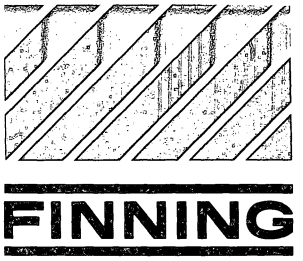
Finning BCPL System

Reference Manual

FINNING



FINNING COMPUTER SERVICES LTD.

HEAD OFFICE: 555 GREAT NORTHERN WAY, VANCOUVER, B.C. V5T 1E2 • PHONE (604) 872-7474 • TELEX 04-508717 • CABLE ADDRESS "FINTRAC

Finning BCPL System

Reference Manual

Revision date of this document:
July 27, 1977

Table of Contents

1 Introduction

BCPL is a programming language designed primarily for non-numerical applications such as compiler-writing and general systems programming. It has been used successfully to implement compilers, interpreters, text editors, game playing programs, and operating systems. The BCPL compiler is written in BCPL and runs on a wide variety of machines including the IBM 360/370 series. This document describes the implementation available on the Data General Nova series processors at Finning.

Some of the distinguishing features of BCPL are:

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

The basic data object is a word (16 bits on the Nova) 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 may be called recursively.

This manual is not intended as a primer; the constructs of the language are presented with scant motivation and few examples. To use BCPL most effectively on the Nova one should have a good understanding of how the machine works and be familiar with its operating system. To the experienced and disciplined programmer it is a powerful and useful language but there are few provisions for the protection of naive users.

2 Language Definition

2.1 Program

At the outermost level, a BCPL program consists of a sequence of declarations. To understand the meaning of a program, it is necessary to understand the meaning of the more basic constructs of the language from which it is made. We, therefore, choose to describe the language from the inside out starting with one of the most basic constructs: the 'element'.

2.2 Elements

```
<element>  ::=  <identifier> \ <number> \
                TRUE \ FALSE \ ? \
                <string constant> \ <character constant>
```

An <identifier> consists of a sequence of letters, digits, periods, and underlines, the first character of which must be a letter.

A <number> is either an integer consisting of a sequence of decimal digits, an octal constant consisting of the symbol '#' followed by octal digits, or a hexadecimal constant consisting of the character pair #X followed by hexadecimal digits. The reserved words TRUE and FALSE denote -1 and 0 respectively (on a 2's complement machine) and are used to represent the two truth values. The symbol '?' may be used anywhere in an expression when no specific value is required, as in:

```
LET OP, A = ?, ?
```

A <string constant> consists of up to 255 characters enclosed in string quotes ("). The internal character set is stripped ASCII (on the Nova). The character " may be represented only by the pair *" and the character * can only be represented by the pair **. Other characters may be represented as follows:

*N	is newline
*C	is carriage return
*T	is horizontal tab
*S	is space
*B	is backspace
*P	is newpage

(These are considered standard; additional escape sequences are described in section 5.31)

Within a string, the sequence

* <newline> [<space> \ <tab>] *

is skipped. Thus, the string

```
"THIS STRING *
*CONTAINS NEWLINES*
* AND SPACES"
```

is precisely equivalent to

```
"THIS STRING CONTAINS NEWLINES AND SPACES"
```

The machine representation of a string is the address of the region of store where the length and characters of the string are packed. The packing and unpacking of strings may be done using the machine dependent library routines PACKSTRING and UNPACKSTRING, and individual characters in a string can be accessed and updated using the library routines GETBYTE and PUTBYTE (see section 2.82).

A <character constant> consists of a single character enclosed in character quotes ('). The character ' can be represented in a character constant only by the pair *'. Other escape conventions are the same as for a string constant. A character constant is right justified in a word. Thus 'A' = 65 (on the Nova).

2.3 Expressions

Because an identifier has no type information associated with it, the type of an element (and hence an expression) is assumed to match the type required by its context.

All expressions are listed below. E1, E2 and E3 represent arbitrary expressions except as noted in the descriptions which follow the list, and K0, K1 and K2 represent constant expressions (whose values can be determined at compile time; see section 2.38).

Primary	element (E1)	
function call	E1() E1(E2,E3,...)	
addressing	E1!E2 @E1 !E1	subscripting address generation indirection
arithmetic	E1 * E2 E1 / E2 E1 REM E2 E1 + E2	 integer remainder

	+ E1	
	E1 - E2	
	- E1	
	ABS E1	absolute value
relational	E1 = E2	
	E1 ~= E2	not equal
	E1 < E2	
	E1 <= E2	
	E1 > E2	
	E1 >= E2	
shift	E1 << E2	left shift by E2(>=0) bits
	E1 >> E2	right shift by E2(>=0) bits
logical	~ E1	not (complement)
	E1 & E2	and
	E1 \ E2	inclusive or
	E1 EQV E2	bitwise equivalence
	E1 NEQV E2	bitwise not-equivalence (exclusive or)
conditional	E1 -> E2, E3	
field selection	SLCT K0:K1:K2	specification
	K1 OF E1	application
table	TABLE K0,K1,K2,...	
valof	VALOF command	

The relative binding power of the operators is as follows:

(highest, most binding)	function call
	! (subscripting)
	@ ! ABS OF
	* / REM
	+ -
	relationals
	shifts (see section 2.34)
	~
	&
	\
	EQV NEQV
	-> SLCT
	TABLE
(lowest, least binding)	VALOF

Operators of equal binding power associate to the left. For example, X + Y - Z is equivalent to (X + Y) - Z.

In order that the rule allowing the omission of most semicolons should work properly, a diadic operator may not be the first symbol on a line.

The function call will be described with the function

definition in section 2.66, and the VALOF expression will be described with the RESULTIS command in section 2.55.

2.31 Addressing operators

A powerful pair of operators in BCPL are those which allow one to generate and use addresses. An address may be manipulated using integer arithmetic and is indistinguishable from an integer until it is used in a context which requires an address. If the value of a variable X is the address of a word in storage, then X+1 is the address of the next word.

If V is a variable, then associated with V is a single word of memory, which is called a cell. The contents of the cell is called the value of V and the address of the cell is called the address of V.

An address may be used by applying the indirection operator (!). The expression !E1 has, as value, the contents of the cell whose address is the value of the expression E1. Only the low-order 15 bits of E1 are used (on the Nova).

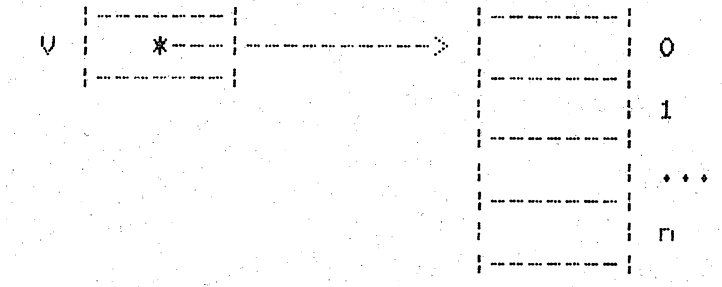
An address may be generated by means of the operator @. The expression @E1 is only valid if E1 is one of the following:

(1) An identifier (not declared by a manifest declaration), in which case @V is the address of V.

(2) A subscripted expression, in which case the value of @E1!E2 is E1+E2.

(3) An indirection expression, in which case the value of @!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 value of V is the address of word 0 of the vector.



The value of the expression V!E1 is the contents of cell number E1 of vector V, as one would expect. The address of this cell is the value of V + E1 hence

@(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 @ and ! are inverse.

The interpretation of !E1 depends on context, as follows:

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

!E1 := E2

E1 is evaluated to produce the address of a cell and E2 is stored in it

(2) @(!E1) = E1 as noted above.

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

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

As a summarizing example, consider the four variables A, B, C and D with initial values @C, @D, 5 and 7, respectively. Then, after the assignment

A := B

their values will be @D, @D, 5, 7.

If, instead, the assignment

A := !B

had been executed, then the values would have been 7, @D, 5, 7.

Finally, if

!A := B

had been executed, then the values would have been @C, @D, @D, 7.

Note that

@A := B

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

2.32 Arithmetic operators

The arithmetic operators *, /, REM, +, -, and ABS act on 16 bit quantities (on the Nova) interpreted as integers.

The operators * and / denote integer multiplication and division. The operator REM yields the integer remainder after dividing the left hand operand by the right hand one. If both operands are positive the result will be positive, it is otherwise implementation dependent (but both remainder and dividend have the same sign on the Nova).

The operators + and - may be used in either a monadic or diadic context and perform the appropriate integer arithmetic operations.

The monadic operator ABS yields the absolute value of an integer number.

The treatment of arithmetic overflow is undefined.

2.33 Relations

A relational operator compares the integer values of its two operands and yields a truth-value (TRUE or FALSE) as result. The operators are as follows:

=	equal
~=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

The operators = and ~= make bitwise comparisons of their operands and so may be used to determine the equality of values regardless of the kind of objects they represent.

An extended relational expression such as

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

is equivalent to

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

Note that the expression between the two relations may be evaluated twice!

2.34 Shift operators

In the expression E1 << E2 (E1 >> E2), E2 must evaluate to yield a non-negative integer. The value is E1, taken as a

bit-pattern, shifted left (or right) by E2 places. Vacated positions are filled with 0 bits.

Syntactically, the shift operators have lower precedence on the left than relational operators but greater precedence on the right. Thus, for example,

```
A << 10 = 14
```

is equivalent to

```
(A<<10) = 14
```

but

```
14 = A << 10
```

is equivalent to

```
(14=A) << 10
```

2.35 Logical operators

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

```
E1 \ E2 & ~E3
```

is as follows:

> E1 is evaluated; if true the whole expression is true, otherwise E2 is evaluated; if false the whole expression is false, otherwise E3 is evaluated; if false the whole expression is true, otherwise the whole expression is false.

In a 'bit' context, the operator ~ (NOT) causes bit-by-bit complementing of its operand. The other operators combine their operands bit-by-bit according to the following table:

Operands		Operator			
		&	\	NEQV	EQV
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	1

2.36 Conditional operator

The expression

E1 -> E2, E3

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

2.37 TABLE

The value of the expression

TABLE K0, K1, K2, ...

is the address of a static vector of cells initialised to the values of K0, K1, K2, ... which must be constant expressions.

2.38 Constant expressions

A constant expression is any expression involving only numbers, character constants, names declared by manifest declaration, TRUE, FALSE, and all arithmetic, relational, shift, logical, and conditional operators.

2.39 Field selectors

Field selectors allow quantities smaller than a whole word to be accessed with reasonable convenience and efficiency. A selector is applied to a pointer using the operator OF (or ::). The selector has three components: size, shift, and offset. The size is the number of bits in the field; the shift is the number of bits between the rightmost bit of the field and the right hand end of the word containing it; the

offset is the position of the word containing the field relative to the pointer.

The precedence of OF is the same as that of the subscription operator (!), but its left operand (the selector) must be a constant expression. A selector is specified using the operator SLCT, whose syntax is as follows:

```
<constant expression> ::= SLCT <size>:<shift>:<offset> \
                          SLCT <size>:<shift> \
                          SLCT <size>
```

where <size>, <shift>, and <offset> are constant expressions. Unless explicitly specified, the shift and offset values are assumed to be zero. A size of zero indicates that the field extends to the left hand end of the word. Selectors are best defined using manifest declarations (see section 2.62).

A selector application may be used on the left hand side of an assignment and in any other context where an expression may be used, except as the operand of @. In the assignment

```
F OF V := E
```

the appropriate number of bits from the right hand end of E are assigned to the specified field. When

```
F OF V
```

is evaluated in any other context, the specified field is extracted and shifted so as to appear at the right hand end of the result.

Judicious use of field selectors rather than inline shifting (e.g., FLAGS OF WORD rather than (WORD ! 2) >> 6 & #17) will increase readability, and can substantially decrease problems in rearranging data structures or transferring to a machine with a different word size. Opportunities for compiler optimization are also improved.

2.4 Section brackets

Blocks, compound commands, and some other syntactic constructions use the symbols $(and $) which are called opening and closing section brackets.

A section bracket may be tagged with a sequence of letters, digits, periods, and underlines (the same characters as are used in identifiers). A section bracket immediately followed by a space or newline is, in effect, tagged with null.

An opening section bracket can be matched only by an identically tagged closing bracket. When the compiler finds

a closing section bracket with a non-null tag, if the nearest opening bracket (smallest currently open section) does not match, that section is closed and the process repeats until a matching opening section bracket is found.

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

2.5 Commands

The complete set of commands is shown here, with E, E1, E2, and K denoting expressions, C, C1 and C2 denoting commands, and D1 and D2 denoting declarations.

routine call	E(E1, E2, ...) E()
assignment	<left hand side list> := <expression list>
conditional	IF E THEN C UNLESS E THEN C TEST E THEN C1 OR C2
repetitive	WHILE E DO C UNTIL E DO C C REPEAT C REPEATWHILE E C REPEATUNTIL E FOR N = E1 TO E2 BY K DO C FOR N = E1 TO E2 DO C
resultis	RESULTIS E
switchon	SWITCHON E INTO <compound command>
transfer	GOTO E FINISH RETURN BREAK LOOP ENDCASE
compound	$(C1; C2; ... $)
block	$(D1; D2; ...; C1; C2; ... $)

Discussion of the routine call is deferred until section 2.66 where function and routine declarations are described.

2.51 Assignment

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

forms:

(1)	The identifier of a variable	<identifier>
(2)	A subscripted expression	E3!E4
(3)	An indirection expression	!E3
(4)	A field selection expression	K OF E3

In case (1) the cell belonging to the identifier is updated. Cases (2) and (3) have been described in section 2.31, and case (4) was discussed in section 2.39.

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
```

2.52 Conditional commands

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

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

2.53 FOR command

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

N must be an identifier and K must be a constant expression. This command will be described by showing an equivalent block.

```
$(
    LET  N, t = E1, E2
    UNTIL N > t DO $(
        C
        N := N + K
    $)
$)
```

If the value of K is negative the relation N > t is replaced by N < t. The declaration

```
LET N, t = E1, E2
```

declares two new cells with identifiers N and t; t being a

new identifier that does not occur in C. Note that the control variable N is not available outside the scope of the command.

The command

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

is equivalent to

```
FOR N = E1 TO E2 BY 1 DO C
```

2.54 Other repetitive commands

```
WHILE  E  DO  C
UNTIL  E  DO  C
C  REPEAT
C  REPEATWHILE  E
C  REPEATUNTIL  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, and so C is executed at least once. In the case of

```
C REPEAT
```

there is no condition and termination must be by a transfer or RESULTIS command in C. C will usually be a compound command or block.

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

```
IF E THEN C REPEAT
```

is the same as

```
IF E THEN $( C REPEAT $)
```

and

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

is the same as

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

2.55 RESULTIS command and VALOF expression

The expression

VALOF C

where C is a command (usually a compound command or block) is called a VALOF expression. It is evaluated by executing the commands (and declarations) in C until a RESULTIS command

RESULTIS E

is encountered. The expression E is evaluated, its value becomes the value of the VALOF expression and execution of the commands within C ceases.

A VALOF expression must contain one or more RESULTIS commands and one must be executed.

In the case of nested VALOF expressions, the RESULTIS command terminates only the innermost VALOF expression containing it.

2.56 SWITCHON command

SWITCHON E INTO <compound command>

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

2.57 Transfer of control

GOTO E
FINISH
RETURN
BREAK
LOOP
ENDCASE

The command GOTO E interprets the value of E as an address, and transfers control to that address, see section 2.67. The

command FINISH causes an implementation dependent termination of the entire program. RETURN causes control to return to the caller of a routine. BREAK causes execution to be resumed at the point just after the smallest textually enclosing repetitive command. The repetitive commands are those with the following key words:

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

LOOP causes execution to be resumed at the point just before the end of the body of a repetitive command. For a FOR command it is the point where the control variable is incremented, and for the other repetitive commands it is where the condition (if any) is tested. ENDCASE causes execution to be resumed at the point just after the smallest enclosing SWITCHON command.

2.58 Compound command

A compound command is a sequence of commands enclosed in section brackets.

```
$( C1; C2; ...  $)
```

The commands C1, C2, ... are executed in sequence.

The operator <> has a similar meaning to semicolon but is syntactically more binding than DO, OR, REPEAT, etc. For example,

```
IF E DO C1 <> C2
```

is equivalent to

```
IF E DO $( C1 ; C2 $)
```

2.59 Block

A block is a sequence of declarations followed by a sequence of commands enclosed together in section brackets.

```
$( D1; D2; ...  ; C1; C2; ...  $)
```

The declarations D1, D2, ... and the commands C1, C2, ... are executed in sequence. The scope of an identifier (i.e., the region of program where the identifier is known) declared in a declaration is the declaration itself (to allow recursive definition), the subsequent declarations and the commands of the block. Notice that the scope does not include earlier declarations or extend outside the block.

2.6 Declarations

Every identifier used in a program must be declared explicitly. There are 10 distinct declarations in BCPL:

> global, manifest, static, dynamic, vector, function, routine, formal parameter, label and for-loop control variable.

The declaration of formal parameters is covered in sections 2.66 and 2.67, and the for-loop is described in section 2.53

The scope of identifiers declared at the head of a block is described in the previous section.

2.61 Global

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

```
GLOBAL $( Name : constant-expression $)
```

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

Global declarations may be combined.

```
GLOBAL $( N1:K1; N2:K2; ...; Nn:Kn $)
```

is equivalent to

```
GLOBAL  $( N1:K1  $)
GLOBAL  $( N2:K2  $)
...
GLOBAL  $( Nn:Kn  $)
```

2.62 Manifest

An identifier may be associated with a constant by the declaration

```
MANIFEST $( Name = constant-expression $)
```

An identifier declared by a manifest declaration may only be used in contexts where a constant would be allowable. It may not, for instance, appear on the left hand side of an assignment. Like global declarations, manifest declarations may be combined.

```
MANIFEST $( N1=K1; N2=K2; ...; Nn=Kn $)
```

is equivalent to

```
MANIFEST  $(  N1=K1  $)
MANIFEST  $(  N2=K2  $)
...
MANIFEST  $(  Nn=Kn  $)
```

2.63 Static

A variable may be declared and given an initial value by the declaration

```
STATIC $( Name = constant-expression $)
```

The variable that is declared is static, that is it has a cell permanently allocated to it throughout the execution of the program (even when control is not dynamically within the scope of the declaration). Like global declarations, static declarations may be combined.

```
STATIC $( N1=K1; N2=K2; ...; Nn=Kn $)
```

is equivalent to

```
STATIC  $(  N1=K1  $)
STATIC  $(  N2=K2  $)
...
STATIC  $(  Nn=Kn  $)
```

2.64 Dynamic

The declaration

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

creates dynamic cells and associates with them the identifiers N1, N2, ..., Nn. These cells are initialized to the values of E1, E2, ..., En. The space reserved for these cells is released when the block in which the declaration appears is left.

2.65 Vector

The declaration

```
LET N = VEC K
```

where K is a constant expression, creates a dynamic vector by reserving K + 1 cells of contiguous storage in the stack, plus one cell which is associated with the identifier N. Execution of the declaration causes the value of N to become the address of the K + 1 cells. The storage allocated is released when the block is left.

2.66 Function and routine

The declaration

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

declares a function named N with m parameters. The parentheses are required even if m = 0. A parameter name has the same syntax as an identifier, and its scope is the expression E. A routine declaration is similar to a function declaration except that its body is a command.

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

If either declaration is within the scope of a global declaration for N, then the global cell will be initialized to the entry address of the function (or routine) before execution of the program. Thus the function may be accessed from anywhere. Otherwise, a static cell is created, is associated with the identifier N, and is initialized to the entry address.

The function or routine is invoked by the call

```
E0(E1, E2, ..., Em)
```

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

```
N(E1, E2, ..., Em)
```

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

Each value passed as a parameter is copied into a newly created cell which is then associated with the corresponding parameter name. The cells are consecutive in store and so the argument list behaves like an initialised dynamic vector. The space allocated for the argument list is released when evaluation of the call is complete. Notice that arguments are always passed by value; the value passed, however, may be an address.

A function call is a call in the context of an expression. If a function is being called, the result is the value of E. A routine call is a call in the context of a command and may be used to call either a function or a routine. A routine call has no result; if called as an expression, the result is undefined.

No dynamic (or vector or formal) variable that is declared outside the function may be referred to from within E.

2.67 Label

A label may be declared by

```
Name:
```

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

```
GOTO Name
```

has the expected effect.

The scope of a label depends on its context. It is the smallest of the following regions of program:

(1) the command sequence of the smallest textually enclosing block,
(2) the body of the smallest textually enclosing VALOF expression or routine,
(3) the body of the smallest enclosing FOR command.

Labels may be assigned to variables and passed as parameters. It is, in general, not useful for them to be declared global, but they can be assigned to global variables.

Using a GOTO command to transfer to a label which is outside the current function or routine will produce undefined (chaotic) results. Such transfers can only be performed using the library functions LEVEL and LONGJUMP which are described in section 2.82

2.68 Simultaneous declaration

Any declaration of the form

```
LET ...
```

may be followed by one or more declarations of the form

```
AND ...
```

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

2.7 Miscellaneous features

2.71 GET directive

It is possible to include a file in the source text of a program using a GET directive of the form:

GET "string"

On the Nova, text of the GET directive is replaced by the text of the file whose filename is "string" (note that the filename extension ".BC" is forced, regardless of any specified). A GET directive should appear on a line by itself.

2.72 Comments and spaces

The character pair // introduces a comment. All characters from (and including) // up to but not including the character 'newline' will be ignored by the compiler. The character pair /* introduces a comment which is terminated by the pair */. This form of comment may extend over several lines.

Blank lines are also ignored.

Space and tab characters may be inserted freely except inside a basic symbol, but space or tab characters are required to separate identifiers or system words from adjoining identifiers or system words.

2.73 Optional symbols and synonyms

The reserved words DO and THEN are synonyms in BCPL. Most implementations of BCPL also allow other synonyms and a list of the synonyms for the Nova implementation can be found in Appendix A.

In order to make BCPL programs easier to read and to write, the compiler allows the syntax rules to be relaxed in certain cases. The word DO (or THEN) may be omitted whenever it is immediately followed by the keyword of a command (e.g. RESULTIS). Any semicolon occuring as the last symbol of a line may be omitted. As an example, the following two programs are equivalent:

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

2.74 SECTION and NEEDS directives

SECTION and NEEDS directives can only occur at the very start of the BCPL program (before any commands, definitions, or GET directives). Each directive must be followed by a string containing a module name (e.g., an external symbol in loader terms). Any number of these directives can appear at the beginning.

SECTION "name" defines a module name to be associated with the entry point of the current segment of code. SECTION directives are only meaningful at the start of a segment of code intended for inclusion in a library.

NEEDS "name" specifies a module that is used by the current segment of code and that this module should be loaded from any library scanned unless the name has already been defined (in another section of code).

2.8 The run-time library

This section summarizes a number of the library functions and routines available on the Nova implementation of BCPL. The routines decribed here are considered standard in that they will usually be found on any implementation. A number of additional routines, as well as more complete descriptions of these routines, are found in section 7.

2.81 Basic Input-Output routines

The input/output facilities of BCPL are quite primitive and simple, and are always invoked by means of function or routine calls.

FINDINPUT(filename) is a function taking a filename string as argument and returning a file descriptor to be used by the input routines. The file is opened for "fast" reading. If the file does not exist, the result is a negative error number.

SELECTINPUT(file-descriptor) is a routine which selects the specified input file for future reading.

RDCH() is a function whose result is the next character from the currently selected input file. If the file is exhausted, it yields ENDSTREAMCH(=-1). The variable CH contains this character when the routine exits. See section 7.46 for a comment regarding compatibility with other implementations.

UNRDCH() is a routine that will cause the next call of RDCH to yield the same character that it returned on its last call for the currently selected input file.

REWIND() repositions the currently selected input file to point to the first record.

ENDREAD() closes the currently selected input file.

FINDOUTPUT(filename) is a function taking a filename string as argument and returning a file descriptor to be used by the output routines. The file is opened for "fast" writing. If the file already exists, the subsequent output is appended to it; otherwise a file of the specified name is created.

SELECTOUTPUT(file-descriptor) is a routine which selects the specified output file for future writing.

WRCH(C) will write the character C to the currently selected output file.

ENDWRITE() closes the currently selected output file.

ENDTOINPUT() closes the currently selected output file and reopens it for reading.

INPUT() is a function that will return with the currently selected input file descriptor.

OUTPUT() is a function that will return with the currently selected output file descriptor.

2.82 Other useful subroutines

PACKSTRING(V,S) is a function which packs the characters V!1 to V!N into S, where N = V!0 & 255. The result is the subscript of the highest element of S used (i.e. N/2 on the Nova).

UNPACKSTRING(S,V) is a routine to unpack characters from the string S into V!1 to V!N when N is the length of the string, and set V!0 = N.

GETBYTE(S,I) is a function whose result is the Ith character of the string S. By convention the zeroth character of a string is its length.

PUTBYTE(S,I,C) is a routine which will update the Ith character of the string S with C.

WRITES(S) writes the string S to the current output stream.

NEWLINE() writes a newline to the current output stream.

WRITED(N,D) writes the integer N to the current output stream right justified in a field of width D places. If D is too small the number is written correctly using as many characters as necessary.

WRITEN(N) is equivalent to WRITED(N,0).

READN() is a function that reads a decimal number from the current input stream leaving the terminating character in CH.

WRITEOCT(N,D) writes the D least significant octal digits of N to the current output stream.

WRITEHEX(N,D) writes the D least significant hexadecimal digits of N to the current output stream.

WRITEF(FORMAT,A,B, ...) is a routine to output A,B, ... to the current output stream according to FORMAT. The FORMAT string is copied to the stream until the end is reached or the warning character '%' is encountered. The character following the '%' defines the format of the next value to be printed as follows:

%%	print '%'
%S	print as a string
%C	print as a character
%N	print as a integer (minimum width)
%In	print as a integer width n
%On	print as an octal number width n
%Xn	print as a hexadecimal number width n

In the last three cases the width n is represented by a single hexadecimal digit. The routine takes the format and a maximum of 11 additional arguments.

MAPSTORE() prints a map of the program area including function and routine names, and the values of all global variables used.

BACKTRACE() prints a summary of the dynamic stack giving the names of all functions and routines currently active and the values of the first few local variables of each.

ABORT(CODE) is called automatically by the system after most faults. It can call BACKTRACE and MAPSTORE in order to provide the user with some postmortem information.

STOP(N) will terminate the job step, returning a completion code N.

LEVEL() is a function whose result is the current value of the run-time stack pointer for use with LONGJUMP. The stack pointer changes only when a function or routine is entered or left.

LONGJUMP(P,L) will cause a non-local jump to the label L at the activation level given by the stack pointer P.

APTOVEC(F,N) is a function which will apply F to two arguments V and N where V is a vector of size N. APTOVEC

could (illegally) be defined in BCPL as follows:

```
LET APTOVEC(F,N) = VALOF $(
    LET V = VEC N
    RESULTIS F(V,N)
$)
```

3 BCPL on the Nova -- An Overview

The files relating to the system are listed below. Users can link to the files in the normal way or use the CLI commands in LINKBCPL.MC which can be executed by typing:

<master-directory>:LINKBCPL

The relevant files are:

LINKBCPL.MC	-	CLI macro for linking
BCPL.SV	-	the command program
BCOMP.SV(.OL)	-	the compiler
BXREF.SV	-	the cross-reference program
BCGN2.SV	-	the code generator
BCDEB.RB	-	the debugging package
BCPLIB.LB	-	the BCPL library
(X)LIBHDR.BC	-	the standard library headers

Further details are given in section 4.1.

3.1 Compilation

The BCPL compiler is usually invoked by use of the command program BCPL.SV. This program calls the compiler + code generator + assembler to compile a BCPL source program into a relocatable binary file. A typical command might be

```
BCPL/S PROG $LPT/L
```

Further details are given in section 4.1.

3.11 Library declarations

The directive

```
GET "LIBHDR"
```

will insert the standard library declarations from the file whose filename is LIBHDR.BC. A listing of the standard library header can be found in Appendix C. Note also that many of the standard support routines require the presence of a "NEEDS" directive in order to load them from the library. The use of XLIBHDR is discussed in section 9.5.

3.12 Diagnostics

The BCPL compiler has three passes: parse, translate and code-generate. There are correspondingly three kinds of error diagnostic.

A parse diagnostic occurs when a relatively simple syntactic error is detected during the first pass of compilation. The message includes a portion of the source program to give the context and a brief description of the probable error. The compiler usually skips to the end of the line before continuing the parse. Later error messages should be viewed with suspicion since the automatic recovery is often not very successful.

Translation phase diagnostics occur in the second pass of compilation and report errors such as the use of an undeclared identifier. Each error is briefly described and a representation of the relevant portion of the parse tree is printed.

Code-generation diagnostics are rare and usually result from table overflows or compiler errors.

3.13 Compilation options

The compilation of a program can be controlled by various compilation options passed to the compiler by the CLI program (in COM.CM). The options for the code-generator are generally different from those of the compiler, and not all of the options can be specified to BCPL.SV. Some users may therefore need to run the compiler (BCOMP.SV) or the code generator (BCGN2.SV) separately in some circumstances. All options are specified by single letters and some are primarily debugging aids for compiler maintenance. Further details are given in sections 4.1 - 4.4.

3.2 Execution

When the program has been compiled and subsequently loaded into a save file it can be executed directly. When running under the CLI it can be run like any other save file. This section describes loading and running a program.

3.21 Loading

Loading is achieved by use of the standard relocatable loader RLDR. The library BCPLIB.LB should be scanned last to load modules that define outstanding external names. Further details are given in sections 4.6 and 4.7. Note that a minimum of two tasks ("2/K") must be specified to RLDR, even if the program is not a multi-tasking one.

3.22 Execution faults

In the event of an execution fault such as division by zero or stack overflow the routine ABORT is called. This will

print the fault number and, depending on bits set in PM.SET, may be followed by a summary of the run-time stack (printed out by BACKTRACE) and a map of the program store and globals (printed out by MAPSTORE). This output is always sent to the file "$LPT". BCPL run-time error messages are listed in Appendix D.

3.23 A demonstration program

Consider the following program held in the file TESTPROG.BC:

```
// THIS IS A DEMONSTRATION BCPL PROGRAM

NEEDS "BCPLC"

GET "LIBHDR"

// THIS INSERTS THE STANDARD GLOBAL DECLARATION

GLOBAL $( TREE:100; TREEP:101 $)

STATIC $( COUNT=0; MIN=0; MAX=0  $)

MANIFEST $(  // THE FOLLOWING NAMES WILL
             // BE USED AS SELECTORS
VAL=0; LEFT=1; RIGHT=2
$)

// THE FUNCTIONS PUT, LIST AND SUM(DEFINED BELOW)
// OPERATE ON A TREE STRUCTURE WHOSE ROOT IS HELD
// IN TREE.  IF T IS A BRANCH IN THIS TREE THEN
// EITHER T=0
// OR  T POINTS TO A TREE NODE AND VAL!T IS AN
//     INTEGER(K SAY), LEFT!T IS A BRANCH CONTAINING
//     NUMBERS <K AND RIGHT!T IS A BRANCH CONTAINING
//     NUMBERS >=K.


LET PUT(K, P) BE  // THE ROUTINE PUT WILL ADD A NODE TO THE
                  // TREE WHOSE ROOT IS POINTED TO BY P.

   $(P UNTIL !P=0 DO
         $( LET T = !P
            P := K<VAL!T -> @LEFT!T, @RIGHT!T  $)

       VAL!TREEP, LEFT!TREEP, RIGHT!TREEP := K, 0, 0
       !P := TREEP
       TREEP := TREEP + 3  $)P

AND LIST(T) BE  // LIST THE NUMBERS HELD IN THE TREE T
     UNLESS T=0 DO $( LIST(LEFT!T)
                      IF COUNT REM 10 = 0 DO NEWLINE()
                      COUNT := COUNT + 1
                      WRITEF(" %I6", VAL!T)
                      LIST(RIGHT!T)  $)
```

```
AND SUM(T) = T=0 -> 0,
             VAL!T<MIN -> SUM(RIGHT!T),
             VAL!T>MAX -> SUM(LEFT!T),
             VAL!T+SUM(LEFT!T)+SUM(RIGHT!T)

AND START() BE $(1

LET V = VEC 600
TREE, TREEP := 0, V
RDCH()              // THIS IS A CONVENIENT WAY
                    // TO ORGANISE A TEST PROGRAM
$( SWITCHON CH INTO $(S

   CASE 'Q': CASE ENDSTREAMCH:
             WRITES("*NEND OF TEST*N")
             FINISH

   CASE 'P': PUT(READN(), @TREE)  // PUT A NUMBER
             LOOP

   CASE 'L': NEWLINE()  // LIST THE NUMBERS IN THE TREE
             COUNT := 0
             LIST(TREE)
             NEWLINE()

   CASE 'S': MIN := READN()
             MAX := READN()
             WRITEF("*NSUM OF NUMBERS FROM %N TO %N IS %N*N",
                    MIN, MAX, SUM(TREE))
             LOOP

   CASE 'M': MAPSTORE(); ENDCASE  // PRINT A STORE MAP

   CASE 'Z': TREE := 0; WRITES("*NTREE CLEARED*N"); ENDCASE

   CASE '*S': CASE '*N': ENDCASE  // IGNORE SPACE AND NEWLINE

   DEFAULT: WRITEF("*NBAD CH '%C'*N", CH); ENDCASE

   $)S

   RDCH()
 $) REPEAT

 $)1   // END OF PROGRAM
```

A typical console dialogue to compile, load, and run the above program stored in the file TESTPROG.BC is reproduced below.

```
>> BCPL TESTPROG
* * * BCPL FILE 'TESTPROG.BC' COMPILED AT <time/date>

FINNING BCPL COMPILER, REVISION 17.7.77
TREE SIZE = 2713
```

```
NOVA 2 CODE GENERATOR, REVISION 7.6.77
PROGRAM SIZE IS 348 WORDS.
                .TITLE TESTPROG; MAIN PROGRAM (NO OVERLAYS)
[COMPILATION COMPLETED]
>> RLDR/P TESTPROG BCPLIB.LB 2/K
TESTPROG.SV     LOADED BY RLDR REV 05.00 AT <time/date>
 TESTP    000445
 NODEB    001624
 BCPLC    001624
 BCPLI    003146
 BCPLX    004337
 MULT     005254
 DIVRE    005273
 RSHIF    005334
 DRSCH    005370
 TRSCH    005370
 KILL     005400
 ATCBM    005400
 NSAC3    006025
 DUMMY    006025


    NMAX 006034
    ZMAX 000336
    CSZE 000000
     EST 000000
     SST 000000

>> TESTPROG

P-1 P54 P3 P80 P34 P-4 P-3 P10 P20
L

   -4      -3      -1       3      10      20      34      54      80
S0 100

SUM OF NUMBERS FROM 0 TO 100 IS 201
H
BAD CH 'H'
Q
END OF TEST
>>
```

4 How to use BCPL on the Nova

4.1 Simple use

The "front-end" program BCPL.SV is normally used to compile BCPL source programs on the Nova. This program in turn calls the compiler, code generator, and macro assembler to produce corresponding relocatable binary files which may then be loaded by the standard system loader (RLDR.SV). In the simplest case,

BCPL PROG

would be sufficient to compile the source program PROG.BC (note that the ".BC" extension is always required) into the relocatable binary file PROG.RB. Intermediate files PROG.OC and PROG.SR are created and subsequently deleted (unless global option "/I" is specified) when no longer required. BCPL.SV references the files BCOMP.SV, BCOMP.OL, BXREF.SV, BCGN2.SV, MAC.SV, MACXR.SV, and MAC.PS. Entries for these files must therefore exist in the current directory, a situation which is most easily achieved by use of the command "LINKBCPL" (see section 4.13).

4.11 Global options

A	assembly code listing (only if local "/L" present)
B	brief listing only (do not list BCPL source)
C	insert call counting (implies "/N")
F	generate fast code (default = compact code)
I	retain intermediate files (-.OC, -.SR)
L*	generate literal OCODE
M*	MAPSTORE after compilation
N	insert routine names
O	compile as overlay (if "/S"), or to use overlays
P	insert profile counting (implies "/C" and "/N")
R	produce cross-reference list on listing file
S	compile as section (i.e., not as main program)
V	generate stack overflow test code
X	syntax check only (no OCODE produced)

Options marked with "*" are primarily of use for compiler maintenance.

4.12 Local options

G	set maximum global number to specified value (default = 255)
L	listing file (default = CONSOLE with "/B" assumed)
S	set initial stack size to specified number of words (default = all of memory except for 1024 words of free vector space) "0/S" specifies 2048

words of stack space

U use specified (single) letter to generate unique section labels (default = "A")

Z set maximum pass zero global number to specified value (default = 155)

An argument without a local option switch is assumed to be the BCPL source filename. The extension ".BC" is used regardless of any extension specified.

4.13 LINKBCPL

The file LINKBCPL.MC contains a sequence of CLI commands to link all the files necessary for a BCPL compilation into the current directory. Assuming that the master directory contains the necessary entries, execution of the following command:

<master-directory>:LINKBCPL

will leave the current directory with sufficient facilities to use all of the BCPL system.

4.2 Stages of compilation

There are five distinct phases in a BCPL compilation. These are, in order:

a)	compilation	BCPL source to OCODE
b)	cross-reference	(optional) listing of name references
c)	code generation	OCODE to assembler source
d)	assembly	assembler source to relocatable binary
e)	loading	relocatable binary to "save" file

Each phase may be performed separately, as described in the remaining portions of section 4.

A typical sequence of commands to compile a program held in the file FOO.BC might be:

```
BCOMP FOO $LPT/L
BXREF FOO
BCGN2/F/P/V FOO $LPT/L
MAC FOO
RLDR/P FOO BCDEB BCPLIB.LB 2/K
```

This would result in FOO.SV ("save" file format). Note that the first four lines of this sequence (i.e., everything except the load command) would normally be replaced by a single simple command to BCPL.SV.

4.3 Compilation

The BCPL compiler is held in the file BCOMP.SV, with overlays in the file BCOMP.OL (this name may not be changed). The compiler overlays itself in two internal phases: the first reads the source text into memory and builds a syntax tree containing a description of the program with a set of accompanying definitions; the second translates this tree into the machine-independent intermediate code OCODE. The same compiler can be used to produce OCODE for other computers if their code generators are available. The compiler reads BCPL source from the file <source>.BC and writes OCODE to the file <source>.OC. Error messages and informative comments are sent to the listing file (see local option "L").

4.31 Global options

A*	list AE tree
B	brief listing only (do not list BCPL source)
D*	set PPDEBUG
E	inhibit page eject at start of listing
G	no GET directives are obeyed
L*	generate literal OCODE
M*	MAPSTORE after compilation
T*	set PPTRACE

Options marked with "*" are primarily of use for compiler maintenance.

4.32 Local options

A	append output to specified file
L	listing file (default = CONSOLE with "/B" assumed)
O	write OCODE output to specified file

An argument without a local option switch is assumed to be the BCPL source filename. The extensions ".BC" for input and ".OC" for output are used regardless of any extensions specified.

4.4 Cross-reference listing generation

An optional step in program production is the generation of a cross-reference listing of the source program. This is performed by a utility program held in the file BXREF.SV, which produces an alphabetical listing of all names used and the line number of each occurrence. Note that no distinction is made between occurrences of the same name appearing in different scopes.

4.41 Local options

C	restrict output line length to specified number of characters (default = 120)
L	listing file (default = "$LPT")

4.5 Code generation

The standard (NOVA 2) code generator for the Nova family of computers is held in the file BCGN2.SV. This name may be changed if desired, except for use with BCPL.SV. The code generator reads OCODE from the file <source>.OC and produces an assembly language source file in <source>.SR as output. This file is complete in all respects, and may be edited by the user if desired. Error messages and informative comments are sent to the listing file (see local option "L").

4.51 Global options

C	insert call counting (implies "/N")
D	inhibit definitions at start of code
E	inhibit ".END" at end of code
F	generate fast code (default = compact code)
L*	accept literal OCODE
M*	MAPSTORE after code generation
N	insert routine names
O	compile as overlay (if "/S"), or to use overlays
P	insert profile counting (implies "/C" and "/N")
S	compile as section (i.e., not as main program)
V	generate stack overflow test code
X*	produce code generation statistics

Options marked with "*" are primarily of use for code generator maintenance.

4.52 Local options

A	append output to specified file
G	set maximum global number to specified value (default = 255)
L	listing file (default = CONSOLE)
M	write macro assembler source to specified file
O	use specified name as overlay file title (".OL" extension forced; default = same name as assembler source)
S	set initial stack size to specified number of words (default = all of memory except for 1024 words of free vector space) "0/S" specifies 2048 words of stack space
T	use specified name as program title (.TITL) (default = same name as assembler source)
U	use specified (single) letter to generate unique

section labels (default = "A")
Z set maximum pass zero global number to specified value (default = 155)

An argument without a local option switch is assumed to be the OCODE filename. The extensions ".OC" for input and ".SR" for output are used regardless of any extensions specified.

4.53 Code generation error messages

Apart from errors detected while decoding the command line, code generator error messages are relatively rare. Possible user errors include:

a) code generating a file that does not contain OCODE

b) program too large or too complex (i.e., tables full)

Other than in these circumstances, messages usually indicate malformed OCODE (possibly due to an error in the compiler) or a bug in the code generator itself. Full details, including listings of source, OCODE, and code produced should be reported.

4.6 Assembly

Assembly is normally performed by the standard Nova-family macro assembler (MAC.SV). As such, all the usual options are available as described for this program. A typical command, therefore, might be:

```
MAC FOO $LPT/L
```

As the file FOO.SR will normally contain a .END directive, user-produced assembly code modules cannot be inserted following the program unless code generator local option E is used.

4.61 Assembly error messages

Apart from errors detected while decoding the command line, assembly error messages are rare. An exception is a message indicating that the symbol "GN<number>" was not defined. This will occur if the values specified by code generator local options G and Z were not sufficiently large, and the code generation should be repeated with increased values. Errors can also result from misuse of code generator global options D and E. Other errors usually indicate code generator bugs, which should be reported with full evidence.

4.7 Loading to a save file

Relocatable binary modules produced by the assembly phase can be loaded into executable save files by the standard relocatable loader RLDR. The run-time system is linked in at this time, and some care is necessary to ensure that modules are loaded in the correct order. In general, the load command line format will be as follows:

RLDR n/K <all BCPL modules> BCPLIB.LB <all non-BCPL modules>

All BCPL modules (including any assembly language routines written to the specifications of section 6.6) must be grouped together, and must precede the reference to the standard BCPL library (BCPLIB.LB) in order to properly initialize globals. The first module in this group must be the main program (i.e., compiled without the global S switch) to provide a starting address for the save file.

The number of tasks specified in the load command (n/K) depends on user requirements, but must be at least two or greater. The following load line:

RLDR/P 2/K FOO SEC1 SEC2 BCPLIB.LB $LPT/L

will create a save file FOO.SV consisting of the three BCPL modules FOO.RB, SEC1.RB, and SEC2.RB, plus all required components of the run-time library.

In general, loading and linking is governed by the declaration (NEEDS) and supply (SECTION) of external symbols. An additional technique is available, however, to force loading of the debugging procedure module ("BCPLD"). This may be brought in by including the name "BCDEB" among the group of BCPL modules, which may occasionally be more desireable than the usual method of compiling a NEEDS "BCPLD" directive.

4.71 Load error messages

Most errors are the result of undefined external names. If these are system names it implies that the library was not properly scanned (i.e., BCPLIB.LB was omitted from or improperly placed in the RLDR command). Alternatively, this can be caused by the failure of the user to provide matching SECTION directives for all NEEDS directives. Finally, if the errors concern global symbols (i.e., GNnnn), then either two or more (or zero!) of the specified modules have been compiled as main programs, or the value specified by the code generator local option G was not sufficiently large.

4.72 Loading with overlays

Overlays are loaded in the normal fashion using RLDR, with

the following special considerations:

1) the overlays must appear in the command line immediately following the library (BCPLIB.LB) specification.
2) the library must be specified a second time immediately following the overlay list.

E.g.,

```
RLDR 2/K PROG BCPLIB.LB [OV0,OV1A OV1B,OV2] BCPLIB.LB
```

For more information concerning the use of overlays, see section 9.1.

5 Code Generation

5.1 Types of code generator

The description below refers to the standard (NOVA 2) code generator BCGN2.SV. This is a sequential code generator producing editable assembly code as output which may be assembled with the standard Nova assemblers.

5.2 Code for operators

This section details the code that the code generator produces for various operations. This is only intended as a guide and the code produced for some operations will undoubtedly change between different versions of the code generator. Not all operations are dealt with; only those of particular interest are discussed.

5.21 Plus and minus

Plus (+) and minus (-) simply involves loading the two values into registers and then adding or subtracting the values. The main optimisation on this is the case S := S+1 which generates the code:

```
ISZ  S          ; Increment memory
JMP  .+1        ; No operation
```

and S := S-1 which similarly uses the DSZ instruction. This saves between one and two instructions and is worthwhile as S := S+1 or S := S-1 is common in BCPL. In addition the INC operation is used when +1 is needed and the value is in a register.

5.22 Multiplication

Multiplication (*) is usually done by entry to a subroutine whose address is on page 0. The values are loaded into registers AC0 and AC1; register AC3 is then cleared and a "JSR @" instruction compiled. The result is returned in AC1. The actual nature of the subroutine depends on the hardware available on the machine and this method is used to avoid changing the code generator to take account of different hardware.

Note that subroutine calls to this routine and to those described below in sections 5.23 and 5.24 are standard JSR calls expecting the link in AC3 unlike the standard BCPL convention for function/routine calls.

5.23 Division and remainder

Division (/) and remainder (REM or %) are generated in the same way as multiply above. The dividend is loaded into AC1 and the divisor into AC0; register AC3 is cleared and a "JSR @" instruction generated via a location on page 0. The same routine is used for divide or remainder and the two results are returned in registers AC0 (the remainder) and AC1 (the quotient). The division is a signed FORTRAN type division; i.e., the remainder and the quotient always have the same sign and division is by truncation towards zero. However, users who require machine independence should avoid assumptions of this nature as this is not defined in the language. Nevertheless, for this implementation the equation

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

holds for all values of A and B.

5.24 Shifts

The logical left and right shifts in BCPL are generated in a similar fashion to the multiplication above. The pattern to be shifted is loaded into register AC0 and the amount of the shift into register AC1; register AC3 is freed and a "JSR @" instruction generated to one of two addresses on page 0 (depending on the direction of the shift). The exact definition of the operation in terms of the amount of the shift is:

```
amount <=  0          ->   no effect
1 <= amount <= 15     ->   as expected
16 <= amount          ->   result is 0
```

Constant shift amounts of 0, 1, 2, 8, and 15 result in the generation of one or two in-line instructions which are more efficient than the generalized subroutine call.

5.25 Logical operations

BCPL provides four diadic logical operations: & (LOGAND), \ (LOGOR), NEQV, and EQV. These all involve loading any two registers with the values involved and then generating a code sequence involving these two registers. The code for the & operation is trivial; for the other operations the code is as follows (assuming the values are in registers A and B):

```
\ (LOGOR)
        COM   A, A          ; A := ^A
        AND   A, B          ; B := B&^A
        ADD   A, B          ; Result is in B
```

relying on the equation A\B = A&^B + B

The code for EQV and NEQV involves three resisters and the third resister must be cleared. If this resister is denoted by I then:

```
NEQV
        MOV   A, I          ; I := A
        ANDZL B, I          ; I := 2*(A&B)
        ADD   B, A
        SUB   I, A          ; Result is in A
```

relysing on the equation A NEQV B = A + B - 2*(A&B)

EQV is the same sequence followed by a COM A, A instruction.

5.3 Characters and strings

The character code for the BCPL implementation on the Nova is 7 bit ASCII (American Standard Code for Information Interchange). A complete listing of relevant codes and associated graphics is given in appendix B.

This section describes the representation of characters and strings.

5.31 Characters and escapes

The representation of characters is as in standard BCPL and the value of a character denotation is as given in appendix B (i.e., 'A' = 65 = #101). However, some characters cannot be directly represented in BCPL and these can be escaped into character denotations and strings. The standard escape character is * which in BCPL is represented as '**' for a single * character.

Other denotations that can be represented in escapes are:

'*0'	null (zero)	0
'*G'	bell	7
'*B'	backspace	8(#10)
'*T'	tab	9(#11)
'*L'	line feed	10(#12)
'*V'	vertical tab	11(#13)
'*P'	form feed	12(#14)
'*F'	form feed	12(#14)
'*C'	carriage return	13(#15)
'*S'	space	32(#40)
'*"'	single "	34(#42)
'*''	single '	39(#47)
'**'	single *	42(#52)

The denotation '*N' is the standard notation for newline in this implementation. Currently this is identical to '*L' but programs should not assume anything about the value of '*N'

as this may change in future implementations.

Note that * followed by any other character (including lower case letters) is treated as that character itself. If a string is too long to fit on one line of text it can be split textually by inserting the sequence "*<carriage control><tabs or spaces>*" anywhere in the string; this is completely ignored (tabs or spaces are optional). Otherwise carriage control characters in a string produce a fault.

5.32 String representation

The standard BCPL string representation is used (see section 2.2). Characters are packed two to a word with the most significant character of any pair in the leftmost byte. The first byte of any string contains the count of the number of characters in the string from 0 to 255, so the maximum length of a string is 255 characters. Empty strings (length 0) are allowed. The byte index starts from 0 (as do vector indices) so that the count is held in byte 0, the first character is in byte 1, the second character in byte 2, and so on.

NOTE!

The only departure from standard BCPL is the additional feature that strings are packed with at least one zero byte at the end. This byte is not included in the count and for all standard BCPL purposes is completely ignored. Its purpose is to allow for the manipulation of text elements that are delimited by a null byte (such as operating system filenames).

The string "DATES" will therefore be represented by a pointer to a storage vector packed as follows:

0	5	'D'
1	'A'	'T'
2	'E'	'S'
3	0	0

where the 7th byte is set to zero (null) by default. Note that user routines which create or copy strings should preserve this convention if the null terminator facility is desired.

5.4 Indirection and address operators

5.41 Indirection

The ! operator in BCPL allows the programmer to use a value as an address (cf @ in assembly code). The code currently generated for this operator is not optimal because of some restrictions and difficulties in implementation (which, hopefully, will be eliminated in future versions).

Thus indirection is usually achieved by indexing the value in register AC3. This makes comparisons such as !I=!J quite inefficient.

5.42 Addresses

The @ operator in BCPL allows the programmer to use the absolute address of a variable as a parameter. For global or static variables this address is a load-time constant. However, stack addresses must be calculated at run-time by adding the constant to the stack pointer in AC2.

6 The Machine Code Interface

... this chapter is not yet available.

7 The Standard Finnins Library

This chapter describes the standard library routines provided with the Finnins BCPL system. Many of these routines have identical counterparts in most BCPL systems (see section 2.8), but a number are unique to the Finnins implementation.

Declarations for the standard library are normally contained in the file LIBHDR.BC (although this may be chansed by the user if desired). Prosrams that wish to use this header should contain the directive

```
GET "LIBHDR"
```

at an appropriate place. A listins of this standard header can be found in Appendix C.

The names referenced in this chapter are the standard BCPL function/routine names. These may be chansed by the user if desired, provided that the slobal declarations reference the same number as the standard names (as shown in the LIBHDR listins).

In the detailed descriptions which follow, an "=" sisn preceding a name is indicative of a function, rather than a routine.

7.1 Library linkase

The BCPL library is kept in the standard format library file BCPLIB.LB. It consists of a number of necessary and optional modules which are loaded by the RLDR operation. The order of the modules in the library is sisnificant, and users wishins to add new modules should place them at the besinnins.

Loadins of the modules is directed by "external" names, produced by the code senerator both automatically and in response to NEEDS directives. Call-by-name is not possible due to the orsanization of the BCPL slobal vector, but the available routines are collected into easily specified srours, each loaded with a sinsle NEEDS directive.

7.2 Basic routines

The routines and functions listed in this section are considered fundamental to the BCPL run-time environment, and are loaded automatically from the library, without the necessity of a NEEDS directive.

7.21 START

START() is not defined by the library and must be supplied by the user. It is (by tradition) global vector entry 1, and this is the routine which is entered (by a standard routine call) after run-time initialization is complete. The global number of START cannot be re-defined by the user. Returning from this routine has the same effect as executing a FINISH command.

7.22 STOP

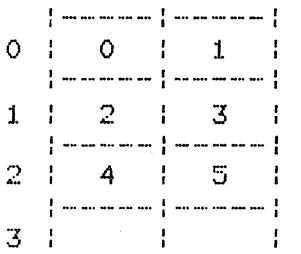
STOP(CODE) is a routine which never returns. It takes a completion code as its single argument; if this is zero, then the program terminates normally (if operating under RDOS, a .RTN is performed). Non-zero completion codes are treated as error messages (.ERTN under RDOS), and may be of two types: negative values are complemented and passed as operating system errors; positive values are incremented by #10000 and passed as BCPL run-time errors (see Appendix D). An example of the use of STOP in an error situation is:

```
INFILE := FINDINPUT("DATAFILE")
IF INFILE < 0 THEN STOP(INFILE)
```

7.23 GETBYTE and PUTBYTE

=GETBYTE(VECTOR,INDEX) is a function which yields a particular byte within a vector. Its first argument is a vector or string address; its second a byte number for that vector or string. The result is the (8-bit) byte specified, which will always be in the range 0-255.

On the Nova (as in most BCPL implementations), bytes are numbered from left to right to correspond with string packing:



If the first argument is a BCPL string, then GETBYTE(V,N) will yield the Nth character of same:

```
GETBYTE("STRING",2)
```

will produce the value 'T'. Also in this situation, N=0 will yield the size of the string (number of characters in the string):

```
GETBYTE("STRING",0)
```

will produce the value 6. Note that this is also the highest byte number of the string (but not including the "invisible" null terminator discussed in section 5.32). This special case of accessing the first byte (length) of a string is more naturally accomplished using the field selector LENGTH, defined in the standard library header. Thus, an expression equivalent to the above would be

```
LENGTH OF "STRING"
```

PUTBYTE(VECTOR,INDEX,BYTE) is a routine which stores a specified byte into a vector or string. The first two arguments are as described for GETBYTE above; the third specifies a byte value (the rightmost 8 bits of the value are used). This routine will update the indicated byte to the new value.

```
LET STRING = "STRING"
PUTBYTE(STRING,2,'P')
```

will change the string addressed by STRING from "STRING" to "SPRING".

7.24 PACKSTRING and UNPACKSTRING

These routines are concerned with the conversion of strings between packed and unpacked formats. The normal packed format of a BCPL string is as defined in section 5.32. Unpacked strings are structured with one byte per word. As in the packed format, the first element (in this case, the first word) is used to contain the string length (i.e., number of characters). The null byte delimiter convention is not used in the unpacked format, so a vector of 256 locations (VEC 255) is sufficient to hold the largest unpacked BCPL string. In packed format, 129 locations (VEC 128) are necessary for the largest string.

=PACKSTRING(VECTOR,STRING) is a function which copies a string from one vector to another, converting from unpacked to packed format in the process. Its result is the subscript of the highest element of the destination vector used (N/2 on the Nova, where N is the number of characters in the string).

UNPACKSTRING(STRING,VECTOR) is a routine which copies a string from one vector to another, converting from packed to unpacked format in the process.

7.25 LEVEL, LONGJUMP, and APTOVEC

=LEVEL() returns as its result a pointer to the stack frame of the calling routine (note that this is the genuine value

of the pointer rather than the contents of AC2, which are offset by 128). The value may be used for LONGJUMP or for debugging purposes such as BACKTRACE.

LONGJUMP(POINTER,ADDRESS) specifies a value for the stack frame pointer (typically as returned by LEVEL), and an address (typically a label) to which control is transferred. The value of the frame pointer must be currently active or the stack will be corrupted when the entered routine attempts to return. If the activation level is not applicable to the routine, the results are undefined; programs should normally ensure that the value used for the frame pointer was obtained in the same routine in which the address label was defined. An illustrative example:

```
GLOBAL $( L:100 ; LABEL:101 $)

LET START() BE $(
   L := LEVEL()
   LABEL:  ....
$)
.
.
.

LONGJUMP(L,LABEL)
```

It is also possible to specify a routine or function address as the second argument (LONGJUMP simulates a routine call to the address with a stack frame size of zero), so the call

```
LONGJUMP(SYSTEM(STACKBASE),START)
```

might be a useful thing to do. The calling routine must not return, however, as there is no longer any place for it to return to!

=APTOVEC(FUNCTION,SIZE) allows the programmer to declare a stack vector whose size is specified at run time. This is normally not allowed in BCPL, as the size of any vector specified by VEC must be known at compile time. A definition of APTOVEC in (somewhat illegal) BCPL terms is:

```
LET APTOVEC(F,N) = VALOF $(
    LET V = VEC N       // define specified vector
    RESULTIS F(V,N)     // apply function to vector
$)
```

The second argument must be a positive integer which is not larger than the space left on the stack (this may be checked by use of the LEVEL() and SYSTEM(STACKTOP) functions). The operation is performed by constructing both the requested vector and an appropriate calling sequence for the referenced function (or routine) on the stack. Any result returned by the function will appear as the result of APTOVEC. Note that stack overflow is checked regardless of the code generator

global "V" option.

7.26 GETVEC and PUTVEC

These routines provide a simple general-purpose memory management system, using a first-fit method with boundary tags, to coalesce blocks being freed with other blocks already free. The internal variable SLOP (which is initialized to 3, its minimum allowable value) is a number such that if N words are requested, and if the first free block of size N or greater is no larger than N+SLOP, then the whole block will be allocated instead of being split up. Larger values of SLOP (which may be set using the SYSTEM function) tend to reduce fragmentation at the expense of unused space in the allocated blocks.

=GETVEC(N) creates a dynamic vector by removing N+1 cells of contiguous storage from the memory management system. The address of the N+1 cells is returned as the value of the function. The storage allocated is released by PUTVEC, in contrast to the VEC allocation described in section 2.65, which is released when the execution block is left.

PUTVEC(VECTOR) is a routine which takes as its argument a pointer to a vector previously allocated by GETVEC; this space is made available for further allocation. Needless to say, grave disorder will result if the space assigned by GETVEC is overrun or if some random number is handed to PUTVEC.

7.27 SYSTEM

=SYSTEM(SPECIFIER,...) allows access to internal variables maintained by the BCPL run-time system. This capability is typically used to determine available space, modify standard parameters (such as the SLOP value used by GETVEC), or for some special-purpose debugging functions. The first argument specifies the operation to be performed, chosen from the following list (the names of which are defined in the standard library header):

GLOBALZBASE returns the base address of the .ZREL globals.
GLOBALNBASE returns the base address of the .NREL globals.
GLOBALBREAK returns the break point of the global vector (i.e., the last global number on page zero).
GLOBALTOP returns the top address of the global vector.
STACKBASE returns the base address of the current task stack.
STACKTOP returns the top address of the current task stack.
STACKSPACE returns the space (in words) remaining on

the current task stack.

VECTORSPACE returns the size of the next vector immediately available from GETVEC (note that this is only significant prior to the execution of any PUTVEC operations, which may fragment the available space).

PROGBASE returns the base address of the program .NREL space.

PROGTOP returns the top address of the program .NREL space (not including space used by stacks or the memory management system).

ADDRESSOFGLOBAL,N returns the address of global number N.

ADDRESSOF,ADDR,AC2,AC3 returns the address referenced by a memory reference instruction (MRI). The second argument is taken as the address of the MRI. If this MRI uses address modes 2 or 3 it is necessary to know the values of registers AC2 and AC3 when the instruction is executed. Arguments three and four are used for these values respectively. If these values are greater than zero they are taken as given; if negative, the routine will return -1 except that, for the contents of AC3, a search is made to see if the register was loaded in the previous 4 instructions, in which case the loaded value is used. Not that the contents of AC2 represent the LEVEL when the instruction was executed; this is the value that should be passed (not the actual contents of AC2), as the routine corrects for the -128 discrepancy.

GROUND returns 0 or 1 if the program is executing in the (RDOS) background or foreground respectively. Zero is returned in a non-RDOS environment.

SETTABWIDTH,N sets the output tab width to N (default value = 8).

SETSLOP,N sets the SLOP value used by GETVEC to N (minimum allowable value = default value = 3).

7.28 Special-purpose basic routines

OVERLAY is described in section 9.1.
SYS is described in section 9.2.
ABORT and PMSET are described in section 8.3.
XIO, ITASK, IENABLE, and IEXIT are described in section 9.5.

7.3 Character (stream) I/O routines

The routines in this and the following section are considered to be "standard" BCPL I/O operations, and as such provide compatibility with other BCPL implementations. These routines are loaded from the library with a NEEDS "BCPLC" directive.

In BCPL, "current input" and "current output" I/O channels are assumed to be referenced by the read and write operations respectively. These operations treat the channels as a continuous stream of characters (bytes).

7.31 FINDINPUT and FINDOUTPUT

These functions are used to open specified files for use as "standard" BCPL I/O channels. The file descriptors which they return may be used as arguments to the SELECTINPUT and SELECTOUTPUT routines.

=FINDINPUT(NAME) opens the specified file for input in the "fast" read mode (see section 7.51), and returns a pointer to a newly created file descriptor (or an error -- see section 7.5) as its result. NAME must be a string containing a legal system file name. This function is equivalent to OPEN(NAME,IO.FREAD).

=FINDOUTPUT(NAME) is similar to FINDINPUT, except that the referenced file is opened in the "fast" write mode. This function is equivalent to OPEN(NAME,IO.FWRITE).

7.32 SELECTINPUT and SELECTOUTPUT

These routines are used to establish the "current" I/O channels for use by the read and write operations. These channels default to the console at initialization time and thenceforth are updated by the user as desired.

In each of these routines, the single argument is either a file descriptor pointer (such as returned by FINDINPUT, etc.) or either of the special descriptors CONSOLE or DUMMY, which are defined in LIBHDR.

SELECTINPUT(FILDES) specifies the current input channel, which is used for all subsequent standard BCPL read operations until the channel is changed by a SELECTINPUT, ENDREAD, or ENDTOINPUT command. Input from the DUMMY file returns ENDSTREAMCH (defined in LIBHDR).

SELECTOUTPUT(FILDES) specifies the current output channel, which is used for all subsequent standard BCPL write operations until the channel is changed by a SELECTOUTPUT, ENDWRITE, or ENDTOINPUT command. Output to the DUMMY file is ignored.

7.33 CH, INCHAN, and OUTCHAN

These are global variables which are used by the BCPL character (stream) I/O routines. They may be referenced by the user, but should not be modified by same (except indirectly, through the use of routines such as RDCH, SELECTINPUT, etc.).

CH is a global variable containing the character most recently input by the RDCH function. It should be noted that READN and READNUMBER (see section 7.46) also modify CH, as they employ RDCH in their operations. CH is maintained for each separate I/O channel, and is properly preserved by SELECTINPUT.

INCHAN is a global variable containing the file descriptor of the "current" input channel. This is maintained primarily for compatibility with other BCPL implementations; its value should only be accessed using INPUT.

OUTCHAN is a global variable containing the file descriptor of the "current" output channel. Like INCHAN, this should normally not be accessed directly, but rather using OUTPUT.

7.34 INPUT and OUTPUT

These functions take no arguments, and are used to access the current file descriptors for the input and output channels, thereby allowing them to be saved and (possibly) restored at a later time.

=INPUT() returns the latest file descriptor passed by SELECTINPUT.

=OUTPUT() returns the latest file descriptor passed by SELECTOUTPUT.

7.35 REWIND

=REWIND() is a function taking no arguments which attempts to (re)position the "current" input channel to the start of the file. It returns any error encountered (see section 7.5).

7.36 ENDREAD, ENDWRITE, and ENDTOINPUT

These functions operate on the currently selected input and output channels as necessary and return any errors encountered (see section 7.5). Output channels are flushed (see section 7.54) prior to closing.

=ENDREAD() will close the current input channel and "unset" its selection.

=ENDWRITE() will close the current output channel and "unset" its selection.

=ENDTOINPUT() will close the current output channel and "unset" its selection. It then reopens the file and selects same as the current input channel.

7.37 RDCH and WRCH

These are the basic single-character read and write operations of BCPL. They operate on the currently selected I/O channels.

=RDCH() reads one character from the "current" input channel. The character is masked with #177 to remove the parity bit and the result copied into CH (see section 7.33) as well as returned in the normal fashion. Carriage return is converted to the BCPL newline character ('*N'); CTRL/Z (ASCII value #32) is converted to ENDSTREAMCH; nulls and line feeds are ignored. These special interpretations may be avoided if necessary by using GETB (see section 7.53).

WRCH(CHAR) writes its single character argument to the "current" output channel. Newline is converted to carriage return followed by line feed; horizontal tabs are simulated using the "space" character. These special interpretations may be avoided if necessary by using PUTB (see section 7.53).

7.38 UNRDCH

UNRDCH() is a routine to "backspace" one character on the currently selected input channel. Its effect is such that the next call to RDCH will return the character currently in CH. The value of CH following the UNRDCH operation, as well as the effect of multiple UNRDCH's is not defined in standard BCPL (in the Finnins implementation, CH is unchanged, and each UNRDCH will "put back" one more copy of same). For a more flexible "un-read" mechanism, see PUTBACK (section 7.54).

7.39 NEWLINE and NEWPAGE

These routines are used for convenience in writins common "termination" characters to the currently selected output channel. They have the additional effect of flushins the channel buffer (see section 7.54).

NEWLINE() writes the newline character ('*N') to the "current" output channel.

NEWPAGE() writes the newpage character ('*P') to the "current" output channel.

7.4 Formatted (stream) I/O routines

As in section 7.3, these routines are considered to be "standard" BCPL I/O operations, and operate on the currently selected I/O channels. They are loaded from the library with

a NEEDS "BCPLC" directive.

7.41 WRITED and WRITEN

These routines are used to output a value as a signed decimal number.

WRITED(VALUE,WIDTH) will write the decimal value to an output field of the specified width, filling in leading spaces if required. Note that the value will always be printed correctly, even if the specified field width is too small (in this case it will overflow into the minimum necessary width). A field width of 6 will always be large enough to ensure the correct alignment of fields on the Nova.

WRITEN(VALUE) is an implementation-independent routine which writes a signed decimal number in the minimum necessary field width. It is equivalent to WRITED(VALUE,0).

7.42 WRITEOCT and WRITEO

These routines are used to output a value as an unsigned octal number.

WRITEOCT(VALUE,WIDTH) will write the octal value to an output field of the specified width (always), truncating or filling with leading zeroes as required.

WRITEO(VALUE) is an implementation-independent routine which will write an octal number in the minimum necessary field width for the machine in use. It is equivalent to WRITEOCT(VALUE,6) on the Nova.

7.43 WRITEHEX and WRITEH

These routines are used to output a value as an unsigned hexadecimal number.

WRITEHEX(VALUE,WIDTH) will write the hexadecimal value to an output field of the specified width (always), truncating or filling with leading zeroes as required.

WRITEH(VALUE) is an implementation-independent routine which will write a hexadecimal number in the minimum necessary field width for the machine in use. It is equivalent to WRITEHEX(VALUE,4) on the Nova.

7.44 WRITES

WRITES(STRING) takes a standard (packed) format BCPL string as its single argument and copies it one character at a time to the "current" output channel (using WRCH).

7.45 WRITEF

WRITEF(STRING,VALUE1,VALUE2,...) takes a standard (packed) format BCPL string as its first argument and copies it one character at a time to the "current" output channel. Within this "format string", occurrences of the sequence %- cause output of the values contained (sequentially) in the second and subsequent arguments. The following output formats are recognized, output being performed by the indicated routines:

%N	numeric value	WRITEN
%Iw	numeric value	WRITED
%Ow	numeric value	WRITEOCT
%Hw	numeric value	WRITEHEX
%S	string value	WRITES
%C	character value	WRCH

"w" in this list refers to a field width specification, which is any legal hexadecimal digit. A maximum of twelve arguments is provided for (the format string plus eleven inserted values). Some examples of the use of WRITEF:

```
        WRITEF("%N + %N = %N",12,34,12+34)
writes   "12 + 34 = 46"

        WRITEF("%O2 OCTAL =%I2 DECIMAL.",9,9)
writes   "11 OCTAL = 9 DECIMAL."

        WRITEF("%C,%C,%S",65,'B',"C,D")
writes   "A,B,C,D"
```

Note that %Iw calls WRITED and can overflow its field in the same way (see section 7.41).

7.46 READNUMBER and READN

These functions are used to read numeric values from the currently selected input channel.

=READNUMBER(RADIX) returns the value of a signed numeric character stream from the "current" input channel, using the specified radix (2-16) for conversion. Leading space, tab, newline, and newpage characters are ignored; "+" and "-" cause the appropriate action, and digits are read until a non-digit character is encountered. If no digits are present, the result is zero; overflow is not detected.

=READN() returns the value of a signed decimal character stream from the "current" input channel. It is equivalent to READNUMBER(10).

Both of the above functions leave the terminating character in CH (as a result of the calls to RDCH). In many versions of BCPL this terminator is left in a global variable called

TERMINATOR. To achieve compatibility with this type of implementation, it is merely necessary to declare a global as follows:

```
GLOBAL $( TERMINATOR : 70 $)
```

where 70 is the global number of CH.

Note also that these functions begin reading their first input character from the currently selected input channel. If the first digit of the number has already been read by a program (which thus determines that it is a number), it is necessary to "put it back" to the channel (probably using UNRDCH) prior to calling the numeric input function.

7.5 RDOS I/O routines

The routines and functions in this section provide a comprehensive collection of I/O facilities which are compatible with the RDOS/RTOS/etc. philosophy. Some minor modifications to the standard RDOS "rules" have been made in the interests of BCPL compatibility, but the user should recognize that these operations are non-standard BCPL, and as such are not easily transportable to other implementations.

Most of these functions return optional error codes; when provided, these errors are the boolean complement of the RDOS error number, and therefore will test as a negative value. If they are passed directly as an argument to the STOP routine, a proper error return will be made to the operating system (see section 7.22). Functions which only return error values return zero when no error condition exists.

It should be noted that inasmuch as these operations interact directly with the operating system, they form the basis of all I/O in this BCPL implementation (i.e., the "standard" BCPL operations are defined in terms of these). If it is desired to use these routines separately, they may be called from the library with a NEEDS "BCPLI" directive. In the descriptions which follow, familiarity with the RDOS I/O environment is assumed, as this will be necessary to make use of these facilities at any rate.

7.51 OPEN and CLOSE

These functions are used to "open" and "close" operating system files in order to allow access to same with the read and write commands.

=OPEN(FILENAME,MODE) is a function taking two arguments, a (standard packed format) BCPL string naming the file to be opened, and a "mode" indicator (described below). The file is opened as specified, a "file descriptor" vector is claimed

from free storage and initialized, and a pointer to this vector (or an error code) is returned as the function result. This pointer will always test as a positive value, and is therefore easily differentiated from an error indicator. A file may be opened in one of five "modes", and this determines which operations may subsequently be performed. The codes for these OPEN modes are defined in the standard library header; they function as follows:

IO.READ opens a file for reading using RDOS-type commands (BYTEREAD, LINEREAD, etc.). The .ROPEN function is used.

IO.READWRITE opens a file for reading and/or writing using RDOS-type commands (BLOCKREAD, BYTEWRITE, etc.). If the file does not exist, a new one is automatically created using .CRAND. The .OPEN function is used.

IO.WRITE opens a file for writing using RDOS-type commands (BYTEWRITE, LINEWRITE, etc.). If the file does not exist, a new one is automatically created using .CRAND. The .APPEND function is used.

IO.FREAD and IO.FWRITE are identical to IO.READ and IO.WRITE respectively, but the files are opened for I/O using single-byte-oriented commands (GETB, PUTC, PUTBACK, etc.). This is implemented by building and using an internal buffer for these channels so as not to suffer from the inefficiencies of RDOS single-byte I/O. Files opened in this manner are referred to as "fast" files in the Finning BCPL documentation.

Note that an attempt to perform I/O to a file whose opening mode is incompatible with the operation type will result in a BCPL run-time error (see Appendix D).

=CLOSE(FILDES) will close the file referenced by its file descriptor argument, returning any error as a result, and restoring the file descriptor vector to free storage. Any file which has been opened in the "fast write" mode will be automatically flushed (see section 7.54).

7.52 DELETE and RENAME

=DELETE(FILENAME) takes as its single argument a (standard packed format) BCPL string naming a file to be deleted. It returns any error encountered as the function result.

=RENAME(OLDNAME,NEWNAME) takes two arguments, each a (standard packed format) BCPL string. The file named by the first argument is renamed as specified by the second argument. Any error encountered is returned as the function result.

7.53 GETB, PUTB, GETC, PUTC, CONSOLEIN, and CONSOLEOUT

These operations all perform single-byte I/O on "fast" files (i.e., files which have been so opened; see section 7.51). They do not return error codes to the user, but may generate run-time error messages.

=GETB(FILDES) returns the next (8-bit) byte read from the file indicated by the (file descriptor pointer) argument. ENDSTREAMCH is returned on end-of-file or reading from the DUMMY channel.

PUTB(FILDES,BYTE) writes the specified byte (the rightmost eight bits of the second argument) to the file indicated by the (file descriptor pointer) first argument. Output to the DUMMY channel is ignored.

=GETC(FILDES) is identical to GETB except that the returned value is treated as an ASCII character (see RDCH in section 7.37).

PUTC(FILDES,CHARACTER) is identical to PUTB except that the supplied byte is treated as an ASCII character (see WRCH in section 7.37).

=CONSOLEIN() provides for direct input from the console (using .GCHAR) without requiring a file OPEN. It "echoes" the received byte and returns it as an ASCII character (see RDCH in section 7.37).

CONSOLEOUT(CHARACTER) provides for direct output to the console (using .PCHAR) without requiring a file OPEN. It prints the specified byte as an ASCII character (see WRCH in section 7.37).

7.54 PUTBACK and FLUSH

PUTBACK(FILDES,CHARACTER) may only reference a file which has been opened in the "fast read" mode (see section 7.51). Its function is to (re)place a character (specified in its second argument) "in front of" the current position in the file indicated by the (file descriptor pointer) first argument. This character is not actually written to the file, but rather retained in a last-in, first-out (LIFO) list, which must be emptied by subsequent read operations before the actual file data is again read. DUMMY channel references are ignored.

FLUSH(FILDES) may only reference a file which has been opened in the "fast write" mode (see section 7.51). Its function is to "flush" the internal buffer (see IO.FWRITE in section 7.51) of any accumulated data. This function is performed automatically whenever the buffer is filled, or in response to such operations as CLOSE, ENDWRITE, NEWLINE, and NEWPAGE, but the user may have some other reason for wishing it to

occur (such as the forced updating of a data file, etc.). DUMMY channel references are ignored.

7.55 BYTEREAD and BYTEWRITE

These functions are similar to the "read/write sequential" operations of RDOS. They each require specification of an internal byte address, which is normally accomplished by shifting a word address one bit left (<<1). CONSOLE and DUMMY channel references are illegal.

=BYTEREAD(FILDES,ADDRESS,NUMBER) will attempt to read the indicated number of bytes from the referenced file into memory, beginning at the specified byte address. It returns the actual count of bytes read (or any error code) as the function result, which may be less than the number requested if an end-of-file is encountered. Note that this differs from the RDOS .RDS in that end-of-file is not treated as an error condition until the file is truly exhausted.

=BYTEWRITE(FILDES,ADDRESS,NUMBER) will attempt to write the indicated number of bytes to the referenced file, beginning at the specified byte address. Any error encountered is returned as a function result.

7.56 BLOCKREAD and BLOCKWRITE

These functions are similar to the "read/write block" operations of RDOS, performing I/O in blocks of 256 words. Unlike the .RDB and .WRB error sequences, any returned error codes do not contain the partial block count. CONSOLE and DUMMY channel references are illegal.

=BLOCKREAD(FILDES,ADDRESS,BLOCK,NUMBER) will attempt to read the indicated number of blocks from the specified file (starting with the desired relative block) into memory, beginning at the specified word address. Any error encountered is returned as a function result.

=BLOCKWRITE(FILDES,ADDRESS,BLOCK,NUMBER) will attempt to write the indicated number of blocks to the specified file (starting with the desired relative block) from memory, beginning at the specified word address. Any error encountered is returned as a function result.

7.57 GETPOSITION and SETPOSITION

These functions are similar to the "GPOS/SPOS" operations of RDOS. They each make use of a two-word "position vector", the contents of which describe a 32-bit relative byte address within the specified file (in this vector, the first word is of higher significance). CONSOLE and DUMMY channel references are illegal.

=GETPOSITION(FILDES,POSITION) will copy the current position of the specified file into the supplied position vector. Any error encountered is returned as a function result.

=SETPOSITION(FILDES,POSITION) will attempt to set the current position of the specified file to that indicated by the supplied position vector. Any error encountered is returned as a function result.

7.58 LINEREAD and LINEWRITE

These functions are similar to the "read/write line" operations of RDOS. They each require specification of a vector which is constructed as a standard (packed) format BCPL string. CONSOLE and DUMMY channel references are illegal.

=LINEREAD(FILDES,STRING) will input a "line" (using the standard RDOS conventions) from the specified file into the supplied vector. In the string which results, the line termination character (which may be a null) is included in the character count. An "invisible" (uncounted) null is always appended to the line, in keeping with the standard string format (see section 5.32). Note that a 135-byte vector (VEC 67) will always be sufficient to contain an input line. Any error encountered is returned as a function result.

=LINEWRITE(FILDES,STRING) will output a "line" (using the standard RDOS conventions) from the supplied string to the specified file. It is the responsibility of the user to ensure that a line termination character is supplied at an appropriate point (if the standard BCPL packed string format is used, a terminating null byte will always exist). Any error encountered is returned as a function result.

7.59 CHANGEPHASE

=CHANGEPHASE(FILENAME,OPTION,AC2) provides access to the .EXEC facility of RDOS. It takes three arguments: a filename in standard (packed) BCPL string format, an option value describing the operation type (mnemonics for these are defined in the standard library header), and a value to be passed in AC2. Any value returned by the called program will be returned as a function result.

7.6 Multitasking routines

The routines and functions in this section provide convenient base level support for multitasking operations which are compatible with the RDOS/RTOS/etc. philosophy. All of these operations are non-standard BCPL, and as such are not easily

transportable to other implementations. Function error results are treated in identical fashion to RDOS I/O errors (see section 7.5). A NEEDS "BCPLM" directive is necessary to load these routines from the library. In the descriptions which follow, familiarity with the RDOS multitasking philosophy is assumed.

7.61 TASK

TASK(ROUTINE,STACKSIZE,I.D.,PRIORITY,ARGUMENT) is the basic multitasking operation, creating a new execution path beginning with the specified routine, which is called with a single argument (the fifth argument in the TASK call). The new task takes the given I.D. and priority level, and executes with the specified maximum stack size (obtained from free storage). Values in the range 0-255 are permitted for the third and fourth arguments: an I.D. of zero specifies that no I.D. number is to be assigned to this task; a priority level of zero specifies that the level of the issuing task is to be used. The new task execution path is terminated by returning from the initially called routine, or executing a FINISH command, at which time the stack space used by the task is returned to free storage. Errors encountered in the TASK operation are considered fatal.

7.62 XMIT, XMITWAIT, and RECEIVE

These functions are used to pass one-word messages, using agreed-upon locations in memory. They may be used for communication between tasks, or as a process interlock facility.

=XMIT(LOCATION,MESSAGE) deposits the specified non-zero "message" in the given location, which must initially contain zero. Any error encountered is returned as a function result.

=XMITWAIT(LOCATION,MESSAGE) is identical to XMIT, except that the issuing task is suspended from execution until the message has been RECEIVEd by another task.

=RECEIVE(LOCATION) suspends the issuing task until the specified location contains a non-zero message. When this occurs (which may be immediately), the message location is reset to zero, the task is readied, and the received message is returned as the function result. No errors are returned by this function.

7.63 DELAY

DELAY(COUNT) is a routine which suspends the issuing task for a period of time equal to the specified number of system clock "ticks".

7.64 PRIORITY

PRIORITY(LEVEL) is a routine which redefines the execution priority of the issuing task to be that value specified by its single argument (range 0-255).

7.65 SUSPEND and READY

These routines are used to provide for arbitrary suspension and (subsequent) readying of tasks, based on their assigned I.D. numbers. Specifying an undefined I.D. number will result in a fatal run-time error.

SUSPEND(I.D.) is a routine which will suspend execution of the task with the specified I.D. number until subsequent execution of a READY operation. If a zero argument is used, the issuing task will be suspended.

READY(I.D.) is a routine which will ready a task with the specified I.D. number which has been previously SUSPENDed. If the task is not suspended, this routine has no effect.

7.7 Timing routines

The operations in this section provide access to timing information maintained by the operating system. These routines are loaded from the library with a NEEDS "BCPLT" directive.

7.71 DATE and TIME

Each of these routines returns information in a (minimum) three word vector supplied by the calling program. In order that they return valid results, it is necessary for the date and time values to have been properly initialized by the system at start-up time.

DATE(VECTOR) causes the current date to be copied into the indicated vector. The order of storage is day (1-31), month (1-12), and year (e.g., 1977).

TIME(VECTOR) causes the current time-of-day to be copied into the indicated vector. The order of storage is seconds (0-59), minutes (0-59), and hours (0-23).

7.72 ELAPSEDTIME

=ELAPSEDTIME() is a function with no arguments which returns as its result the number of seconds which have elapsed since the start of the current BCPL program. Note that integer

overflow is not checked, so an elapsed time greater than 32,767 seconds (9.1 hours) will be returned incorrectly.

8 Debussins Facilities

... this chapter is not yet available.

9 Special Facilities

9.1 Overlay routine

The BCPL overlay routine allows for a convenient interface with the operating system overlay facilities. It is part of the standard run-time library (does not require a NEEDS directive), and currently supports one overlay mode.

9.11 Preparing the overlay sections

The simplest overlay is that held in one source file and compiled as a single BCPL section. This is compiled with the global S and O switches, the former to indicate that it is a section (not the main program) and the latter to include overlay initialization code at the end of the segment.

If several sections are to be loaded into a single overlay, only the final section should be compiled with the global O switch. E.g.,

RLDR ... [A B C,D,E F,G] ...

Sections C, D, F, and G are the only ones to be compiled with the O switch. All sections, of course, require the S switch.

The main (resident) program which will use the overlays must also be compiled with the global O switch. In this case, the code generator will recognize that it is the main program and not an overlay, and will generate special code for the overlay load operation.

9.12 Loading the overlays

Overlay components prepared as described in section 9.11 form standard operating system overlays, and may be loaded in the normal fashion using RLDR. This is discussed in section 4.72.

9.13 Using the overlays

The overlay file is opened automatically when the program is initialized (on channel 0, but the user does not normally need to know this). Overlays are brought into memory by use of the OVERLAY routine.

OVERLAY(SEGMENT.NUMBER) is a routine to bring an overlay segment into memory. Segments are numbered sequentially, beginning with zero and increasing by one each time a comma appears in the load command sequence. Overlays are brought

in as requested, but are (re)initialized unconditionally. That is, every call to OVERLAY initializes all global routine entries defined in the specified overlay segment. It is therefore possible for overlays to share globals that contain constant routines; this, however, should be done only with extreme caution.

Some restrictions exist on overlay code apart from those naturally imposed by the structure (e.g., no OVERLAY calls from within an overlay -- see the operating system manual). Overlay sections must not contain NEEDS directives. Also, unpredictable faults occur if global routines are called when the overlay in which they are defined is not present in memory. Such faults are extremely difficult to detect, and users should exercise appropriate caution when allocating routines to overlays.

9.2 System call function

A special function is provided to enable the user to communicate directly with the operating system without having to resort to assembly language programming. This mechanism is, of course, highly implementation-dependent, and should be used sparingly. It is part of the standard run-time library and does not require a NEEDS directive.

=SYS(COMMAND,AC0,AC1,AC2,VECTOR) performs the operating system function specified by the COMMAND argument, using initial values for accumulators 0-2 as specified by the arguments AC0, AC1, and AC2. The result vector, which must be a minimum of three words in length (VEC 2), will be set to the three accumulator values following the operation. A function result of TRUE is returned if an error exit was taken by the operating system, FALSE otherwise.

Note that the necessary accumulator values are passed exactly as specified, and it is the responsibility of the user to generate byte addresses, bit values, or whatever is required.

9.3 Argument input functions

A pair of functions is provided to allow the user convenient access to the console argument input mechanisms of the operating system. These functions, if required, are loaded from the library with a NEEDS "BCPLA" directive.

=UNIQUENAME(STRING) edits a character string (typically a filename) according to whether the executing program is in the "foreground" or "background" environment. Within the STRING argument, all occurrences of the 3-character group "%<Bchar><Fchar>" are replaced by the single character

<Bchar> if the program is in the background, <Fchar> otherwise. Finally, all space characters are removed and the address of the resultant string (the same address as the original string, which has been permanently modified) is returned as the function result. Thus the sequence

```
COM.FILE := OPEN(UNIQUENAME("% FCLI.CM"),IO.READ)
```

would open the appropriate CLI command file.

=COMARG(FILDES,NAMEVEC,SWITCHVEC) is a function to input and parse the next argument from the CLI command file. FILDES is the appropriate file descriptor; NAMEVEC is a vector of sufficient size to accept the argument name as a standard (packed) BCPL string; SWITCHVEC is a vector of length 32 (VEC 31) to record the bit settings of the two words of switch information (TRUE = set, FALSE otherwise). SWITCHVEC!0 corresponds to local/global switch A, SWITCHVEC!1 to switch B, and so on. The number of local or global switch settings for the current argument (i.e., a count of the number of TRUE entries in the SWITCHVEC vector) is returned as the function result. If no arguments remain in the CLI command file, the special value ENDSTREAMCH (defined in LIBHDR.BC) is returned as the function result.

9.4 Network I/O function

A special function is provided to enable the user to communicate using the Finning network protocol ("NETCOM"). An understanding of this protocol is assumed for the following description.

This mechanism requires the use of modules in the NETCOM library which correspond to the particular hardware configuration in use. These modules, being non-BCPL in origin, must be loaded following BCPLIB.LB in the load command sequence. The BCPL NETCOM function is loaded with either a NEEDS "BCPLN" directive (which disallows phantom receive operations) or a NEEDS "BCPLP" directive (which links the phantom receive commands into the BCPL GETVEC facility -- dynamically assigned buffers may therefore subsequently be returned using the PUTVEC operation). If both directives are encountered during the relocatable load operation, "BCPLP" will take precedence.

=NETCOM(OPCODE+LENGTH,STATUS.VECTOR,BUFFER) causes one of the standard NETCOM operations to be performed. The operation codes are defined (in LIBHDR.BC) as manifest constants in such a fashion that the number of bytes in the transmission need merely be added to the appropriate op-code. BUFFER is the word address of the communication buffer (supplied by the user in all cases except for "phantom receives"). STATUS.VECTOR is a four word (minimum) vector which contains the initiating process number (!0), the responding process

number (!1), the actual messase lensth followins the transfer (!2), and the address of the communication buffer specified or dynamically assisned (!3). Any error code supplied by NETCOM is returned as the function result. A special case:

```
PORT := NETCOM(-1)
```

is provided to allow determination of the current MCA port number.

9.5 Extended library functions

The routines and functions listed in this section are very special-purpose in nature, and are not defined in the standard library header (LIBHDR.BC). They are instead described in an additional ("extended") library header (XLIBHDR.BC) which is listed in Appendix D. Persons wishins to use any of these operations may either reference the entire extended library with a GET "XLIBHDR" directive, or merely reproduce the desired function name(s) in a GLOBAL declaration within the user prosram. Note that the same slobal number must be used if this second alternative is emplosed.

9.51 Strins manipulation

These functions are used to manipulate strinss in standard (packed) BCPL format (see section 5.32). The "invisible null" termination convention is maintained throughout. A NEEDS "STRFUNC" directive is necessary to load these functions from the library.

=COPYSTR(STRING1,STRING2) copies the contents of STRING1 to STRING2, which must be a vector larse enoush to contain same. The strins lensth is returned as a function result.

=SEARCHSTR(STRING1,STRING2,INDEX) searches STRING2 for the first occurrence (if any) of STRING1, besinnins the search at the specified index. The index of the found substrins (or zero if there was no match) is returned as a function result.

=INSERTCHAR(CHARACTER,STRING,INDEX) inserts the specified character in STRING at the specified index. The strins is extended with space characters if necessary. If INDEX is zero, the character is appended to the strins. The index of the inserted character is returned as a function result.

=INSERTSTR(STRING1,STRING2,INDEX) inserts STRING1 in STRING2 at the specified index. STRING2 is extended with space characters if necessary. If INDEX is zero, STRING1 is appended to STRING2. The index of the last inserted character is returned as a function result.

=EXTRACTSTR(STRING1,STRING2,INDEX,LENGTH) extracts a substring from STRING2, beginning at the specified index and continuing to the end of the string, or LENGTH characters, whichever is less. This substring is copied to STRING1, which must be a vector large enough to contain same, and the length of STRING2 is adjusted to reflect the extracted characters. The length of STRING1 is returned as a function result.

=SCANSTR(STRING1,STRING2,STRING3) searches STRING3 from the beginning for an occurrence of STRING2. If found, all characters to the left of the match are moved to STRING1; all characters to the right of the match (only) remain in STRING3. If not found, all characters from STRING3 are moved to STRING1, leaving STRING3 empty. The function result is TRUE if a match is found, FALSE otherwise.

9.52 Time routine

WRITIME(STRING) will write the time and date to the currently selected output channel in the format "1.29 P.M. ON 27 JULY 1977", followed by STRING. A NEEDS "TIMFUNC" directive is necessary to load this routine from the library.

9.53 Double precision arithmetic

These operations are used to provide a double precision integer arithmetic capability in the Finning BCPL system (allowing values in the range +/- 2,147,483,647). A NEEDS "DBLFUNC" directive is necessary to load these routines from the library.

The most negative number (i.e., a one bit followed by 31 zero bits, which is -2,147,483,648) is reserved to have a meaning of "undefined". All arithmetic functions check for overflow, and set the result to undefined if it occurs. Moreover, if any argument is undefined, the result is undefined. Further operations, therefore, will propagate the undefined value rather than generate meaningless results.

The double precision variables upon which these functions operate are best created as two-word vectors (VEC 1). The following arithmetic operations are provided:

```
=D.MOV(A,B)    --   B := A
=D.ADD(A,B)    --   A := A + B
=D.SUB(A,B)    --   A := A - B
=D.NEG(A)      --   A := -A
=D.ABS(A)      --   A := ABS A
=D.MUL(A,B)    --   A := A * B
=D.DIV(A,B)    --   A := A / B
=D.REM(A,B)    --   A := A REM B
```

The above functions return a result of TRUE if the arithmetic

'result' was undefined, FALSE otherwise.

In addition, two I/O operations are provided which operate on the currently selected channels.

=D.READN(A) is the double precision equivalent of READN (see section 7.46), except that a pointer to the two-word result vector must be provided as an argument. This pointer is returned as a function result.

=D.WRITED(A,WIDTH) is the double precision equivalent of WRITED (see section 7.41).

9.6 The "Q" library

... this section is not yet available.

Appendix A

Basic symbols and synonyms

The following list of words and symbols are treated as atoms by the syntax analyzer. The name of the symbol or its standard representation on the Nova is given in the first column, and examples or synonyms are given in the second.

Basic symbol	Examples and synonyms
identifier	A H1 FORST TAX.RATE
number	126 7249 #3771
string constant	"A" "*NTEST"
character constant	'X' ')' '*N' '"'
TRUE	
FALSE	
?	
(	
)	
:=	
@	LV
!	RV
*	
/	
REM	
+	
-	
ABS	
=	EQ
^=	NE
<=	LE
>=	GE
<	LS
>	GR
<<	LSHIFT
>>	RSHIFT
^	NOT
&	/\ LOGAND
\	\/ LOGOR
EQV	
NEQV	
->	
.	
TABLE	
VALOF	
;	
<>	
:	
$(	$(AB $(1
$)	$)AB $)1
VEC	

```
BE
LET
AND
BREAK
LOOP
ENDCASE
RETURN
FINISH
GOTO
RESULTIS
SWITCHON
INTO
REPEAT
REPEATUNTIL
REPEATWHILE
DO                      THEN
UNTIL
WHILE
FOR
TO
BY
TEST
OR                      ELSE
IF
UNLESS
CASE
DEFAULT
SLCT
OF                      ::
```

Appendix B

ASCII character codes

The following table contains a list of all the ASCII characters recognized by the BCPL compiler.

Decimal	7-bit Octal	Character
7	007	BEL (bell)
8	010	BS (backspace)
9	011	HT (horizontal tab)
10	012	NL (newline)
11	013	VT (vertical tab)
12	014	FF (form feed)
13	015	CR (carriage return)
32	040	SP (space)
33	041	!
34	042	"
35	043	#
36	044	$
37	045	%
38	046	&
39	047	'
40	050	(
41	051	)
42	052	*
43	053	+
44	054	,
45	055	-
46	056	.
47	057	/
48	060	0
49	061	1
50	062	2
51	063	3
52	064	4
53	065	5
54	066	6
55	067	7
56	070	8
57	071	9
58	072	:
59	073	;
60	074	<
61	075	=
62	076	>
63	077	?
64	100	@
65	101	A
66	102	B
67	103	C

68	104	D
69	105	E
70	106	F
71	107	G
72	110	H
73	111	I
74	112	J
75	113	K
76	114	L
77	115	M
78	116	N
79	117	O
80	120	P
81	121	Q
82	122	R
83	123	S
84	124	T
85	125	U
86	126	V
87	127	W
88	130	X
89	131	Y
90	132	Z
91	133	[
92	134	\
93	135	]
94	136	^
95	137	(underline)
97	141	a
98	142	b
99	143	c
100	144	d
101	145	e
102	146	f
103	147	g
104	150	h
105	151	i
106	152	j
107	153	k
108	154	l
109	155	m
110	156	n
111	157	o
112	160	p
113	161	q
114	162	r
115	163	s
116	164	t
117	165	u
118	166	v
119	167	w
120	170	x
121	171	y
122	172	z
124	174	| (vertical bar)
126	176	~ (tilde)

Appendix C

Standard library header

This appendix contains a copy of the standard library header from the file "LIBHDR.BC".

```
//////////////////////////////////////////////////////////
//                                                      //
//  Finnins BCPL System:  "Standard" Library Header     //
//                                                      //
//  Programmer:  D. Dyment, Finnins Tractor             //
//                                                      //
//  Revision Date:  24/July/77                          //
//                                                      //
//////////////////////////////////////////////////////////

GLOBAL $(GLOB

    // In the comments describing these global entries,
    // the following abbreviations are used:
    //      *  - the global is initialized automatically
    //      A  - NEEDS "BCPLA" (argument input module)
    //      C  - NEEDS "BCPLC" (character I/O module)
    //      D  - NEEDS "BCPLD" (debugging module)
    //      I  - NEEDS "BCPLI" (basic I/O module)
    //      M  - NEEDS "BCPLM" (multitasking module)
    //      N  - NEEDS "BCPLN" or "BCPLP" (network I/O modules)
    //      T  - NEEDS "BCPLT" (timer module)
    //      Q  - the global requires the "Q" library
    //      fn - the global references a function
    //      rt - the global references a routine
    //      v  - the global is a variable

SYSTEM          : 0   // * fn  allows access to system data
START           : 1   // * rt  entry point to begin processing
STOP            : 2   // * rt  return to operating system
ABORT           : 3   // * rt  deals with errors
PMSET           : 4   // *  v  data for abort/postmortem routines
GETVEC          : 5   // * fn  obtains a vector from free storage
PUTVEC          : 6   // * rt  returns a vector to free storage
SYS             : 7   // * fn  execute operating system call
XIO             : 8   // Q fn  execute processor I/O instruction

LEVEL           :10   // * fn  returns current stack pointer value
LONGJUMP        :11   // * rt  does non-local jump
APTOVEC         :12   // * fn  "APply TO VECtor"
GETBYTE         :13   // * fn  reads a byte from a packed string
PUTBYTE         :14   // * rt  writes a byte to a packed string
PACKSTRING      :15   // * fn  pack vector of char's into string
```

```
UNPACKSTRING    :16  // * rt  unpack strins into vector
ITASK           :17  // Q rt  creates an interrupt task
IENABLE         :18  // Q rt  enables priority interrupt system
IEXIT           :19  // Q rt  exits an interrupt service routine

TASK            :20  // M rt  creates a new task
XMIT            :21  // M fn  transmits an inter-task messase
XMITWAIT        :22  // M fn  transmits a messase & waits
RECEIVE         :23  // M fn  receives an inter-task messase
DELAY           :24  // M rt  delays execution of current task
PRIORITY        :25  // M rt  chanses current task priority
SUSPEND         :26  // M rt  suspends current task execution
READY           :27  // M fn  makes ready a suspended task

OPEN            :30  // I fn  open a file for I/O
CLOSE           :31  // I fn  close an open file
DELETE          :32  // I fn  delete a file
RENAME          :33  // I fn  rename a file
GETB            :34  // I fn  read byte from a "fast" file
PUTB            :35  // I rt  write byte to a "fast" file
GETC            :36  // I fn  read char. from a "fast" file
PUTC            :37  // I rt  write char. to a "fast" file
PUTBACK         :38  // I rt  return char. to a "fast" file
FLUSH           :39  // I rt  flush "fast" file output buffer

BYTEREAD        :40  // I fn  sequential byte read from file
BYTEWRITE       :41  // I fn  sequential byte write to file
BLOCKREAD       :42  // I fn  block read from file
BLOCKWRITE      :43  // I fn  block write to file
CONSOLEIN       :44  // I fn  read character from console
CONSOLEOUT      :45  // I rt  write character to console
GETPOSITION     :46  // I fn  read current file position
SETPOSITION     :47  // I fn  update current file position
LINEREAD        :48  // I fn  read a line from a file
LINEWRITE       :49  // I fn  write a line to a file

MAPSTORE        :50  // D rt  output routine names, entry counts
BACKTRACE       :51  // D rt  unwind stack
PRINTGLOBALS    :52  // D rt  write out slobal vector
PRINTENTRIES    :53  // D rt  write out all routine entries
PRINTPROFILES   :54  // D rt  write profile counts
POSTMORTEM      :55  // D rt  seneral debussins routine
USERDEBUG       :59  // D rt  user-supplied debussins routine

DATE            :60  // T fn  returns date in a vector
TIME            :61  // T fn  returns time of day in a vector
ELAPSEDTIME     :62  // T fn  returns elapsed time since start
COMARG          :65  // A fn  read an argument from (F)COM.CM
UNIQUENAME      :66  // A fn  constructs "sround"-related names
CHANGEPHASE     :67  // I fn  load and enter named program
OVERLAY         :68  // * fn  load overlay, resettins used slobals
NETCOM          :69  // N fn  performs network I/O


CH              :70  // G  v  last character read from CIF
INCHAN          :71  // G  v  current input file (CIF) descriptor
```

```
OUTCHAN          :72  // C  v  current output file (COF) descriptor
SELECTINPUT      :73  // C rt  selects input file as current
SELECTOUTPUT     :74  // C rt  selects output file as current
RDCH             :75  // C fn  read a character from CIF
UNRDCH           :76  // C rt  causes RDCH to return current CH
WRCH             :77  // C rt  write a character to COF
NEWLINE          :78  // C rt  write a newline to COF
NEWPAGE          :79  // C rt  write a newpage to COF

WRITES           :80  // C rt  write a string to COF
WRITEF           :81  // C rt  write a formatted string to COF
WRITED           :82  // C rt  write decimal number in given width
WRITEOCT         :83  // C rt  write octal number in given width
WRITEHEX         :84  // C rt  write hex number in given width
WRITEN           :85  // C rt  write decimal number to COF
WRITEO           :86  // C rt  write octal number to COF
WRITEH           :87  // C rt  write hexadecimal number to COF
READN            :88  // C fn  read decimal number from CIF
READNUMBER       :89  // C fn  read number in given radix

FINDINPUT        :90  // C fn  open file for "fast" input
FINDOUTPUT       :91  // C fn  open file for "fast" output
REWIND           :92  // C fn  rewinds CIF if possible
ENDREAD          :93  // C fn  end current input; unset selection
ENDWRITE         :94  // C fn  end current output; unset selection
ENDTOINPUT       :95  // C fn  closes COF and reopens for input
INPUT            :96  // C fn  returns CIF descriptor
OUTPUT           :97  // C fn  returns COF descriptor

$)GLOB

MANIFEST $(MAN      // System manifest constants

ENDSTREAMCH    = -1
BYTESPERWORD   = 2
MAXINT         = 32767
LENGTH         = SLCT 8:8
DUMMY          = 0
CONSOLE        = -1


// Postmortem settings decoded by ABORT

PM.PM     = 1
PM.MAP    = 2
PM.GLOB   = 4
PM.ENT    = 8
PM.BACK   = 16
PM.PROF   = 32
PM.USER   = 64
PM.TRAP   = 128
PM.ABORT  = #100000
```

```
// CHANGEPHASE options
\
PHASE.SWAP       = 0
PHASE.SWAPDEB    = 1
PHASE.CHAIN      = #100000
PHASE.CHAINDEB   = #100001


// SYSTEM specifiers

GLOBALZBASE          = 0   // base address of ZREL globals
GLOBALNBASE          = 1   // base address of NREL globals
GLOBALBREAK          = 2   // break point of globals
GLOBALTOP            = 3   // top address of globals
STACKBASE            = 4   // base address of current task stack
STACKTOP             = 5   // top address of current task stack
STACKSPACE           = 6   // space available on current stack
VECTORSPACE          = 7   // space available from GETVEC
PROGBASE             = 8   // base address of program NREL
PROGTOP              = 9   // top address of program
ADDRESSOFGLOBAL      = 10  // address of global #N
ADDRESSOF            = 11  // memory reference address
GROUND               = 12  // user "ground" indicator
SETTABWIDTH          = 13  // output tab width (default = 8)
SETSLOP              = 14  // GETVEC "efficiency" (minimum = 3)


// OPEN modes

IO.READ              = 0   // read-only
IO.FREAD             = 1   // "fast" read (buffered)
IO.WRITE             = 2   // write
IO.FWRITE            = 3   // "fast" write (buffered)
IO.READWRITE         = 4   // read/write


// NETCOM op-codes

NET.ORCV     = #177777     // "open" receive
NET.DRCV     = #003777     // "directed" receive
NET.XMIT     = #007777     // transmit
NET.XCV      = #013777     // transceive
NET.XXMIT    = #017777     // "transparent" transmit
NET.QXMIT    = #023777     // "quick" transmit
NET.TERM     = #027777     // "terminate"
NET.PORCV    = #077777     // "phantom open" receive
NET.PDRCV    = #103777     // "phantom directed" receive

$)MAN
```

Appendix D

Extended library header

This appendix contains a copy of the extended library header from the file "XLIBHDR.BC".

```
////////////////////////////////////////////////////////////
//                                                        //
//  Finnins BCPL System:  "Extended" Library Header       //
//                                                        //
//  Prosrammer:  D. Dyment, Finnins Tractor               //
//                                                        //
//  Revision Date:  27/July/77                            //
//                                                        //
////////////////////////////////////////////////////////////


GLOBAL $(XGLOB

    // In the comments describins these slobal entries,
    // the followins abbreviations are used:
    //      S  - NEEDS "STRFUNC" (strins functions)
    //      T  - NEEDS "TIMFUNC" (time functions)
    //      D  - NEEDS "DBLFUNC" (double precision functions)
    //      fn - the slobal references a function
    //      rt - the slobal references a routine

COPYSTR         :100  // S fn  copies a strins
SEARCHSTR       :101  // S fn  searches strins for substrins
INSERTCHAR      :102  // S fn  inserts character in strins
INSERTSTR       :103  // S fn  inserts strins in strins
EXTRACTSTR      :104  // S fn  extracts substrins from strins
SCANSTR         :105  // S fn  scans and "splits" a strins

WRITIME         :109  // T rt  prints formatted time and date

D.MOV           :110  // D fn  double precision move
D.ADD           :111  // D fn  double precision add
D.SUB           :112  // D fn  double precision subtract
D.NEG           :113  // D fn  double precision nesate
D.ABS           :114  // D fn  double precision absolute value
D.MUL           :115  // D fn  double precision multiply
D.DIV           :116  // D fn  double precision divide
D.REM           :117  // D fn  double precision remainder
D.READN         :118  // D fn  double precision READN
D.WRITED        :119  // D rt  double precision WRITED

$)XGLOB
```

Appendix E

BCPL run-time error messages

The run-time system will detect various errors under conditions as described in this manual. There are 8 such errors, and they all act as "BCPL run-time" error returns (see section 7.22). As such, they will be passed as arguments to ABORT, STOP, POSTMORTEM, or USERDEBUG.

The association between error numbers and their meanins is as follows:

Error #1	Stack overflow
Error #2	Unassigned global or local variable
Error #3	Unimplemented facility
Error #4	Library error
Error #5	Division overflow
Error #6	Insufficient space for vector (or stack)
Error #7	Debug package not loaded
Error #10	File use incompatible with "OPEN" mode

When these errors are detected, the system returns to the previous execution level with return (error) code #10000+n.

Appendix F

... this appendix is not yet available.