A' - 10
|| RETURNS THE EBCDIC EQUIVALENT OF THE BCPL INTEGER X (<16)

|| NOTE: IN THE FOLLOWING DESCRIPTIONS, THE TERM 'STRING'
|| REFERS TO ANY VECTOR FORMATTED THE SAME AS A BCPL STRING
|| CONSTANT.


AND PACKSTRING(V,S) BE

|| FORMS THE CHARACTERS IN THE VECTOR V INTO A STRING AND
|| PLACES THE RESULT INTO S. V SHOULD BE OF THE FORM:
||          V.(0) = NUMBER OF CHARACTERS IN V
||          V.(1),...,V.(V.(0)) = CHARACTERS

   $1 LET N = V.(0)
      LET I,J = 0,0
      V.(N+1), V.(N+2) := 0,0
      UNTIL J > N DO
          $ S.(I) := V.(J) LS BYTE1SHIFT |
                     V.(J+1) LS BYTE2SHIFT | V.(J+2)
            I,J := I+1, J+3   #1
```

```
AND UNPACKSTRING(S,V) BE

|| EXPANDS THE STRING S INTO A VECTOR OF SINGLE CHARACTERS
|| AND PLACES THE RESULT IN V. V IS IN THE SAME FORMAT AS
|| THE INPUT TO PACKSTRING.

   $1 LET N = S.(0) RS BYTE1SHIFT
      LET I,J = 0,0
      UNTIL J > N DO
          $ LET W = S.(I)
            V.(J) := W RS BYTE1SHIFT
            V.(J+1) := W RS BYTE2SHIFT & BYTEMAX
            V.(J+2) := W & BYTEMAX
            I,J := I+1, J+3   #1

AND PACKSTRING4(V,S) = VALOF

|| PACKS THE CHARACTERS IN V INTO S, FOUR CHARACTERS PER
|| WORD. V SHOULD BE IN THE SAME FORMAT AS INPUT TO
|| PACKSTRING. THE NUMBER OF CHARACTERS PACKED IS RETURNED.
|| EXTRA BYTES IN THE LAST WORD ARE FILLED WITH BLANKS.

   $1 LET N = V.(0)
      LET I,J = 0,1
      V.(N+1), V.(N+2), V.(N+3) := '*S', '*S', '*S'
      UNTIL J > N DO
          $ S.(I) := V.(J) LS 22 | V.(J+1) LS 14 |
                     V.(J+2) LS 6 | V.(J+3) SRS 2
            I,J := I+1, J+4  #
      RESULTIS N   #1


AND UNPACKSTRING4(N,S,V) BE

|| UNPACKS THE FIRST N CHARACTERS FROM S AND PLACES THEM IN
|| V, ONE CHARACTER PER WORD. V IS IN THE SAME FORMAT AS
|| INPUT TO PACKSTRING.

   $1 LET I,J = 0,1
      V.(0) := N
      UNTIL J > N DO
          $ LET W = S.(I)
            V.(J) := W RS 22
            V.(J+1) := W RS 14 & BYTEMAX
            V.(J+2) := W RS 6 & BYTEMAX
            V.(J+3) := W LS 2 & BYTEMAX
            I,J := I+1, J+4   #1


AND WRITES(S) BE

|| WRITES A STRING OR SINGLE CHARACTER ONTO THE FILE
|| 'OUTPUT'.
```

```
   $1 TEST 0 <= S <= BYTEMAX THEN WRITECH(OUTPUT,S)
       OR $ LET V = VEC 260
            UNPACKSTRING(S,V)
            FOR I = 1 TO V.(0) DO WRITECH(OUTPUT,V.(I))    #1


AND WRITEN(N) BE

|| WRITES THE DECIMAL VALUE OF N ONTO THE FILE 'OUTPUT'.

   $1 LET V = VEC 20
      LET NEG = N < 0
      LET J = VALOF
          $ IF NEG DO TEST N = -536870912
                          THEN $ WRITES('-536870912')
                                 RETURN  #
                           OR N := -N
            FOR I = 0 TO 20 DO
                $ V.(I) := N REM 10
                  N := N / 10
                  IF N = 0 RESULTIS I  #
            RESULTIS 20  #
      IF NEG DO WRITECH(OUTPUT,'-')
      FOR I = 0 TO J DO
            WRITECH(OUTPUT,FORMDIGIT(V.(J-I)))    #1


AND WRITEX(N) BE

|| WRITES THE HEXADECIMAL VALUE OF N ONTO THE FILE 'OUTPUT'.

    $ FOR I = 28 TO 0 BY -4 DO
            WRITECH(OUTPUT, FORMDIGIT(N RS I & 15))  #


AND GETPARM(PL,P) BE

|| REMOVES THE FIRST PARAMETER FROM THE PARAMETER LIST PL
|| AND PUTS IT INTO P. PL SHOULD BE THE UNPACKED FORM OF A
|| STANDARD OS/360 JCL PARAMETER LIST.

   $1 LET I,J,PC = 1,1,0
      UNTIL I > PL.(0) DO SWITCHON PL.(I) INTO
         $2 CASE '(':   PC := PC + 1
                        GOTO L

            CASE ')':   PC := PC - 1
                        IF PC >= 0 GOTO L
                        PL.(0), P.(0) := 0,0
                        RETURN

            CASE ',':   UNLESS PC = 0 GOTO L
                        I := I + 1
                        BREAK
```

```
            DEFAULT:
            L:          P.(J) := PL.(I)
                        I,J := I+1, J+1  #2
        P.(0), J := J-1, 1
        FOR K = I TO PL.(0) DO PL.(J), J := PL.(K), J+1
        PL.(0) := J - 1   #1


AND EQSTRING(S1,S2) = VALOF

|| RETURNS 'TRUE' IF THE STRINGS S1 AND S2 ARE IDENTICAL,
|| OTHERWISE THE RESULT IS 'FALSE'.

     $ LET N = (S1.(0) RS BYTE1SHIFT) / BYTESPERWORD
       FOR I = 0 TO N DO UNLESS S1.(I)=S2.(I) RESULTIS FALSE
       RESULTIS TRUE   #


AND GTSTRING(S1,S2) = VALOF

|| RETURNS 'TRUE' IF STRING S1 IS ALPHABETICALLY GREATER
|| THAN STRING S2, OTHERWISE RETURNS 'FALSE'.

    $1 LET A1, A2 = S1.(0) & "FFFF", S2.(0) & "FFFF"
       UNLESS A1 = A2 RESULTIS A1 > A2
     $ LET N1,N2 = S1.(0) RS BYTE1SHIFT, S2.(0) RS BYTE1SHIFT
       FOR I = 1 TO (N1 <= N2 -> N1, N2) / BYTESPERWORD DO
           $ LET A1, A2 = S1.(I), S2.(I)
             UNLESS A1 = A2 RESULTIS A1 > A2  #
       RESULTIS N1 > N2   #1


AND CAT(SL,SR,S) = VALOF

|| CONCATENATES THE STRINGS SL AND SR AND PLACES THE
|| RESULTING STRING IN S (WHICH IS ALSO THE FUNCTION VALUE).

    $1 LET NL,NR = SL.(0) RS BYTE1SHIFT, SR.(0) RS BYTE1SHIFT
       LET V = VEC 515
       UNPACKSTRING(SL,V)
     $ LET T = V.(NL)
       UNPACKSTRING(SR,V+NL)
       V.(NL), V.(0) := T, NL+NR
       PACKSTRING(V,S)
       RESULTIS S   #1


AND ACTIVATE(PCB,STACK,ENTRY,
             P1,P2,P3,P4,P5,P6,P7,P8,P9,P10) BE

|| ACTIVATES A PROCESS (COROUTINE).
|| PCB         IS THE LVALUE OF A WORD TO BE USED TO IDENTIFY
||             THE PROCESS AND ITS CURRENT STATE, I.E. A
||             PROCESS CONTROL BLOCK.
```

```
|| STACK       IS THE LVALUE OF THE FIRST WORD OF A VECTOR TO
||             BE USED BY THE PROCESS FOR STACK SPACE.
|| ENTRY       IS THE ENTRY POINT OF THE BCPL ROUTINE WHICH
||             IS TO BE EXECUTED WHEN THE PROCESS IS FIRST
||             RESUMED.
|| P1,...,P10  ARE THE ARGUMENTS TO BE PASSED TO THE BCPL
||             ROUTINE WHEN IT IS FIRST RESUMED.
|| ACTIVATE MAY BE CALLED BY ANY PROCESS ANY TIME AS LONG AS
|| PCB AND STACK DO NOT REFER TO DATA AREAS ALREADY IN USE.

   $1 MANIFEST $ STM = "908F100" LS 2 # || STM 8,15,0(1)
      LET ARGS = LV P1 - 8
      UNLESS ENTRY.(0) = STM DO ENTRY := (ENTRY & TRUE).(2)
      STACK.(0), STACK.(1), STACK.(2) :=
                    ENTRY, ENTRY+1024, ENTRY+2048
      STACK.(6), STACK.(7) := STACK, ENTRY + (10 SRS 2)
      FOR I = 8 TO 17 DO STACK.(I) := ARGS.(I)
      PCB.(0) := STACK    #1


AND RESUME(PCB) BE

|| SAVES THE STATE OF THE CURRENTLY ACTIVE PROCESS
|| (IDENTIFIED BY GLOBAL VARIABLE CURRENTP) AND RESUMES
|| (FROM ITS PREVIOUSLY SAVED STATE) THE PROCESS IDENTIFIED
|| BY PCB.  PCB IS THE LVALUE OF THE WORD ASSOCIATED WITH
|| THE PROCESS BY ACTIVATE.  (NOTE: BEFORE THE FIRST CALL OF
|| RESUME, CURRENTP MUST BE SET TO THE LVALUE OF SOME WORD.
|| THIS PCB MAY BE USED TO RESUME THE MAIN PROGRAM.)

   $1 LET SWITCH_STACKS(P) BE (LV P).(-2) := CURRENTP.(0)
      CURRENTP.(0) := LV PCB - 8
      CURRENTP := PCB
      SWITCH_STACKS()    #1
```

REFERENCES

[1] Richards, Martin, The BCPL Reference Manual, Project MAC Memorandum M-352-1, Massachusetts Institute of Technology, February 16, 1968.

[2] Strachey, C. (Editor), CPL Working Papers, a technical report, London Institute of Computer Science and the University Mathematical Laboratory, Cambridge, 1966.

[3] IBM System/360 Principles of Operation, Form A22-6821, IBM Systems Reference Library.

[4] IBM System/360 Operating System: Supervisor and Data Management Services, Form C28-6646, IBM Systems Reference Library.

[5] IBM System/360 Operating System: Job Control Language, Form C28-6539, IBM Systems Reference Library.