B C P L

Reference Manual

James E. Curry
and PARC staff

Compiled on: September 14, 1979

Computer Sciences Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

TABLE OF CONTENTS

SECTION 1
INTRODUCTION

BCPL is a general purpose recursive programming language which is particularly suitable for systems programming applications. Versions of BCPL exist on various computer systems, including CTSS at Project MAC, the GE635 under GE COS, the TX-2 at Lincoln Lab, and the PDP-11, as well as for the Nova. The Nova version of BCPL was bootstrapped from the TX-2 implementation, and incorporates most of the features introduced into BCPL at Lincoln, including a version of structures.

This manual uses an informal syntactic notation. Ellipsis ("...") indicates repetition. Lower-case words are reserved words. Upper-case words represent syntactic classes, the most common of which are:

NAME:	an identifier
EXP:	a BCPL expression
CONST:	an expression involving only constants
REF:	a memory reference expression
STAT:	a BCPL statement or compound statement

SECTION 2
A SAMPLE PROGRAM

2-1 . . . . . The Queens Problem

The following program is a complete, working example of BCPL. It solves the "8-Queens" problem, generating all 8*8 chessboard configurations of eight queens such that no queen can capture any of the others. The central procedure "Queens(Col)" is called with a column number as its argument; it assumes that there are no conflicts in the columns to the left, and tries to place a queen in the current column. "Queens" calls itself recursively to iterate over the columns to the right, or prints a picture of the board if a solution has been found. Three global vectors, "Horiz", "UpDiag", and "DnDiag", are maintained to indicate whether a queen has already been placed in a particular row, upward-diagonal, or downward-diagonal; an attempt to place a queen in an occupied line results in rejection. A solution vector "Row" is maintained for typeout, remembering which row the queen is in for each column.

The program consists of two source files: "QUEENS" and "QUEENS1". The first file contains the main program and some IO procedures; the second contains the "Queens" procedure.

2-2 . . . . . Source Code -- QUEENS

```
// Solution of 8 Queens problem -- Main Program

get "iox"       // Include definitions for IO package

manifest boardsize = 7  // Rows & Columns are numbered 0-7

external
  [     Solutions       // Total number of solutions
        Row             // Row!I  = occupied column in row I
        Horiz           // Horiz!I = true if row I is occupied
        UpDiag          // UpDiag!I= true if up-diagonal I is occupied
        DnDiag          // DnDiag!I= true if down-diagonal I is occupied
  ]
external Queens         // The procedure that does the work
external                // Some extra IO procedures
  [     WriteS
        WriteN
        WriteL
  ]

static
  [     Solutions = 0   // No solutions initially
        Row = nil       // Global vectors -- set up by Main
        Horiz = nil
        UpDiag = nil
        DnDiag = nil
  ]
static TTYstream        // The stream used by WriteS, etc.

let Main() be
  [main
        // Initialize the global vectors
        let v = vec boardsize; Row = v
        let v = vec boardsize; Horiz = v
        for i = 0 to boardsize do Horiz!i = false
        let v = vec boardsize*2; UpDiag = v
        let v = vec boardsize*2; DnDiag = v
        for i = 0 to boardsize*2 do UpDiag!i, DnDiag!i = false, false

        // Initialize output to TTY
        initbcplio()
        TTYstream = open("")

        // Do the work
        Queens(0)

        // Print number of solutions
        WriteN(Solutions)
        WriteS(" solutions found*n")
  ]main

and WriteS(S) be writestr(TTYstream, S)

and WriteN(N) be writedec(TTYstream, N)

and WriteL() be writestr(TTYstream, "*n")
```

2-3 . . . . . Source Code -- QUEENS1

```
// Solution of 8 Queens problem -- Queens procedure

manifest boardsize = 7  // Rows & Columns are numbered 0-7

external
  [     Solutions       // Total number of solutions
        Row             // Row!I  = occupied column in row I
        Horiz           // Horiz!I = true if row I is occupied
        UpDiag          // UpDiag!I= true if up-diagonal I is occupied
        DnDiag          // DnDiag!I= true if down-diagonal I is occupied
  ]
external Queens         // The procedure that does the work
external                // Some extra IO procedures
  [     WriteS
        WriteN
        WriteL
  ]

let Queens(Col) be
  [queens
        // There are no conflicts in columns left of  Col

        let UpDiag2, DnDiag2 = UpDiag+boardsize-Col, DnDiag+Col
        // UpDiag2, Dndiag2 are the diagonal vectors for this column

        for n = 0 to boardsize do
         [rowloop       // Try to put a Queen in each row of this column

                if Horiz!n % UpDiag2!n % DnDiag2!n loop // Can't - go on

                // There are no conflicts to the left, so we can
                Row!Col = n     // Remember for typeout

                test Col eq boardsize   // Done?

                ifnot [ Horiz!n,UpDiag2!n,DnDiag2!n = true,true,true
                        // Now a Queen is in this column
                        Queens(Col+1) // Find all solutions to the right
                        // Now remove the Queen
                        Horiz!n,UpDiag2!n,DnDiag2!n = false,false,false
                      ]

                ifso  [ // Print the solution
                        WriteL()
                        for r = 0 to boardsize do
                         [  for c = 0 to boardsize do
                                WriteS(Row!r eq c ? " Q", " .")
                            WriteL()
                         ]
                        Solutions = Solutions + 1
                      ]

         ]rowloop       // Do the next row
  ]queens
```

2-4 . . . . . Notes on the Source Code

The file "IOX" contains external declarations for a basic IO library; "QUEENS" uses "initbcplio", "open", "writestr", and "writedec" from this library.

The manifest and external declarations appear in both source files. These declarations would usually be put into a separate file; each source file would "get" this file in order to include the declarations.

The static declarations appear only in "QUEENS"; static variables must be declared as static only once, although they may be declared external in many files. "Solutions" is initialized to 0; the statics for the global vectors will be initialized by the main procedure, so they are initialized to "nil". "TTYstream" is declared static but not external, so it is local to "QUEENS", as is "Main".

The main program allocates the vector space for the global vectors by declaring four local vectors (all named "v") and storing the address of the first elements in the external variables for the vectors. This is the simplest way to get space which is global to several procedures (or to a recursive procedure); the space is global to "Queens" since it is allocated by the procedure which calls "Queens".

Note that declarations may be intermixed with statements.

2-5 . . . . . Compiling and Loading QUEENS

To compile the source file QUEENS, just type

```
BCPL QUEENS
```

(Only one source file may be compiled at a time.) The compiler will print

```
BCPL 2.0 -- QUEENS.BR = QUEENS
```

and begin compiling the program. If no errors are detected, the BCPL relocatable binary file QUEENS.BR will be created, and the compiler will print

```
QUEENS.BR --  217 (143) WORDS
```

The numbers are the length of the code generated in octal (decimal). QUEENS1 is compiled similarly.

To load the program, type

```
BLDR/D/L/V QUEENS QUEENS1 IO1 IO2
```

This will create the file QUEENS.SV, an executable Nova save file, from the BCPL relocatable binary files QUEENS.BR, QUEENS1.BR, IO1.BR, and IO2.BR. (The latter two files are the input-output routines.) The /D switch causes the Nova debugger to be loaded into the save file. The /L/V switches create a symbol table file named QUEENS.BS, containing information about where things will be in core when the program runs; a listing of this file is included in the section on Loading (Section 9). The loader prints

```
BLDR 2.0 -- QUEENS.SV, QUEENS.BS
```

at the beginning of the loading process, and when it is done,

```
QUEENS.SV -- 14162  (6256)  WORDS
```

The numbers give the size of the save file in octal (decimal).

To run the program, just type QUEENS. It will print out 92 solutions.

SECTION 3
DECLARATIONS AND PROCEDURES

3-1 . . . . . BCPL Variables

BCPL is a vaguely ALGOL-like language (it is block-structured; it allocates procedure space dynamically, so recursion is permissible; and most BCPL statements correspond roughly to ALGOL statements, although there are syntactic differences). The major difference between BCPL and ALGOL is that all ALGOL variables are declared with data-types (integer, real, boolean, string, array, procedure, label, pointer, etc.), whereas all BCPL variables have the same data-type: a 16-bit number. In ALGOL, the meaning of an expression is dependent both on its context and on the data-types of the entities involved, and only expressions with certain data-types may appear in a given context. In BCPL, any expression may be used in any context; the context alone determines how the 16-bit value of the expression is interpreted. BCPL never checks that a value is "appropriate" for use in a given way. For example, an expression which appears in a "goto" statement is assumed to have as its value the address of someplace which is reasonable to jump to; the thing following a "goto" need not be a label. The advantages of this philosophy about data-types are that it allows the programmer to do almost anything, and that it makes the language conceptually simple. The disadvantages are that the user can make errors which would have been caught by data-type checking, and that some things must be done explicitly which ALGOL-type languages would do automatically (implicit indirection on pointer variables, operations on multi-word values such as real numbers and strings, type conversion, etc.).

Although BCPL has only one data-type, it does distinguish between two kinds of variables: static and dynamic. They differ as to when and where the cells to which they refer are allocated. A static variable refers to a cell which is allocated at the beginning of program execution (i.e., by the BCPL loader); it refers to the same memory cell for as long as the program runs. A dynamic variable refers to a cell which is (conceptually) allocated when the block in which it is defined is entered, and exists only until execution of that block terminates. The space from which the dynamic variable is allocated is created dynamically when the procedure containing its defining block is called.

As in ALGOL, variable names (and other names) are defined in declarations. The lexical scope of a declared name (the portion of the source text in which the name is defined) is governed by BCPL's block structure.

3-2 . . . . . Scope Rules

At the outermost level, a BCPL source file consists of a sequence of global declarations followed by a multiple procedure declaration. The possible global declarations are:

```
external [NAME; ...; NAME]

static [NAME = CONST; ...; NAME = CONST]

manifest [NAME = CONST; ...; NAME = CONST]

structure NAME : [ ... ]
```

The external and static declarations define static variables; the manifest declaration defines literals; the structure declaration defines templates for symbolic references to partial-word and multi-word data.

A multiple procedure declaration has the form

```
let NAME(ARG, ..., ARG) BODY
and NAME(ARG, ..., ARG) BODY
...
and NAME(ARG, ..., ARG) BODY
```

where BODY is either "be STAT" or "= EXP".

The NAMEs in external, static, manifest, and structure declarations at the outermost level are defined from the point of declaration to the end of the source file; all of the NAMEs in the "let ... and ..." sequence at the outermost level are defined in all of the BODYs. These are the only names which are globally defined. All other names are defined either as ARGs in the procedure declarations, or in local declarations within compound statements in the BODYs.

A compound statement is a sequence of statements and declarations, separated by semicolons, and enclosed within the brackets "[" and "]". (If a carriage return separates two statements, the semicolon can be omitted.) The brackets have a function similar to that of the words "begin" and "end" in ALGOL. A compound statement may be used wherever a simple statement can be; in this manual, "STAT" always means either a simple statement or a compound statement. Compound statements are used when two or more statements are needed in a context in which BCPL expects a single statement (e.g., as the body of a procedure, or as one of the arms of a conditional statement). Compound statements delimit the scope of locally declared names.

Local declarations may be intermixed with statements (unlike ALGOL, in which declarations may appear only at the beginning of a compound statement). "Declaration" here includes dynamic variable declarations ("let NAME1, ..., NAMEn = EXP1, ..., EXPn"), as well as the external, static, manifest, structure, and procedure declarations mentioned above. The following rules govern the scope of local declarations:

1) A local declaration may appear in a compound statement only in the following contexts: at the beginning of a statement, or after a semicolon (including a semicolon implicitly inserted by the compiler between statements on different lines), or following a statement label that follows a semicolon. The effect of this rule is to disallow things like "if x eq 0 then let y = 0 (although "if x eq 0 then [let y = 0 ...] is perfectly legal). A declaration may be labeled.

2) A declaration starts a block; the block ends at the end of the compound statement containing the declaration. A name defined in the declaration is known only within the block introduced by the declaration, and in sub-blocks contained within that block if the name is not redeclared.

3) (Exception to rule (2).) A dynamic variable is not known in any procedure body other than the one in which it was declared. Thus, if the procedure "g" is declared inside of the body of procedure "f", the dynamic variables defined in "f" are not known to "g". (This is because the dynamic variables of "f" reside in space which is dynamically allocated when "f" is called. When "g" is called, it does not know where this space is; in fact, there might be more than one execution of "f" in progress when "g" is called, or there might not be any active execution of "f".)

4) A statement label ("NAME: ...") appearing within a block is treated as if it were a static variable declared immediately after the declaration which begins the block. So a label is known throughout its enclosing block, but not outside that block.

3-3 . . . . . Manifest Constants

The declaration

manifest [NAME1 = CONST1; ...; NAMEn = CONSTn]

defines NAME1 through NAMEn as manifest constants. (If there is only one NAME, the brackets are not necessary.) The expressions CONST1 through CONSTn must be constant expressions; that is, their values must be computable by the compiler. The meaning of a program would be unchanged if each manifest name were replaced by a string of digits representing its value. In particular, manifest names do not have addresses.

3-4 . . . . . Structure Declarations

(Structures are described in Section 6 of this manual.)

3-5 . . . . . Static and External Variables

Static variables may be declared in four ways: by a static or external declaration, by a procedure declaration, or by a statement label assignment.

The declaration

static [NAME1 = CONST1; ...; NAMEn = CONSTn]

defines NAME1 through NAMEn as static variables, and causes them to be initialized with the values CONST1 through CONSTn at the beginning of program execution (i.e., in the "save file" created by the loader). (If there is only one NAME, the brackets are not necessary.) The CONSTs must be expressions whose values are computable by the compiler. If it doesn't matter what the variable is initialized to, the " = CONST" should be left out, or " = nil" should be used.

Any of the NAMEs that are preceded by an "@" will be allocated by the loader in page zero. Such variables are called "common" variables. They can be addressed directly by the compiled code, whereas normal static variables must be addressed by indirection through a literal; so common variables are more efficient. However, there is room in page zero for only about 150 (decimal) common variables; the loader will complain if too many common variables are assigned.

The procedure declarations

let NAME(ARG, ..., ARG) be STAT

let NAME(ARG, ..., ARG) = EXP

declare NAME as a static variable which is to be initialized by the loader to the address of the code compiled for the procedure.

The procedure declaration is discussed fully in the sections on procedure and dynamic variable declarations.

A statement label assignment

NAME: STAT

declares NAME as a static variable which is to be initialized by the loader to the address of the code compiled for STAT. A label assignment does not begin a block; the name is treated as if it were declared immediately after the declaration which begins the smallest enclosing block. Thus, a label is defined throughout the block in which it appears.

The declaration

external [NAME1; ...; NAMEn]

declares NAME1 through NAMEn as external static variables. (If there is only one NAME, the brackets are not necessary.) The purpose of the external declaration is to allow separately compiled pieces of a program to reference the same variables. Within a given source file, the scope of an external variable is the same as that of other types of variables; but if two or more separately compiled source files declare a given name external, the loader will make each refer to the same cell. In (exactly) one of the source files in which a given name is declared external, the name should also be declared as a static variable (by a static declaration, a procedure declaration, or a statement label assignment) someplace within the scope of the external declaration. (Note that the static declaration must follow the external declaration.) This is not a re-definition of the name, but rather tells the loader how to initialize the external static variable. The loader will complain about an external variable which is not declared static someplace, or about one which is declared static more than once.

NAMEs that are preceded by an "@" in an external declaration will be defined as common variables. A NAME that is declared both external and static may be designated as common in either or both declarations.

Note that only static variables may be external.

3-6 . . . . . Procedure Declarations

There are two kinds of BCPL procedures: "functions", which return a value upon completion, and "routines", which do not. A function is defined by a declaration of the form

let NAME (ARG1, ..., ARGn) = EXP

A routine is defined by

let NAME(ARG1, ..., ARGn) be STAT

NAME is the name of the function or routine being defined. (Actually, NAME becomes a static variable which will be initialized with the address of the procedure, as noted in the section on static variables.) ARG1 through ARGn are the formal parameters (dummy arguments) of the procedure. They are either NAMEs, or the special symbol "nil", indicating an unnamed argument. ARG1 through ARGn become the first n dynamic variables declared in the procedure body. If there are no dummy arguments, the declaration is of the form "let NAME( ) be STAT" or "let NAME( ) = EXP".

In the function declaration, EXP is the expression whose value is returned when the function is called. EXP may be a simple BCPL expression; but for most functions it will be an expression of the form "valof STAT", where STAT may be a compound statement. The STAT in a "valof" expression should contain at least one "resultis" statement. The STAT is executed until a statement of the form "resultis EXP" is encountered; then EXP becomes the value of the "valof" expression, and therefore the result of the function. The "valof" expression will also terminate when control would otherwise pass to the statement following STAT. If this happens, the value of the "valof" expression is garbage.

In the routine declaration, STAT is the statement which is executed when the routine is called. STAT may be a compound statement. STAT may contain one or more "return" statements; the routine returns when a "return" statement is executed, or when control would otherwise pass to the statement following STAT.

A multiple procedure declaration has the form

```
let NAME1(ARG, ..., ARG) be STAT (=EXP)
and NAME2(ARG, ..., ARG) be STAT (=EXP)
...
and NAMEn(ARG, ..., ARG) be STAT (= EXP)
```

This declares the procedures NAME1 through NAMEn "simultaneously"; that is, all of the NAMEi's are known in each of the procedure bodies. (So, for example, NAME1 can call NAME2 and NAME2 can call NAME1.) The ARGs, of course, are defined only in their corresponding procedure bodies.

A procedure body may contain procedure declarations; the names of such procedures will be local to the defining body (unless they are declared external). But remember rule (3) in the section on the scope of dynamic variables: dynamic variables are defined only in the body of the defining procedure, and not in sub-procedure bodies. For this reason, all procedures in a BCPL program are usually defined at the top level.

3-7 . . . . . Procedure Execution

A procedure is called by a statement or expression of the form

```
EXP(EXP1, EXP2, ..., EXPn)
```

EXP determines the procedure to be executed; EXP1 through EXPn are the actual parameters. If there are no actual parameters, the form is "EXP( )". A procedure call is an expression if it appears in a context in which a value is expected (e.g., in the right-hand side of an assignment statement); otherwise, it is a statement. The calling mechanism is the same in either case. The only difference is that in the context of an expression, the procedure is expected to return a value; if it doesn't (because it is a "routine" rather than a "function"), a garbage value will be used. A value which is returned by a function called in the context of a statement is discarded.

EXP will usually be a NAME which is either declared in a procedure declaration in the current source file, or declared external in the current file and declared as a procedure in another file. But in general, EXP may be an arbitrary BCPL expression; for example: "(n eq 0 ? f, g) (x, y)". The formal rule is that the location referenced by the expression "rv EXP" is the location to which control is to be transferred (via a "JSR"). The section on Runtime Environment goes into more detail on this.

When a procedure is entered, it first allocates some "frame" space from someplace in memory. This "frame" is a block of memory which the procedure will use for the actual parameter values, for any dynamic variables and vectors declared within the procedure, and for any temporary storage needed by the procedure. The space is de-allocated when the procedure executes the "return" or "resultis" corresponding to the call that allocated the frame.

After the frame space is allocated, the values of EXP1 through EXPn are stored in the first n words of the frame. These n words are those referenced by the n formal parameters ARG1, ..., ARGn in the procedure declaration, assuming that the procedure is called with exactly the number of actual parameters as it was declared to have. (No check is made to see if actual and formal parameters match. If there are fewer actual parameters, the formal parameters with no corresponding actual parameters will have garbage values. If there are more actual parameters than formal parameters, the actual parameters with no corresponding

formal parameters will be lost; but this may create havoc by clobbering memory words beyond the end of the newly created frame.)

Note that each formal parameter takes on the value of its corresponding actual parameter at the beginning of the procedure call. This implies that procedure calls are implemented by the "call by value" mechanism (in the ALGOL sense); assigning a value to a formal parameter within a procedure does not affect the value of the corresponding actual parameter in the calling routine, although it does change the value of the formal parameter for the remainder of the procedure execution. Suppose the function "next" is defined by:

```
let next(x) = valof [ x = x + 1; resultis x ]
```

and called as follows:

```
a = 0; b = next (a)
```

After the call of next, "a" will still be 0, but "b" will be 1. We can write "next" in such a way as to allow it to change the value of "a" by using the address-manipulation primitives of BCPL:

```
let next (xaddr) = valof
    [ rv xaddr = rv xaddr + 1; resultis rv xaddr ]
```

Then calling "next" as follows:

```
a = 0; b = next (lv a)
```

will cause both "a" and "b" to have the value 1.

After the procedure frame has been allocated and the actual parameters have been stored in the frame, the procedure body is executed. If the procedure terminates normally (with "return" or "resultis", or by falling through the last statement), the frame space is deallocated and control returns to the caller. If the procedure exits with a "goto", the frame space is not deallocated, and the frame pointer is not changed. This is a bad thing to do.

3-8 . . . . . Dynamic Variables

A dynamic variable refers to a cell at some fixed position in the frame associated with the current execution of the procedure in which it is defined. This cell is only allocated to the variable while the block defining the variable is active (e.g., while the block is being executed, or while a procedure called from within the block is being executed). Outside of the block, the cell is used for something else.

Dynamic variables are declared in two ways: in a dynamic variable declaration, and as formal parameters in a procedure declaration.

The dynamic variable declaration

```
let NAME1, ..., NAMEn = EXP1, ..., EXPn
```

allocates n consecutive frame cells to NAME1 through NAMEn, and compiles code to assign the values of EXP1 through EXPn to NAME1 through NAMEn. Unlike other declarations, this declaration is executable; for a given execution of a procedure, NAME1 through NAMEn always refer to the same frame cells, but the values stored in these cells are recomputed each time the declaration is executed. The assignment is done left-to-right.

The EXPs may be any BCPL expression. In addition, there are two special cases: "nil" and "vec CONST".

If EXPi is the symbol "nil", the variable NAMEi is declared, but no value is assigned to NAMEi. Thus, "let x = nil" declares x, but compiles no code; "x" will have some garbage value until something is assigned to it.

If EXPi is the special expression "vec CONST" (where CONST is an expression that can be evaluated by the compiler), the value assigned to NAMEi will be the address of the first word of a block of CONST+1 consecutive frame cells. This "vector" of CONST+1 cells is allocated from the frame space, and NAMEi is initialized to point to that vector. These cells exist as long as NAMEi exists; they are used for something else outside of the block in which the declaration appears.

In a procedure declaration

```
let NAME(ARG1, ..., ARGn) be STAT
```

or

```
let NAME(ARG1, ..., ARGn) = EXP
```

ARG1 through ARGn are declared as dynamic variables; their scope is the entire procedure body. (Recall that the declaration defines NAME as a static variable.) The declaration is equivalent to

```
let NAME( ) be
    [ let ARG1, ..., ARGn = nil, ..., nil; STAT ]
```

or to

```
let NAME( ) = valof
    [ let ARG1, ..., ARGn = nil, ..., nil; resultis EXP ]
```

That is, ARG1 through ARGn are the first n dynamic variables declared in the procedure body, and therefore refer to the first n cells in the frame. The procedure call "NAME(EXP1, ..., EXPm)" stores the values of the m actual arguments in the first m cells of the newly created frame. So if m > n, cells n + 1 through m will be clobbered. If m = n, all is well. If m < n, ARGs m + 1 through n will have garbage values. This permits procedures to be called with a variable number of actual arguments, as long as enough formal arguments are declared to provide space for the largest actual argument list. For example, if we define a procedure something like

```
let f(x0, x1, x2, ..., x20) be
    [     let arg = lv x0
          ... arg!i ...
    ]
```

then the expression "arg!i" references the ith argument.

The ARGs are usually NAMEs, but the special symbol "nil" is also legal as an ARG. The "nil" has the effect of leaving space for an argument, but not declaring a name for that argument. So the procedure "f" above might also have been defined as

```
let f(x0, nil, nil, ..., nil) ...
```

Argument i can still be referenced by "arg!i".

In procedures which are called with a variable number of arguments, the "numargs" facility may be useful. An argument list in a procedure declaration may take the form

```
let NAME(ARG1, ..., ARGn ; numargs NAME) ...
```

The NAME following " ; numargs" is declared as a dynamic variable in the procedure body; when the procedure is entered, NAME is set to the number of actual arguments in the procedure call. Note the semicolon preceding "numargs".

SECTION 4
EXPRESSIONS

4-1 . . . . . Memory References

There are four kinds of BCPL expressions which refer to memory cells: variable names, rv-expressions, vector reference expressions, and structure reference expressions. These are the only things that can appear as the left-hand side of an assignment statement "REF = EXP" or as the argument of an lv-expression "lv REF". In an assignment statement, REF specifies the cell to be modified. The value of an lv-expression is the address of the cell specified by REF. (These two contexts are the only ones in which the form of the expression is restricted.) In all other contexts, the value of a memory-reference expression is the value contained in the specified cell.

Memory reference expressions are described below in terms of the Nova instructions compiled. There are six Nova op-codes that reference memory: LDA ac, STA ac, JMP, JSR, ISZ, DSZ. The symbol "OP" in the description below designates one of these op-codes; the address of the op-code is in standard Nova form (@ displacement, index). In general, an assignment statement generates a STA; a procedure call generates a JSR; and other contexts generate a LDA.

dynamic variable names:

> Dynamic variables are allocated cells in the first 200 (octal) words of the frame for the procedure in which they are declared. While a procedure is being executed, AC2 always points at the procedure's frame; so dynamic variables are referenced by "OP n,2", where "n" is the offset of the dynamic variable in the frame. This imposes a limit on how many dynamic variables a procedure may declare; the practical limit is about 100 (decimal) dynamic names in a given scope. (Because the frame is allocated dynamically when a procedure is called, dynamic variables cannot be accessed directly from any procedure other than the one in which they are declared, as noted in scope rule (3) in Section 3.)

static variable names:

> Static variables are allocated space by the loader, either in "common" (page zero) or in another area of memory which is fixed during loading. Common variables are accessed by "OP n,0", where $0 < n < 377$. Other static variables are not directly addressable, since they are in some arbitrary area of core, so they are addressed through indirection by "OP @n,1" (that is, "OP @.+n"), where n is the PC-relative offset ($-200 < n < 177$) of a word containing the address of the static variable.

vector references: EXP1 ! EXP2

> This expression references a memory cell whose address is given by the value of (EXP1 + EXP2). The reason for calling an expression like "A!I" a "vector reference" is the following. Suppose that the value of the variable "A" is the address of the first word of a zero-origin one-dimensional array (a "vector"). Then the expression "A!I" references the Ith word of the vector A, since the value of the expression "A+I" is the address of this word. Note that the "!" operator is commutative.
>
> In general, vector references generate code to compute the sum of EXP1 and EXP2 in AC3 (e.g., "LDA 0,EXP1; LDA 3,EXP2; ADD 0,3"), and then reference the vector element with

"OP 0,3". In the case where EXP2 (or EXP1) is a small constant (-200 < n < 177), EXP1 (or EXP2) is loaded into AC3, and the vector element is accessed by "OP n,3". In any case, a vector reference always uses indexing through AC3. See the note on rv-expressions below.

rv-expressions: rv EXP, @EXP:

This expression references a memory cell via indirect addressing through EXP. In general, the value of EXP is computed and stored in a temporary cell in the frame, and the reference is done by "OP @n,2", where n is the offset of the temp cell. There are several special cases: If EXP is a dynamic variable name, "OP @n,2" is used, where n is the frame offset of the variable. If EXP is a common variable name, "OP @n,0" is used, where n is the page zero address of the variable. On the Nova, if EXP is a static variable name, "OP @n,1" is used (that is, "OP @.+n), where n is the PC-relative offset of a word containing the address of the static variable with the indirect bit (bit 0) set. If EXP is a vector reference, "OP @n,3" is used, after loading AC3 appropriately.

The expression "rv EXP" may also be written "@EXP".

An rv-expression always generates an indirect reference through a memory cell. A vector reference always generates an instruction which is indexed by AC3. Therefore, on the Nova, "rv EXP" is not necessarily equivalent to "EXP1!EXP2" when the values of (EXP) and (EXP1 + EXP2) are the same: the rv-expression will always cause a multiple indirection if EXP has bit 0 set; a vector reference will never do so, since indexing ignores bit 0. On the Alto the two are always the same, since all 16 bits are part of the memory address.

structure reference expressions:

These are described in the section on structures.

4-2 . . . . . Constants

BCPL recognizes the following constructs as constants:

* A name which is declared "manifest" is treated as if it had been replaced by its value.

* A string of digits is interpreted as a decimal integer. It may not exceed 2**15-1 (32767 decimal, 77777 octal).

* A string of digits preceded by a "#" is interpreted as an octal integer. It must be less than 2**16-1 (177777 octal, 65535 decimal).

* A string of digits immediately followed by "B" or "b" is also interpreted as an octal integer. If the "B" or "b" is immediately followed by a (decimal) number n, the octal value is shifted left n bits. Thus, #1230, 1230B, and 123B3 all represent the same value. One-bits may not be shifted out of bit 0.

* The reserved words "true" and "false" are constants with values #177777 and 0 respectively.

* A "$" followed by any printing character other than "*" represents a constant whose value is the 7-bit ASCII code of the character. "*" is an escape character; the following escapes are recognized:

  *s *S space (#40)

  *t *T tab (#11)

*n *N	carriage return (#15)
*c *C	carriage return (#15)
*l *L	line feed (#12)
*"	double quote (#42) [$" is also O.K.]
*nnn	The octal number "nnn". [Exactly three digits.]
**	* (#52)
Note:	"*" followed by anything else gives an error.

The compiler evaluates most expressions that involve only constants, and treats the resulting value as a single constant. (The exceptions are "selecton" and "valof" expressions. Conditional expressions like "CONST ? CONST1, CONST2" are evaluated; the value is CONST2 if CONST is 0, and CONST1 otherwise.) Throughout this manual, the symbol "CONST" (described as "an expression which can be evaluated by the compiler") means either one of the constant constructs above, or an expression involving only constants.

4-3 . . . . . Precedence of Expressions

In order of decreasing precedence, the legal BCPL expressions are:

NAME; constant; string literal; table literal; (EXP)

EXP(EXP1, ..., EXPn)

EXP1!EXP2

EXP>>NAME.NAME.... ; EXP<<NAME.NAME....

lv EXP; rv EXP; + EXP; -EXP

EXP1 <mul> EXP2 (<mul>: *, /, rem, lshift, rshift)

-EXP1 + EXP2; EXP1 - EXP2

vec CONST

EXP1 <rel> EXP2 (<rel>: eq, ne, ls, le, gr, ge)

not EXP

EXP1&EXP2

EXP1%EXP2

EXP1 xor EXP2; EXP1 eqv EXP2

EXP ? EXP1, EXP2

selecton EXP into ...

valof STAT

Operators with the same precedence are left-associative, except for "<mul>", "&", "%", "xor", and "eqv", which are right-associative. Precedence and associativity can be changed by parenthesizing. Some cases to note:

"a/b*c" is "a/(b*c)"

"rv v!i" is "rv(v!i)"

"rv p>>a.b" is "rv (p>>a.b)"

"v!p>>a.b" is "(v!p)>>a.b"

"v!i+j" is "(v!i)+j"

"a%b&c" is "a%(b&c)"

"a & b eq c" is "a & (b eq c)"

Precedence only determines the way in which an expression is parsed; nothing is implied about order of evaluation. In general, the order in which the sub-expressions of an expression are computed is unspecified. So, although "f(x) + g(y) * h(z)" means "f(x) + (g(y) * h(z))", no assumption should be made about which function is executed first.

4-4 . . . . . BCPL Expressions

string literals

A sequence of characters enclosed in double quotes (") is a string literal. Its value is the address of the first word of a block of memory containing the string. A BCPL string is stored two bytes per word, left-hand byte first, with the left-hand byte of the first word containing the number of characters in the string. If the string has an even number of characters, the right-hand byte of the last word is 0; but if it has an odd number of characters, the last word of the string contains the last two characters, not two 0 bytes. Note that BCPL strings are not compatible with Nova DOS strings.

Strings have a maximum length of 255 characters. The character "*" appearing in a string literal is an escape character, as described for character constants.

table [CONST1; ...; CONSTn]

The value of a table expression is the address of the first word of a block of memory containing the CONST values.

EXP ( )
EXP (EXP1, EXP2, ..., EXPN)

The value of EXP is assumed to be the address of a BCPL function. This function is called with the values of EXP1, ..., EXPN as arguments. The value of the function call is the value returned by the function via a "resultis" statement. See the section on procedure execution for details.

The call is implemented by a Nova JSR instruction (a memory reference op-code) to "rv EXP". So if EXP has bit 0 set, a multiple indirection will take place. If bit 0 is zero, the value of EXP is the address of the first instruction executed.

The empty argument list "( )" is necessary if there are no arguments. "x = f( )" calls a function, but "x = f" puts the address of the function in "x". Forgetting the "( )" is a common error; be careful.

lv REF

REF must be a variable name, a vector reference, an rv-expression, or a structure reference; anything else gives an error message. The value of the lv-expression is the address of the cell which REF references (but see the note on "lv(rv EXP)" below).

The value of "lv NAME", if NAME is a dynamic variable, is the sum of the current frame pointer (which is in AC2) and the offset of the variable in the frame (a constant). This address is valid only while the block in which the variable was declared is active.

The value of "lv NAME", where NAME is a static variable, is the address of the static variable. This is a constant throughout the execution of the program, since static variables never move. (But "lv NAME" is not a compile-time CONST.)

The value of "lv(EXP1!EXP2)" is the sum of the values of EXP1 and EXP2.

The value of "lv (rv EXP)" is the address of the cell that "rv EXP" references. On the Nova, if EXP has bit 0 set, "rv EXP" would cause a multiple indirection; in this case, the value is computed by following the indirection chain. There is nothing special about bit 0 on the Alto; it is just another bit of the address.

The value of "lv (EXP>>NAME.NAME....)" is the address of the word which contains the first bit of the referenced field.

rv EXP
EXP1 ! EXP2

See the section on Memory References (Section 4-1).

+EXP

The value is the value of EXP.

-EXP

The value is the two’s-complement of the value of EXP.

EXP1 * EXP2

The value is the low-order 16 bits of the 32-bit signed product. If one of the EXPs is a constant whose value is a power of 2, a left shift is done; otherwise the standard Nova multiply sequence is done. There is currently no way to get at the high-order part of the product, or to detect overflow.

EXP1 / EXP2
EXP2 rem EXP2

The standard Nova signed integer divide sequence is done. (Division by a power of 2 is not done by shifting.) The "/" expression gives the 16-bit signed quotient; the "rem" expression gives the 16-bit remainder, which has the same sign as EXP1. If EXP2 is zero, the results are undefined. There is currently no way to detect this.

EXP1 lshift EXP2
EXP1 rshift EXP2

The value is the value of EXP1 shifted left or right EXP2 bits. Vacated positions are filled with 0's. Bits shifted off either end of the 16-bit word are lost. The shifts are logical, not arithmetic, in that the sign bit may be changed. There are currently no arithmetic- or circular-shift operators.

EXP1 + EXP2
EXP1 - EXP2

The value is the sum (difference) EXP1 and EXP2. The statement "EXP = EXP + 1" generates an ISZ or DSZ followed by a NOP. There is currently no way to detect overflow.

EXP1 eq EXP2
EXP1 ne EXP2
EXP1 ls EXP2
EXP1 le EXP2
EXP1 gr EXP2
EXP1 ge EXP2

EXP1-EXP2 is computed and compared with 0; the value of the relational expression is always either "true" (#177777) or "false" (0). *Warning: This differs from a genuine signed comparison of EXP1 and EXP2 if |EXP1-EXP2| is greater than 2**15-1.*

not EXP

The value is the logical complement (one's-complement) of the value of EXP. But see the note on "&" and "%" below.

EXP1 & EXP2
EXP1 % EXP2

In most contexts, the value is the logical-and or logical-or of EXP1 and EXP2. However, in the context of the Boolean part of an "if", "unless", "test", "while", "until", "repeatwhile", or "repeatuntil" statement, or of a conditional expression, the evaluation of an expression involving "not", "&", or "%" is optimized. This optimization can change the meaning of the expression. For example, the sequence "if a & b then ..." is not always the same as the sequence "x = a&b; if x then ...", even if the evaluation of "a" and "b" do not involve side effects. See the section on conditional statements.

EXP1 xor EXP2
EXP1 eqv EXP2

The value of the "xor" expression is the logical exclusive-or of EXP1 and EXP2. The value of the "eqv" expression is the logical complement of this value.

EXP ? EXP1, EXP2

The value is the value of EXP1 if EXP is non-zero, or the value of EXP2 if EXP is zero. EXP is optimized if it involves "not", "&", or "%"; see the section on conditional statements.

valof STAT

This expression causes the statement STAT to be executed until a "resultis EXP" statement is encountered or until control would otherwise pass to the statement following STAT. If a "resultis EXP" is executed, EXP becomes the value of the "valof STAT" expression. If execution of STAT terminates, the expression has a garbage value. The "valof" expression is usually used as a function body; but it may be used anyplace an expression can be.

selecton EXP into
[case CONST1: EXP1
...

```
    case CONSTn:  EXPn
    default:  EXP0
]
```

This expression is equivalent to

```
valof switchon EXP into
    [   case CONST1:  resultis EXP1
        ...
        case CONSTn:  resultis EXPn
        default:  resultis EXP0
    ]
```

That is, its value is EXPi if the value of EXP is CONSTi, or EXP0 if EXP is not equal to any of the CONSTs. If no "default" label appears, the "selecton" expression will have a garbage value if none of the cases is matched.

newname NAME

This expression evaluates at compile time to "true" if the NAME is appearing in the source file for the first time. It evaluates to "false" if it has appeared before (including previous "newname" constructs). This construct is useful in conjunction with conditional compilation or the /M compiler switch (command-line declarations).

SECTION 5
STATEMENTS

5-1 . . . . . Assignment Statements:

REF = EXP

The value of EXP is stored into the memory cell referenced by REF. See the section on Memory References (Section 4-1).

REF1, ..., REFn = EXP1, ..., EXPn

This statement is equivalent to the sequence "REF1 = EXP1; ...; REFn = EXPn". The assignments are made left-to-right.

5-2 . . . . . Routine Calls:

EXP ( )
EXP(EXP1, EXP2, ..., EXPn)

A routine call differs from a function call only in that a routine call occurs in a context where a statement is expected, whereas a function call occurs in a context where an expression (a value) is expected. The calling sequence for routines is identical to that for functions.

5-3 . . . . . Conditionals and Iterative Statements:

The evaluation of EXP in an "if", "unless", "test", "while", "until", "repeatwhile", or "repeatuntil" statement is optimized if EXP involves "not", "&", or "%". In general, EXP "succeeds" if it is non-zero, "fails" if it is 0. But "EXP1&EXP2" is tested by first testing one of the EXPs; if it "fails", the &-expression "fails", and the other expression is not evaluated. Similarly, in "EXP1%EXP2", one of the EXPs is tested; if it "succeeds", "EXP1%EXP2" succeeds. A "not EXP" "succeeds" if EXP "fails", and "fails" if EXP "succeeds".

This optimization has two significant consequences:

a) In a statement such as "if f(x) & g(x) do ...", it is not guaranteed that both functions will be executed; so any side-effects of "f" and "g" cannot be depended on.

b) The statement "if x & y do ..." is not necessarily equivalent to the sequence "z = x&y; if z do ...". For example, if "x" has the value 1 and "y" has the value 2, "z = #x&y" would assign the value 0 to "z", because "1&2" is zero; so "if z do ..." will consider "z" to "fail". But both "x" and "y" are nonzero, so "if x&y do ..." will consider "x&y" to "succeed". In general, "&" should be used in conditional statements only when its operands are known to take on only the values "true" (#177777) or "false" (0). Note that this is the case for relations; so "if x ne 0 & y ne 0" does the right thing.

if EXP do STAT
unless EXP do STAT

The "if" statement executes STAT if EXP succeeds. The "unless" statement executes STAT if EXP fails. The word "do" may be replaced by the word "then", but (unlike ALGOL) no "else" clause is allowed; use the "test" statement for two-armed conditionals. The "do" or "then" may be omitted if STAT appears on the same line as the "if" or "unless" clause, and if STAT is one of the following types of statements:

"if" "unless" "test" "while" "until" "for" "goto" "return" "resultis" "switchon" "break" "loop" "endcase" "docase"

test EXP then STAT1 or STAT2
test EXP ifso STAT1 ifnot STAT2
test EXP ifnot STAT2 ifso STAT1

Each of the above "test" statements executes STAT1 if EXP succeeds, or STAT2 if EXP fails. Both clauses must be present; use the "if" statement or the "unless" statement for one-armed conditionals. If "then" and "or" are used, they must appear in that sequence; the STAT following "then" is the true branch. If "ifso" and "ifnot" are used, they may appear in either order; the STAT following "ifso" is the true branch.

while EXP do STAT
until EXP do STAT

The "while" statement executes STAT as long as EXP succeeds. The "until" statement executes STAT as long as EXP fails. The test on EXP is done before the first execution of STAT. The word "do" may be omitted in the same contexts as for the "if" statement.

The "while" statement is equivalent to:

"goto M; L: STAT; M: if EXP goto L"

The "until" statement is equivalent to

"goto M; L: STAT; M: unless EXP goto L"

STAT repeatwhile EXP
STAT repeatuntil EXP

The "repeatwhile" statement executes STAT as long as EXP succeeds. The "repeatuntil" statement executes STAT as long as EXP fails. STAT is executed once before the test on EXP is done. STAT may be a single statement or a compound statement.

The "repeatwhile" statement is equivalent to:

"L: STAT; if EXP goto L"

The "repeatuntil" statement is equivalent to:

"L: STAT; unless EXP goto L"

STAT repeat

The "repeat" statement executes STAT repeatedly (until terminated by a "break", "return", "resultis", "endcase", "docase", or "goto" statement). It is equivalent to:

"L:STAT; goto L"

for NAME = EXP1 to EXP2 by CONST do STAT

NAME is a legal variable name; EXP1 and EXP2 may be arbitrary expressions; "by CONST" may be missing (1 is assumed), but if present, it must be a constant expression. The "for" statement is (logically) equivalent to the following block:

```
[       let NAME, lim, inc = EXP1, EXP2, CONST
        goto M
L:      STAT
        NAME = NAME + inc
M:      test inc ge 0
        ifso if NAME ge lim goto L
        ifnot if NAME le lim goto L
]
```

Several things about the "for" statement should be noted:

1) The controlled variable is implicitly declared as a new dynamic variable; it is defined only in STAT, and not accessible after the loop terminates.

2) EXP2 is evaluated only once, at the beginning of the "for" statement.

3) As noted, CONST (if present) must be a constant expression. If it is negative, the termination test is reversed.

4) STAT is not executed if the initial condition fails the termination test (like ALGOL, unlike FORTRAN).

5) STAT is executed when the controlled variable is equal to the limit.

break
loop

These are single-word BCPL statements which are legal only in the context of an iterative statement. The effect of "break" is to jump to the statement immediately following the smallest textually enclosing iterative statement. The effect of "loop" is to jump to the point at which the next iteration starts: to the test in a "while", "until", "repeatwhile", or "repeatuntil" statement; to the increment of NAME in a "for" statement; or to the beginning of a "repeat" statement.

5-4 . . . . . Conditional Compilation Statements:

compileif EXP then [<sequence>]

compiletest EXP then [<sequence>]

These constructs allow alternative code sequences to be chosen at compile time; they are analogous to "if" and "test." There are several restrictions on the use of these statements:

The EXP must be comprised of operations on manifest and numeric constants, so that it may be evaluated at compile time.

A conditional compilation construct can appear wherever a "let" would be legal (Not, for example, within a statement or declaration, or directly following "then," "ifso," "ifnot," or "case").

Although the syntax of conditional compilation parallels that of conditional statements, the brackets ([ ]) are mandatory. A <sequence> is a legally separated sequence of commands and declarations. The <sequence> may contain declarations which will apply to commands which follow the conditional construct, as long as the uses of the variable are also conditionally compiled.

Conditional selections are done at a time after "get" files have been read. As a result, "get" commands are unaffected by conditionals -- the files are always read.

The auxillary constructs "ifso," "ifnot," "then," "do," and "or" may all be used with the conditional compilation tests:

compiletest EXP then [<sequence1>] or [<sequence2>]

5-5 . . . . . Labels and Goto Statements:

NAME: STAT

Any BCPL statement may be labeled. A label is effectively a declaration of a static variable which is initialized with the address of the labeled statement. It differs from other declarations in that it does not implicitly start a new block. Instead, it is treated as if it appeared at the beginning of the smallest textually enclosing block. See the section on static declarations for details.

goto EXP

A Nova JMP is done to "rv EXP". The EXP is usually a label, but need not be. Control is transferred to the memory location which is referenced by "rv EXP".

5-6 . . . . . Returns:

return
resultis EXP

These statements cause a return from the procedure in which they appear. "return" is only legal in a routine body; "resultis EXP" is only legal in a function body.

5-7 . . . . . Switches:

switchon EXP into CASEBLOCK

CASEBLOCK is a BCPL block which contains labels of the form "case CONSTi:", where the CONSTi are constant expressions. CASEBLOCK may also contain a label of the form "default:". The effect of a "switchon" statement is as follows: If the CASEBLOCK contains a "case" label whose constant CONSTi is equal to the value of EXP, a jump is done to that label. If no CONSTi matches the value of EXP, a jump is done to the "default" label if there is one, or to the statement immediately following the CASEBLOCK if there is no default label.

The appearance of a "case" label does not terminate the preceding case. That is, in

```
switchon Char into
[       case $A:x = 1
        case $B:x = 2
        default:x = 0
]
```

"x" will be 0 no matter what "Char" contains. The statements "x = 1" and "x = 2" should be followed by a jump to the end of the CASEBLOCK. The single-word BCPL statement "endcase" would accomplish this.

Case labels are legal only in CASEBLOCKs, and not in any sub-blocks of a CASEBLOCK. In connection with this, recall that a declaration implicitly begins a new block. Therefore the sequence

```
switchon x into
[       case 0:         let temp = 0
                        ...
        case 1:         ...
]
```

will cause the compiler to complain that "case 1:" does not appear in a CASEBLOCK. The code which uses "temp" must be enclosed in a block of its own which does not span other case labels.

Switches are implemented by grouping the case values into one or more value ranges in which listed values are fairly dense, and doing an indexed branch on each of these ranges. Case values which do not fall into these clusters are checked individually if all of the indexed branches fail.

endcase

This single-word statement is legal only within the scope of a "switchon" statement. It causes a transfer to the end of the smallest enclosing "switchon" statement.

docase EXP

This statement is legal only within the scope of a "switchon" statement or "selecton" expression. It causes a transfer to the case label denoted by EXP within the smallest enclosing CASEBLOCK, by performing the switching activities again using EXP as an index. This construct allows one to merge several cases with a terminating case, or to generate flexible looping constructs. The unlikely sequence

```
i = 5; s = "STR0"
switchon i into
[       case 0: write(s); endcase
        case 1: s = "STR1"; docase 0
        ...
        case 5: s = "STR5"; docase 0
        ...
]
```

would cause the string "STR5" to be written.

5-8 . . . . . Single-Word Statements

finish
abort

These single-word statements terminate execution of the program (on the Nova by a ".RTN" system call). The "abort" statement causes a message to be typed on the terminal. DOS

return
break
loop

These statements are described above.

SECTION 6
STRUCTURES

6-1 . . . . . Structure declarations and references

The structure facility allows the user to define templates for symbolically referencing partial-word fields of variables, and individual words and partial-words of vectors. (A "vector" in BCPL means any block of consecutive memory words). For example, a program which manipulates rectangular areas on a display might be using four-word blocks in memory to represent the center coordinates, width, and height of the significant areas on the screen. This program could declare a structure for referencing these blocks as follows:

```
structure rectangle :   [ x        word
                          y        word
                          width    word
                          height   word
                        ]
```

The structure is used in conjunction with the ">>" operator. For example, if the program has a variable cursor which points at (i.e., contains the address of the first word of) a four-word block, the expression cursor>>rectangle.width references the width field of that block, and is equivalent to the expression cursor!2. So the program can contain statements like

```
cursor>>rectangle.width = 1
```

and

```
let cursortop = cursor>>rectangle.x + cursor>>rectangle.height
```

The declaration defines rectangle as a four-word structure, with fields named x, y, width, and height, each of which is one word wide. The fields of a structure are positioned sequentially, so the x field refers to the first word of a referenced block, the y field to the second word, etc.

The operator ">>" (pronounced "right-lump") expects an expression on the left, and a description of the field to be referenced on the right. The value of the left-hand expression is taken as the address of the block of memory to be referenced. The right-hand side, in the simplest cases, consists of the name of the structure describing the block, followed by ".", followed by the name of the field to be referenced. The left precedence of ">>" is higher than that of all expression operators except procedure calls and vector subscripts; so

```
a(b)>>s.f        means        (a(b))>>s.f

a!b>>s.f         means        (a!b)>>s.f
```

but all other left-hand operands of ">>" must be parenthesized.

It is often convenient to define a structure consisting of a field list at the outermost level, without a single top-name. For example:

```
structure          [ x          word
                     y          word
                     width      word
                     height     word
                   ]
```

This structure describes a configuration of fields identical to that of rectangle. However, references to the fields of the structure require only the field name, as in cursor>>width.

Structures may also contain partial-word fields, as in the following example:

```
structure area :   [ visible    bit 1
                     blinking   bit 1
                     color      bit 5
                     x          bit 9
                     blank      bit 2
                     border     bit 5
                     y          bit 9
                     width      byte
                     height     byte
                   ]
```

This structure describes three-word blocks which hold various pieces of information about rectangular areas of the display. The field-size specifier bit N, where N is a constant expression, defines a field which is N bits wide; the specifier byte defines a field which is 8 bits wide. A bit field may not overlap a word boundary; the special name blank (a reserved word) is used in the above declaration to leave an unnamed two-bit field in the second word in order to prevent such an overlap. A byte field must begin on a byte boundary. A word field must begin on a word boundary. No automatic filling-out to boundaries is done; blank fields must be supplied explicitly when needed.

With the above definition of area, assuming that cursor points at an area block, we reference the width field with cursor>>area.width, just as for rectangle. But the definition of area makes this a reference to the leftmost 8 bits of the third word of the vector cursor. The statement

```
cursor>>area.width = w
```

is equivalent to

```
cursor!2 = ((w lshift 8) & #177400) + (cursor!2 & #377)
```

(The structure reference generates much better code than this). The rightmost 8 bits of cursor!2 are unchanged. Similarly, the statement

```
w = cursor>>area.width
```

stores the left-hand byte of cursor!2 into w, right-adjusted, with 8 leading zero bits; it is equivalent to

```
w = (cursor!2 rshift 8) & #377
```

6-2 . . . . . Nested fields

A structure may contain substructures nested to any reasonable depth. For example, we might define a structure for vectors representing displayed lines of text as follows:

```
structure textline :    [ string      word
                          color       byte
                          linenum     byte
                          margin :    [ left        byte
                                        right       byte
                                        ]
                          font :      [ templates   word
                                        charsize :  [ width       byte
                                                      height      byte
                                                    ]
                                        ]
                        ]
```

Now if the variable title is a pointer to a five-word block of memory containing textline data, its fields are referenced by:

```
title>>textline.string
title>>textline.color
title>>textline.linenum
title>>textline.margin.left
title>>textline.margin.right
title>>textline.font.charsize.width
title>>textline.font.charsize.height
title>>textline.font.templates
```

That is, a field is specified to ">>" by a sequence of substructure names separated by ".", ending with the field name.

A substructure name may be used as a field name; that is, it may be the last name on the right-hand side of ">>". So

```
title>>textline.margin
```

is a legal structure reference expression, referring to the full word title!2. However, a ">>" expression may not refer to a field that is longer than 16 bits, or to one that overlaps a word boundary; so

```
title>>textline.font
```

is illegal, since the total length of font's subfields is 32 bits.

It is often the case that a group of fields in a structure are identical to those in another structure or substructure. For example, we might want to define a structure for vectors which represent rectangular display areas containing a word of text as follows:

```
structure sign :    [ text      word
                      textsize  byte
                      textcolor byte
                      visible   bit 1
                      blinking  bit 1
                      color     bit 5
                      ...
                    ]
```

That is, a sign contains all of the information for a area (visible, blinking, etc.), plus three additional fields. We can define sign as above without having to copy the field definitions of area as follows:

```
structure sign :    [ text          word
                      textcolor     byte
                      textsize      byte
                      @area
                    ]
```

Within a structure declaration, an "@" followed by a previously defined structure name is replaced by the body of that structure's definition. So the above definition of sign is equivalent to:

```
structure sign :    [ text          word
                      textcolor     byte
                      textsize      byte
                      [ visible                 bit 1
                        blinking    bit 1
                        color                   bit 1
                        ...
                      ]
                    ]
```

The brackets surrounding the inner field list have no effect, like unnecessary parentheses surrounding expressions. So references like stop>>sign.color are legal with either definition.

We could alternatively have made the area fields part of a substructure in sign as follows:

```
structure sign :    [ text          word
                      textcolor     byte
                      textsize      byte
                      textarea :    @area
                    ]
```

or even

```
structure sign :    [ text          word
                      textcolor     byte
                      textsize      byte
                      area :        @area
                    ]
```

In the latter case, references to the area fields look like stop>>sign.area.color.

6-3 . . . . . Subscripted fields

It is possible to have structure fields which are replicated, with individual replications referred to in structure reference expressions by integer subscripts. A simple example is a structure which describes BCPL-format strings:

```
structure string :  [ length                byte
                      char^1,255    byte
                    ]
```

A "^" following a field name in a structure declaration indicates that the field is to be replicated; the "^" is followed two constants, separated by "," , which specify the subscripts of the first and last replications. So in the above example, the field char is replicated 255 times, with the replications numbered from 1 thru 255. Now if s is a pointer to a BCPL string, the expression

```
s>>string.char^4
```

references the fourth character of the string, which is in the left half of s!2. A subscript in a structure reference expression may be an arbitrary BCPL expression; the precedence of the "^" operator is higher than any other operator, so any subscript other than a name or number must be parenthesized, e.g.,

```
s>>string.char^(i+j) = 0
```

In references to a subscripted field, the user must be sure to remember what low-subscript value was specified in the declaration. For example, in the above definition of string, the first character is referenced by

```
s>>string.char^1
```

and the last meaningful character by

```
s>>string.char^(s>>string.length)
```

But if the char field had been defined as char^0,254* byte, these references should be

```
s>>string.char^0
```

and

```
s>>string.char^(s>>string.length-1)
```

The low-subscript and high-subscript given in a structure declaration determine the number of bits occupied by the replicated field:

```
(high-low+1)**(number of bits in one replication)
```

Since a structure is only a template, and allocates no memory on its own, the only significance of this number is that it determines the position of subsequent fields, if any, in the structure. (It also determines the value of the size expression, which will be described later). In the string example, char is the last field, so it makes no difference how many replications are specified. But suppose that we had chosen to include a text string in sign blocks, rather than a pointer to the string in the first word. The definition of sign would then be:

```
structure sign :    [ @string
                      textcolor   byte
                      textsize    byte
                      area :      @area
                    ]
```

(Note the uses of the "@" construct). We would then reference the ith character of a sign with

```
stop>>sign.char^i
```

With this definition, space for the maximum-length string would have to be left in every sign block, since the expression stop>>sign.textcolor would be complied as a reference to the left half of stop!128. It would be better to specify @string as the last thing in sign, so that variable-length blocks could be used.

Any structure name, substructure name, or field name may be declared as subscripted, subject to the SUBSCRIPTED STRUCTURE RULE given below. For example, we might define a structure that describes tables of area descriptors as follows:

```
structure areatable :   [ numareas              word
                          area^1,100 : @area
                        ]
```

A areatable is a block of storage which contains some number of three-word subblocks, each of which is formatted as a area block. The first of the area blocks starts in the second word; the first word of a areatable holds the number of area blocks in the table. If the variable screen points at a areatable block, the expression

```
screen>>areatable.area^5.width
```

would reference the width field of the fifth three-word entry; that is, the left-half of screen!14. Note that the subscript is applied to the name which is replicated in the declaration (area), not at the end of the ">>" expression.

The above expression is somewhat unwieldy. There are two ways in which the structure could be modified so as to shorten the references to its subfields. One way is to eliminate the numareas field, and attach the subscript to the name areatable:

```
structure areatable^1,100 : @area
```

With this definition, the width field of the fifth entry would be referenced with

```
screen>>areatable^5.width
```

Note that if the numareas field had been included, it would have been replicated along with the area fields. (An extra word could be allocated above areatable blocks to hold the number of entries, and accessed as screen!-1; but there is no way to reference this word as part of the structure).

The second way in which areatable could be redefined is to post-subscript the area field list:

```
structure areatable :   [ numareas     word
                          @area^1,100
                        ]
```

This form of subscript declaration (subscript applied to a bracketed field list, which is what @area is equivalent to) replicates the substructure defined by the field list (100 three-word blocks in this example), but subscripts in references to the structure appear after the individual field names. So a reference to the width field of the fifth entry would be

```
screen>>areatable.width^5
```

Only the area fields are replicated; so it was possible to include the numareas field in this version of the structure.

Subscripted substructures may contain subscripted fields or sub-substructures to any depth. For example, we might describe a table of file names with:

```
structure filetable^1,50 :   [ length        byte
                                 char^1,15    byte
                               ]
```

The length of the ith name is referenced by

```
t>>filetable^i.length
```

and the jth character of the ith name by

```
t>>filetable^i.char^j
```

Multiple subscripts are also allowable. For example, a 4x3 matrix of double-precision numbers might be described by:

```
structure matrix^1,3^1,4 :    [ high        word
                                low         word
                              ]
```

This structure describes a storage area which consists of a four-fold replication of a three-fold replication of a two-word block. In references to a matrix block, the first subscript specifies which of the four outer replications is to be referenced, and the second indicates which of its three two-word blocks is wanted. So elements of a matrix appear in memory in the following order:

```
m>>matrix^1^1.high
m>>matrix^1^1.low
m>>matrix^1^2.high
m>>matrix^1^2.low
m>>matrix^1^3.high
m>>matrix^1^3.low
m>>matrix^2^1.high
m>>matrix^2^1.low
...
m>>matrix^4^3.high
m>>matrix^4^3.low
```

Note that the order of subscripts in the matrix structure reference is the reverse of the subscripts in the declaration.

SUBSCRIPTED STRUCTURE RULE: The replicated field or substructure must begin on a word boundary and be a multiple of 16 bits wide, or begin on a byte boundary and be 8 bits wide. Subfields within a replicated substructure need not satisfy this restriction; it applies only to the size and position of the full replicated element. For example,

```
f^1,10 [ a bit 3 ; b bit 13 ]
```

and

```
[ a bit 3 ; b bit 5 ]^1,10
```

are both legal; but

```
a^1,10 bit 3
```

and

```
b^1,10 bit 13
```

are not.

6-4 . . . . . Overlays

It is often the case that a portion of a structure must be referenced with different sets of fields at different times; therefore the compiler allows parallel field lists to be declared. For example, the following structure is a description of the Nova instruction format:

```
structure instr :    [ logical bit 1
                       [ acs  bit 2 ; acd bit 2
                         func bit 3
                         shft bit 2 ; cry bit 2
                         nlod bit 1 ; skp bit 3
                       ]
                      =[ op bit 4
                         i  bit 1
                         x  bit 2
                         d  bit 8
                       ]
                     ]
```

The bracketed field lists joined by "=" refer to the same portion of the structure (bits 1 to 15). If p points to an instruction, the expression p>>instr.logical references bit 0 of the instruction. On the Nova, this bit distinguishes between arithmetic/logical instructions and memory-reference instructions; a program would use this bit to determine whether it is appropriate to reference p>>instr.acs, etc. or p>>instr.op, etc.

Parallel substructures need not be of equal length; the position of subsequent fields is determined by the longest of the overlaid substructures.

6-5 . . . . . Left-lump structure references

The operator ">>" uses the value of its left-hand operand as the address of the data to be referenced. There is another structure reference operator, "<<" (pronounced "left-lump"), which takes a variable as its left-hand operand, and loads data from or stores data into the variable itself, rather than treating the variable as a pointer. To illustrate, suppose we have defined

```
structure [ lh byte ; rh byte ]
```

and that the value of the variable p is #001003. The statement

```
q = p>>rh
```

stores into q the right-hand 8 bits of the number contained in memory location #1003; it is equivalent to

```
q = p!0 & #377
```

The statement

```
q = p<<rh
```

stores into q the value #000003, which is the right half of the value of p; it is equivalent to

```
q = p & #377
```

Similarly, the statement

```
p>>rh = q
```

is equivalent to

```
p!0 = (p!0 & #177400) + (q & #377)
```

which stores a value into the right half of location #1003. The statement

```
p<<rh = q
```

is equivalent to

```
p = (p & #177400) + (q & #377)
```

which stores into the right half of the variable p.

The "<<" operator should normally be used only with structures that are one word wide. The compiler will interpret a statement like

```
p<<area.width = w
```

(a reference to the third word of a structure) to mean

```
(lv p)>>area.width = w
```

This will store into the location which is two words below the place in memory where p happens to be allocated. It is dangerous to assume anything about the allocation of BCPL variables, except in special cases such as consecutively declared dynamic variables, so use this feature with care.

The left-hand operand of a "<<" expression may be a vector-subscript expression or an rv-expression, instead of a variable name. The statement

```
v!i<<area.width = w
```

means

```
(lv v!i)>>area.width = w      ,or,equivalently,      (v+i)>>area.width
```

and

```
(@p)<<area.width = w
```

means

```
p>>area.width = w
```

(Note where parentheses are needed in the above expressions).

6-6 . . . . . Heffalump structure references

The operator "=>" (pronounced "heffalump") is convenient for referencing structures that are accessed indirectly. The expression

```
a=>s.x
```

is equivalent to the expression

```
(@a)>>s.x.
```

Here the variable a contains the address of a memory word (say, p) whose contents in turn address a block of data that the structure s describes. The information in this block may be freely relocated, provided one also changes p to indicate the new location. Any variable, a, containing the address of p will still be able to access the data using the heffalump construct.

6-7 . . . . . Other structure operators

The "lv" operator may take a structure reference expression (">>" or "<<" expression) as its operand. Its value is the address of the memory word which would be referenced by the structure expression. The field referenced need not be a full-word field.

It is sometimes necessary to determine the location or width of a field in a structure. Two special operators are provided for this: "size" and "offset". Both are unary operators which take a field specification as an operand (that is, a construct that can appear to the right of ">>" or"<<". The value of a "size" expression is the size, in BITS, of the specified field. For example:

```
size area.width          (value is 8)
size area                (value is 48)
size string.char^i       (value is 8)
size string.char         (value is 2040)
```

A "size" expression is always a compile-time constant, even if a variable subscript expression is involved. Note that if a subscript is missing in the field specification, the size of the entire replication is returned.

The value of an "offset" expression is the BIT number, counting from bit 0 at the beginning of the structure, of the first bit of the specified field. For example:

```
offset area.width        (value is 32)
offset area              (value is 0)
offset string.char^5     (value is 40)
offset string.char^i     (value is 8*i)
offset string.char       (value is 8)
```

An "offset" expression is a constant unless a variable subscript expression is involved.

Keep in mind that "size" and "offset" return values in BITS, not in words. To get a vector for an area block, for example, you must say

```
let cursor = vec (size area) / 16
```

6-8 . . . . . Syntax of structure declarations

```
STRUCTDECL      structure STRUCTGROUP

STRUCTGROUP     STRUCTITEM
                STRUCTITEM = STRUCTITEM = ... = STRUCTITEM

STRUCTITEM      NAME : FIELDDESCR
                NAME ^ SUBSCR : FIELDDESCR
                blank : FIELDDESCR
```

	STRUCTLIST STRUCTLIST ^ SUBSCR
STRUCTLIST	[STRUCTITEM ; STRUCTITEM ; ... ; STRUCTITEM]
FIELDDESCR	bit bit CONST byte byte CONST word word CONST STRUCTLIST STRUCTLIST ^ SUBSCR
SUBSCR	CONST , CONST SUBSCR ^ CONST , CONST

The colons in STRUCTITEM are really only necessary if a carriage return precedes a STRUCTLIST; in other places they may be omitted. The semicolons separating STRUCTITEMs in a STRUCTLIST may be omitted if a carriage return separates the STRUCTITEMs.

SECTION 7
SOURCE FILE CONVENTIONS

7-1 . . . . Declaration files

The word "get" followed by a file name enclosed in quotes ("...") causes the file to be included in the compilation, as if the contents of the file appeared in the source text. The most common use of "get" files is to include a common set of manifest, external, and structure declarations in a number of source files that will be loaded together. The compiler will ignore a second "get" on a "get" file that it has already read (this facilitates certain uses of the precompilation feature; see description of the /G compiler switch).

7-2 . . . . Labeled brackets

Brackets may be labeled with a sequence of letters and digits immediately following the "[" or "]". When a labeled "]" is seen by the compiler, each unmatched "[" (whether it is labeled or not) is implicitly matched until the "[" with the same label is matched. Thus, in:

```
if n gr 0 do [1 i = 1
        until i gr n do
                [2 x!i = 0; i = i + 1 ]1
```

the "]1" closes both compound statements. Note that a carriage return, space, or tab must be present between an unlabeled "[" and a statement that starts with a name. Usually some error will be detected quickly if no space is left (as in "if n gr 0 do [x = 0 ... "). But sometimes the resulting statement will be legal (as in "if n gr 0 do [rv x = 0 ..."). In such cases, the error may not be detected until the end of the source text; this is often the cause of a non-obvious "unmatched section bracket" syntax error.

7-3 . . . . Semicolon insertion

If two statements are separated by a carriage return, a semicolon is not required between them. This is accomplished by having the lexical analyzer replace a carriage return by a semicolon if it is preceded by a symbol which might end a statement and followed by a symbol which might begin a statement. Carriage returns are ignored (treated as spaces) in other places. This implies that a BCPL statement may extend over two or more lines, with the carriage returns occurring anywhere in the statement except before a "+" or "-", or before the "(" which begins a function argument list. So

```
x = a -
  (b*c)
```

will be interpreted properly (no semicolon inserted), but

```
x = a
 - (b*c)
```

and

```
x = a-f
  (b,c)
```

will give a parsing error, because semicolons will be inserted at the carriage returns ("+", "-", and "(" might begin a statement).

Semicolons will also be inserted at carriage returns in external, manifest, static and structure declarations, and in the constant list of table expressions.

Carriage returns may no appear in string constants. To include a carriage return, use *N or *C.

7-4 . . . . . Do/Then insertion

The words "do" and "then" are equivalent; so one may write

```
if x ls 0 then x=-x
```

or

```
if x ls 0 do x=-x
```

The "do" (or "then") in an "if," "unless," "while," "until," or "for" statement may be omitted if the symbol which would follow the "do" is one of the following

if	for	break
unless	switchon	loop
while	goto	finish
until	return	abort
test	resultis	endcase

Thus one may write:

```
if x eq 0 resultis -1
while x ls 0 goto L
unless x gr 0 break
for i=1 to 10 switchon v!i into [ ... ]
```

7-5 . . . . . Comments

Comments may appear anywhere in the source text, and begin with a pair of slashes (//). The slashes and the remainder of the line on which they lie are ignored.

7-6 . . . . . Upper case vs. Lower Case

Source files may be upper-case only, or upper- and lower- case. If lower-case is used, reserved words must be lower-case. The basic rules for case are as follows:

If the first word of the source program (i.e., of the file named in the command line) consists of all lower-case characters, the compiler will distinguish words on the basis of case; and reserved words must be typed in lower-case.

If the first word is not entirely lower-case, the compiler will, in effect, convert everything to upper-case on input. The global switch /U will also cause input to be converted, even if the first word is in lower-case.

This rule has implications for both compiling and loading. For compilation:

1. If your program is entirely upper-case, any "get" files specified in the program will be treated as upper-case files, even if they were prepared in lower-case. So an upper-case program can use a file of declarations (e.g., IOX for the IO package), as long as that declaration file does not depend on case to distinguish between names.

2. If your program wants to distinguish names on the basis of case, reserved words must be typed in lower case, both in your program and in any "get" files which the program needs. So in order to use a declaration file which was prepared in upper case, you must either use the /U switch (if you don"t care about case) or change the declaration file's reserved words to lower-case (if you do care about case in your program).

The BCPL loader (BLDR) normally distinguishes external names on the basis of case. So if you want to load upper-case and lower-case .BR files together, you must use the /U global switch on BLDR (or, alternatively, recompile the lower-case programs with /U). In particular, you must use BLDR/U if you load the IO package (IO1.BR, IO2.BR) with upper-case programs, or recompile the source files (IO1, IO2) with BCPL/U.

SECTION 8
COMPILATION

8-1 . . . . . Normal compilation

The BCPL compiler consists of six files, normally called BCPL.SV, BCPL.YL, BCPL.YC, BCPL.YS, BCPL.YT, and BCPL.YG. The .SV file is the main program; the .Y* files contain the code for the five passes of the compiler. The .Y* files must have the same name as the save file and the given extensions; so to rename the compiler, you must rename the .Y* files as well as the .SV file.

Normally, to compile a source file (e.g., QUEENS.3), just type

BCPL QUEENS.3

(Only one source file may be compiled at a time.) (No extension is automatically assumed for the source file name.) The compiler will print

BCPL 2.0 -- QUEENS.BR = QUEENS.3

and begin compiling the program. (2.0 is the current version of the compiler.) If no errors are detected, the BCPL relocatable binary file QUEENS.BR will be created, and the compiler will print something like

QUEENS.BR -- l426 (790) WORDS

The numbers are the length of the code generated in octal (decimal).

If an error is detected in the source text, the compiler will generally print each offending line and indicate the error(s) found in that line. The compiler will continue to look for further errors as long as it can do so without getting confused, and finally print the message

n ERRORS IN QUEENS.3

Some errors are grounds for immediate termination of compilation. The most common ones are trying to compile a source file that does not exist, or typing a command line that BCPL does not understand. Suitable messages are printed to indicate such errors. It is also possible to have a program which is "too big", in one respect or another, for BCPL to handle. This usually results in a message like "FRAME SPACE OVERFLOW" or "OUT OF FRAME SPACE". You must split the program into separately compilable files when this happens.

The compiler normally assumes that the Nova console is a CRT terminal. Therefore, after producing 20 lines of terminal output, it rings the bell (if any), prints a colon, and waits for the user to type a carriage-return or line-feed before proceeding. Carriage-return produces 20 more lines; line-feed produces one more line; 0 followed by carriage-return or line-feed causes the compiler to proceed without further pauses.

8-2 . . . . . Global switches

These switches can be attached to the name BCPL (or a whatever you call your compiler); e.g., "BCPL/U/A QUEENS.3".

/U Treat the source file as if it had been typed entirely in upper case. (See the section on upper/lower case considerations.)

/P Turn off the "pause" feature described above.

/F Write error messages onto the file QUEENS.BT (if the source file name was QUEENS.3) instead of printing them on the terminal. If /F is given, the compiler prints the message

BCPL 2.0 -- QUEENS.BR,QUEENS.BT = QUEENS.3

at the beginning of compilation.

/A Produce an assembly-language listing of the code generated. (This is useful if you want to see what kind of code BCPL generates, or if you are having a hard time debugging a particular piece of code. But the listing file is big -- it takes a long time to generate and print -- so you probably don"t want to make a habit of requesting it.) The listing is written on the file QUEENS.BT, unless the /T switch is given; error messages still appear on the terminal, unless /F is given.

/T Causes all output (error messages and the /A listing, if requested) to appear on the terminal. The file QUEENS.BT is not created.

Summary: /F alone sends error messages to QUEENS.BT. /A/F sends both errors and the assembly listing to QUEENS.BT; /A/T sends both to the terminal. /A alone sends errors to the terminal, and the assembly listing to QUEENS.BT. /F/T is illegal; /T alone has no effect.

/D Causes the compiler to indicate when it starts a new compilation phase (LEX, CAE, SAE, TRN, and NCG), and prints debugging information with error messages.

/H Causes the compiler to pause (by entering the Nova debugger) between compilation phases and after error messages. To resume, type (ESC)R, not (ESC)P.

(/D and /H are generally useful only to compiler gurus.)

/G This switch is used to generate "precompiled" declarations files. Any source file (which may contain "get" statements) may be precompiled, using the /G global switch. For example,

BCPL/G DECLDRIVER

will precompile DECLDRIVER and create the files DECLDRIVER.BD and DECLDRIVER.BC. DECLDRIVER is typically just a list of "get" statements, consolidating declaration files. Subsequently, the precompiled declarations may be used with the local /G switch (see below); precompiling increases the speed of the compiler slightly if the same declarations are to be included in many files.

/S See the local /S switch, below. The global version simply provides a site-dependent default value for the switch argument.

8-3 . . . . . Local switches

These switches are attached to names following the compiler name in the command line; e.g., "BCPL QUEENS.3 QUEENS.LS/A":

name (no switches) The name is taken as the source file name. No extension is assumed; you must type "name.ext" if the source file has an extension. The source file name is used to generate the names for the relocatable binary (.BR) file and the text output (.BT) file (unless these are specified by the local switches /A, /F, /R). On the Nova, if a device is specified with the name (e.g., DPl:QUEENS.3), that device will be used for files specified in "get" directives in the source text; and for the output files (unless these are specified by the local switches /A, /F, /R). If no device is specified, the default device is used (the device given in the last DIR command to DOS), even if the compiler is running on a different device (e.g., if you have typed "DIR DP0; DPl:BCPL QUEENS...", QUEENS and its "get" files will come from DP0). There are no "devices" on the Alto.

name/A Like the global /A switch, but the assembly listing is written onto "name" rather than QUEENS.BT. If "name" is a file name, the extension .BT will be appended to it if it has no extension; to create a file with no extension, use "name./A". If "name" is a device (e.g., MC0:XGP.), it should be terminated with a "."; the output will be sent to the device named.

name/F Like the global /F switch, but writes error messages onto "name" as for /A above. ("name/A/F" does the obvious thing, but you cannot send errors and the assembly listing to two different files.)

name/R Causes the relocatable binary file to be named "name" instead of QUEENS.BR. The .BR extension is appended to "name" if it has no extension; to create a file with no extension, use "name./R".

name/G The named file is a file of precompiled definitions, created with the global /G switch (see above). For example, the command

```
BCPL DECLDRIVER/G TEST
```

will compile test, including the declarations precompiled in DECLDRIVER.

number/V The decimal number is used to set the "manifest constant" for use with the /M switch, below.

name/M This switch declares the name to be a manifest constant, with the value taken from the last setting of the /V switch (default is true, -1). The value will apply throughout compilation, excluding any part of the compilation introduced through the precompilation (/G) option.

If used in conjuction with "newname," this can be used to override standard settings for parameters.

Caution: Nova DOS will convert all keyboard input to upper case; names given to the /M switch in this manner will therefore be upper case. However, the /M switch does not trigger the "upper case" detector (section 7-6).

name/L

name/T These switches cause the compiler to print the source text (/L) and intermediate compilation results (/T) as it proceeds through its various phases. The phases are specified by the individual characters of "name":

L for the lexical analyzer
C for the parser
S for the symbol table generator
T for the Ocode generator
l for the code generator, pass l
2 for the code generator, pass 2.

E.g., "C1/L" would cause the compiler to print each line of source text as it parses it, and again as it makes a first pass at generating code for the line. The output would go to the file QUEENS.BT unless the global /T switch were given. These switches are primarily for debugging the compiler. But they might be helpful occasionally in tracking down an obscure error, or one for which the error message does not provide enough context to locate the offending statement in the source text.

number/S The number is interpreted in octal. Its value is used instead of the first instruction of code normally issued for each procedure (see the runtime environment section). The same number, incremented by #400, is used instead of the standard procedure return instruction. This facility allows an installation to customize its procedure storage allocation facilities.

SECTION 9
LOADING

9-1 . . . . . Normal loading

The BCPL loader on the Alto is found on the file BLDR.RUN. A symbol file BLDR.SYMS also exists for use in loader maintenance.

The BCPL loader on the Nova consists of four files, normally called BLDR.SV, BLDR.YU, BLDR.YI, and BLDR.YD. The .Y* files are copies of files that the loader needs for initialization of the save file which it creates. The .Y* files must have the same name as the loader; so if you rename BLDR.SV, you must rename the .Y* files as well.

A typical command to BLDR on the Alto looks like:

```
BLDR/L/V QUEENS QUEENS1
```

and on the Nova looks like:

```
BLDR/D/L/V QUEENS QUEENS1 IO1 IO2
```

This would create the file QUEENS.RUN (.SV on the Nova), an executable save file, from the BCPL relocatable binary files QUEENS.BR, etc. The /L/V switches create a symbol table file named QUEENS.BS, containing information about where things will be in core when the program runs. A typical .BS file listing is attached. The /D switch on the Nova loads the debugger.

BLDR will accept concatenated .BR files as well as .BR files created directly by the compiler. That is, if F1.BR, F2.BR, ..., Fn.BR are all BCPL relocatable binary files, and F.BR is their concatenation, then including F in a BLDR command has the same effect as including F1 F2 ... Fn. The purpose of this feature is to allow multi-file subroutine packages of BCPL routines to be distributed as one file rather than as a collection of files.

9-2 . . . . . Errors

Errors in the command line to BLDR are fatal; the loader immediately aborts. Most such errors will result in a message like

```
Bad switch L in QUEENS/L/S
```

Undefined file names, and other operating system-detected errors will result in something like

```
Cannot open QUEENS.BR
```

Fatal error messages are always printed on the terminal.

The loader detects two types of external name conflicts. If an external name is defined (by "static

[name = ...]" or by "let name (...) be ...") in more than one relocatable binary file, the loader generates a message like

QUEENS2.BR

The EXTERNAL NAME name was also defined in QUEENS1.BR

for each such conflict detected in QUEENS2. On the Alto, the static for "name" will contain the first value given it. If an external name is declared to be a common (page zero) variable in some files (by "external [@name; ...]") but not in the first file in which the name appears, the loader genrates a message like

QUEENS2.BR

The COMMON NAME name was not declared COMMON in QUEENS1.BR

These messages appear in the .BS file if one is being created; the message

n errors during loading

is printed on the terminal if any name conflicts are detected. You must recompile the offending files and reload before attempting to run the program.

External names which have been used but not defined result in the message

n undefined externals

being printed on the terminal. The names are listed in the .BS file if one is being created; or on the terminal otherwise.

The loader also generates "warnings" if it detects space allocation conflicts in the save file being created. The most common of these are

Not enough COMMON space

if too many common (page zero) variables have been declared, and

Not enough STATIC space was reserved

if too many non-page-zero statics have been used. The available page zero space cannot be increased; you must redefine some common variables to be ordinary statics. The space reserved for statics can be specified with the local /W switch; see below for this and for other space allocation controls.

The error, warning, and undefined/multiple-definition error counts are separate; if you are told that was one undefined external and one error, there are two things wrong. The error being reported is not the undefined external but a different one.

9-3 . . . . . Global switches

/D (Nova only) Load the Nova debugger into the save file. This switch is legal only if no assembly language file is specified with the /I switch; if you load assembly language programs, you should include the debugger when you load them with DOS's RLDR. This switch is not needed on the Alto, since debugging is done with Swat.

/U Convert the names of all external symbols to upper case. This is needed, for example, if

you are loading the DOS IO package (IO1, IO2) with programs written in upper case; the IO procedure names in your files are upper case, but in IO1 and IO2 they are defined in lower case. Without /U, the upper case externals in your programs would be undefined. (Alternatively, you could recompile the IO package source files with BCPL/U.)

/W — Do not print warning messages. Normally the loader will tell you if you do something suspicious, like loading a program on top of something else. If you know what you are doing, and if the warning messages bother you, you can turn them off with /W.

/L/V/N — Generate lists of static variable names. /L prints procedure and label names, sorted by the location of the procedure or label in the code; the /L listing is, in effect, a core map. /V prints non-procedure names (variables). /N prints all static names, sorted by address. The most useful combination is /L/V; it lists all statics, separating procedure names from variable names. The listings go to the file "savefilename.BS" unless the /T switch is used.

/T — All printed loader output (errors, warnings, and listings) is sent to the terminal. Normally, if listings are requested, they are sent to a file. Error and warning messages, and other load map data if there are no listings, normally go to the terminal.

/F — All printed output is sent to the file "savefilename.BS", except for fatal error messages, which always go to the terminal.

/M — (Alto only) Don’t produce a .SYMS file.

/K — (Alto only) Don’t read SYS.BK. (The facilities of the Alto operating system are made accessible to user programs via static variables that refer to system procedures or system scalars. Because these objects are not defined in a user’s Bcpl program, he must declare the names to be external. The loader automatically reads the file Sys.bk to determine how to match up the user’s references with the operating system objects. This arrangement does not require re-loading programs when objects in the operating system move. The K switch should only be used if you do not want the loader to perform this service for you, e.g., if you are loading the operating system itself.)

/R — (Alto only) Don’t complain if the same BR file name appears more than once in the file list (presumably in different overlays). Load the code each time it appears, but only allocate the statics once. Each such static, like any multiply defined static, will contain the first value assigned to it. This is relevant only if at least one of the occurrences of the BR file is in resident (non-overlay) code.

/B — (Alto only) Append overlay files to the RUN file instead of creating separate BB files. Each overlay will start on a new disk page.

/I — (Alto only) Initialize all code-pointing statics defined in Type B overlays to point to the procedure SwappedOut, which had better be defined in the resident code.

9-4 . . . . . Local switches -- group 1

These switches provide global information to the loader. All occurrences of these switches must appear before any of the group 2 switches, and before the first relocatable binary file name.

name/S — The name of the save file to be created. (If not specified, the name of the first relocatable binary file is used.) If "name" has no extension, .RUN is used (.SV on the

Nova). The "name" will also be used for the name of the .BS file unless the local /F switch is used, and on the Alto for the .SYMS file, unless the /M switch is used.

name/F All output is sent to the file "name". If "name" has no extension, .BS is used.

name/I (Nova only) Assembly language file. The file "name" (extension .SV if "name" has none) is assumed to be a Nova save file. The save file created by BLDR is initialized to the contents of this file (except for locations 300-377) at the beginning of loading. If the Nova debugger is to be loaded, it must have been loaded with the /I file. If no /I file is specified, a blank save file (BLDR.YI) is used, or if the global switch /D is specified,

name/U (Nova only) BCPL runtime routines. This switch allows the user to replace the standard runtime routines (get new frame, multiply, etc.) with his own. (These normally come from BLDR.YU.) The specified file is a Nova save file, but it is special in several respects.

number/N Maximum number of names allowed (octal). The default is 1000 (512 decimal). BLDR must allocate a certain amount of fixed space for each name, and must also have room for the name strings themselves. If you have a large number of long names, BLDR may run out of room, and print a storage exhausted message; or you may have more than 512 names. In either case, you may be able to load by adjusting the number of names allowed with /N. You may also be able to get more room with /C, if none of your .BR files have as much as 5000 words of code. (The /N switch does not affect the default /W value - see below).

number/C Maximum (octal) size of code in a single .BR file. The default is 5000. The /C switch is useful either if you have an especially big .BR file, or if you need more name space (see /N). (The compiler message "QUEENS.BR -- 1426 (790) WORDS" indicates the size of the code compiled, in octal and decimal).

number/Z The (octal) starting address for allocating common (page zero static variables). If not specified, common starts at octal 50 on the Alto, and on the Nova at ZMAX of the /I file, which is 60 if global /D is specified, 50 otherwise.

number/V The (octal) starting address for allocating static variables. If not specified, statics start on the Alto at octal 1000, and on the Nova just after the BCPL runtime routines (which are loaded just after the /I file).

number/W The maximum number (octal) of non-page-zero static variables. The default is 400 (256 decimal). If no /V is specified, this amount of space is reserved in the save file at the default starting address for statics; code will be loaded after this space unless /O is given on the Alto, or /P is given on the Nova. If the starting address for statics is specified with /V, it is the user's responsibility to see that enough space is left for static variables at that address; /W is then just used in checking that static and code space do not overlap.

number/J (Nova only) The maximum number (octal) of overlay files permitted. The default is 10 (8 decimal).

number/K (Nova only) The maximum number (octal) of .BR files which may be loaded.

name/M (Alto only) The first name of the SYMS file (defaults to the same name as the RUN file).

number/O (Alto only) The location to start loading code (instead of its usual place right after the statics).

9-5 . . . . . Local switches -- group 2

These switches control the loading of BCPL code into the save file. The loader also has facilities for creating "overlay" files to allow code to be swapped in dynamically; see the section on overlays below.

name (no switches) A BCPL relocatable binary file. If "name" has no extension, .BR is assumed (this is the extension normally used by the compiler). The code in the file is loaded into the save file at the current PC.

name/I The file "name.BR" is considered to be the beginning of a series of "initialization code" files which extends to the end of the resident or of the A-overlay code in which the name appears. A relocation table (see Overlays, below) will be appended after the code of the series. The table will contain a pair [static address, relative PC] for each code-pointing static defined since the last /I. The idea is that your program after initialization can set all the those statics to point to SwappedOut (see Global Switch /I).

number/P Set the current PC to "number" (octal).

$number/P Add "number" to the current PC. No spaces may appear between the "$" and the "number".

letter/Q

letter/X

letter/Y The "letter" is a single character A-Z. These switches associate the current PC with the letter so that the PC can later be restored with the form of /P described below. /Q uses the value of the current PC; /X uses the larger of the current PC and the value (if any) currently associated with the "letter"; /Y uses the smaller of the current PC and the current value of the "letter".

letter/P Set the current PC to the value last assigned to the "letter" by /Q, /X, or /Y. If no value has been assigned, an error is reported.

The final PC value, after all files have been loaded (not counting the overlays on the Alto), is taken as the address of the start of frame space when the program executes. (This value can be changed on the Nova with a final /P specification.) Execution will begin with the first procedure defined in the first relocatable binary file loaded. This procedure will be called with one argument, a 32 (decimal) word vector whose contents are:

word 0: The last value assigned to "A" by /Q, /X, or /Y.
. . .
word 25: The last value assigned to "Z" by /Q, /X, or /Y.
word 26: The address at which statics were loaded.
word 27: The address of the last static variable.
word 28: The address of the first procedure loaded.
word 29: The address (+1) of the last word of BCPL code loaded.
word 30: The final value of PC (frame space start on the Nova).
word 31: The highest memory address available on the Nova,
the location of the relocation table if /I was used on the Alto.

9-6 . . . . . Nova Save file image

The save file produced by BLDR on the Nova looks just like an ordinary Nova save file. The core image it produces is organized as follows (all numbers are octal):

0...15
: (Not part of a save file. Nova save files start with location 16; DOS considers locations 0-15 sacred. The addressess listed below are core addresses; subtract 16 (octal) if you are looking at the save file itself (e.g., with OEDIT).

16...277
: An image of these words from the /I file. Common variables will normally be allocated starting at ZMAX, the first page zero (.ZREL) location not used by the /I file; this can be changed by the /Z switch to BLDR.

300...377
: Reserved part of page zero (used by the BCPL runtime routines). You should refrain from clobbering these locations, unless you know what you are doing. Locations 340-377 are relocated by BLDR to point at various runtime routines.

400...777
: An image of these words from the /I file. DOS depends heavily on this page being correct, so users should not clobber it. BLDR fixes a few words in this page to make the save file look as if it was created by the Nova loader.

1000-NMAX-1
: An image of the rest of the /I file. NMAX /Is the first unused word of the /I file. If there is no /I file, NMAX will be approximately 4300 if /D was used (the debugger is about 3300 words long), 1000 otherwise.

NMAX...UMAX-1
: The BCPL runtime routines. These currently are about 700 words long.

UMAX...VMAX-1 (if /V was not used)
: Space for static variables, unless the starting address for statics was explicitly specified by /V. The size of the space reserved (VMAX-UMAX) is 400, unless changed with /W.

VMAX... (if /V was not used)
UMAX... (if /V was used)
: The default starting address for loading BCPL code. If the group 1 switch specifications are followed by just a list of file names, the BCPL code will be loaded sequentially starting here, unless the PC is changed with /P.

The format of an Alto save file is described in the Alto Operating System Reference Manual, section 4.9.

9-7 . . . . . Overlays

All occurrences of these switches must appear after all .BR file names which are to be loaded into the "resident" save file have been specified.

name/A
: Create the file "name" (extension .BB if "name" has no extension) and load the following relocatable binary files sequentially into that file. The code is intended to be read into core and run at the current value of PC; procedures and labels defined in the

files loaded into "name" will point at this area of core. The PC should not be changed (with /P) between the .BR files. The file "name" (or the subfile of the RUN file if Global /B was used) has the format:

word 0: value of PC at the first .BR file loaded
word 1: length of the code in words
word 2: 0 (this word is 1 for a /B file - see below)
word 3: L, the word at which the relocation table starts, if any
word 4: length of the file or subfile in words
word 5: page number of this disk page on the Alto, 0 on the Nova
word 6: 0
. . .
word 15: 0
word 16: (this is the first word of code)
. . .
(if there is a relocation table, see below)

N.B.: The first word of the code for each .BR file is the length of the code for that file; the second word is executable.

name/B Similar to /A, but in addition, the file "name" contains information about which procedure and label pointers must be fixed when the code is read into core. /B is used when the place at which the code will be executed is not known at load-time.

All code compiled by BCPL is self-relocating; that is, the code contains no absolute addresses which point at the code. The only words which must point into the code are the static variables which are defined as procedures and labels. Therefore, in order to dynamically relocate the code from one or more .BR files, all that is necessary is to initialize the procedure and label variables defined in the .BR files. This is the purpose of the relocation pair list at the end of a /B file.

word 0: value of PC at the first .BR file
word 1: length of code in words
word 2: 1 (to distinguish between /A and /B files)
word 3: L, the word at which the relocation table starts
word 4: length of the file or subfile in words
word 5: page number of this disk page on the Alto, 0 on the Nova
word 6: 0
. . .
word 15: 0
word 16: (this is the first word of code)
. . .
word L: number of relocation pairs N
word L+1: static address
word L+2: relative PC
. . .
word L+N*2-1; static address
word L+N*2: relative PC

When the code is read in at location P, each "static address" must be set to P+ "relative PC", so that the procedures and labels which reference the code will point to the correct places. The following procedure will do this on the Nova; it assumes the standard IO package and a routine to get a block of storage from someplace in core.

```
let swapin(filename) be
[       let channel=open(filename)
        let header=vec 15
        readseq(channel,header lshift 1,32)     //read 16 word header
        let length=header!1                     //length of code
        let codestart=getblock(length)          //get core for code
```

```
        readseq(channel,codestart lshift 1,length*2) //read code
        setpos(channel,header!3 lshift 1)       //get to relocation info
        let n=readbin(channel)                  //number of pairs
        for i=1 to n do
        [  let p=readbin(channel)               //static address to fix
           let codeaddr=readbin(channel)        //offset in code
           @p=codeaddr+codestart                //fix static variable
        ]
        close(channel)
]
```

It should be noted that string constants and label constants are part of the code BCPL compiles; the pointer to the constant block is recomputed each time the string or table expression is evaluated. So non-resident code must be careful about its use of strings and tables.

Although the relocation pair table is the actual authority for producing correct addresses in statics that reference overlay code, a better BS file listing will result if each name/B entry is followed by 0/P, to reset the PC value assigned during the load.

9-8 . . . . . Alto Operating System Linkage

To facilitate operating system linkage, two kinds of text files are accepted by BLDR: files specifying static locations (.BJ files) and files specifying static values (.BK files). The former are specified by filename/J or filename/H and the latter by filename/K. All the BJ files must precede the first BR and all the BK files must follow the last BR!!! Remember that the loader automatically reads SYS.BK at the very end, unless Global /K has been specified.

The format of a typical line in a BJ or a BK file is:

staticName octalNumber(s) codes

A BJ line is ignored unless the staticName is declared external in some BR. A BK line is ignored unless the staticName is declared external in some BR and is never defined in any BR or BJ. Thus, a BJ file specifies only the locations of operating system statics defined and/or referenced in the program, while the BK serves to initialize only operating system statics referenced but not defined in the program.

In a BJ file, the last octalNumber on each line specifies the location at which the loader should allocate the static staticName. In a BK file, the first octalNumber specifies the initial value of the staticName. The first-last rule is framed to allow simple construction of these text files by editing a BS file.

The recognized "codes" on each line of a BJ file are as follows (note: if a BJ file is cited as filename/H, all codes are ignored, and the default is invoked):

U=UND=UNDEF
: (default) The staticName must be defined in this load.

P, L, V
: Another load (the operating system) defines the staticName to be a procedure (P), label (L), or variable (V); it must not be defined here

R (with P or L)
: The static points to relocatable code

The codes on each line of a BK file are as follows:

P(default), L, V
: Another load (the operating system) defines the staticName to be a procedure (P), label (L), or variable (V)

R (with P or L)
The static points to relocatable code

Unrecognized codes are ignored.

To simplify the composition of the text files, there are "bases" which are added to each octalNumber. The bases are specified by individual lines of the form:

octalNumber

Comments may be included in a text file between / and carriage return.

The loader cannot initialize a static unless it is in the static area of memory. Thus, UND entries in a BJ file which place a code-pointing or initialized static outside the legal area result in a warning message.

The loader keeps track of the minimum and maximum locations in the static area that are mentioned in BJ files (including those statics unused in any BR), and avoids allocating statics in that region thereafter.

The way the loader informs the operating system of the linkages is by listing the addresses of all statics initialized by BK entries in a table appended to the resident code (after the relocation table, if /I is used) and recording the number of these statics in the file header. The operating system assumes that the values of those statics are really "indices" into a static area in the OS (in which order will not change) from which the contents of the designated OS statics are copied into the corresponding user program statics.

SECTION 10

RUNTIME ENVIRONMENT

10-1 . . . . . Procedure Frame Format

Whenever code compiled by BCPL is being executed, AC2 points to the first word of the frame for the procedure which owns the code. (AC2 is not changed by "goto," so one should not jump across procedure boundaries; no check is made for this either at compile time or run time.) While the procedure Q is running (i.e. after a call has been executed from the procedure P and Q's frame is initialized), the frame belonging to Q contains:

(AC2)+0:	address of P's frame
(AC2)+1:	(temp -- see below)
(AC2)+2:	(temp -- see below)
(AC2)+3:	(temp -- see below)
(AC2)+4,5,...	arguments passed to Q by P
.	dynamic variables for Q
.	dynamic temps needed by Q
.	vectors declared in Q

The frame belonging to P, the procedure that called Q, contains:

word 0:	address of the frame of P's caller
word 1:	address (-1) within P to which Q should return
word 2:	(address (+2) of the start of P)
word 3:	(temp used by P to pass arguments to Q)
word 4,5,...	arguments, dynamic variables, temps, vectors for P

The frames belonging to P's caller and earlier ancestors of P have the same format as P's frame. The only useful information contained in the frame of the procedure currently executing (Q) is word 0; the return address for Q is in P's frame, not in the current frame. Words 2 and 3 of P's frame need not be preserved by Q once Q's frame has been allocated. Words 1, 2 and 3 of Q's frame are available as temps for the BCPL runtime routines (and for users' machine-language procedures -- see below) while Q is running.

10-2 . . . . . Procedure Calls

Assume that Q is the currently executing procedure, and that Q is about to call the function R with two arguments: z=R(x,y). (Calls with more than two arguments will be described below.) The code in Q for this statement will look something like this (assuming x, y and z are directly addressable):

```
LDA 0,x         //put arg1 in AC0
LDA 1,y         //put arg2 in AC1
JSR @R          //call R (R points to first instruction)
2               //number of arguments passed
STA 0,z         //store result passed back in AC0
```

The JSR will transfer to the following code in R:

```
STA 3,1,2       //save return address (in Q's frame)
```

```
JSR @370          //set up R's frame
n                 //size of frame needed by R
JSR @367          //(not executed unless >3 arguments)
(first instruction in R's body)
```

The "getframe" routine, pointed to by location 370, does most of the work for entering a procedure. Its responsibilities are to set AC2 to point to a block of storage at least n words long for R's frame, to save the original contents of AC2 (Q's frame pointer) in word 0 of R's frame, and to store the two arguments passed to R in words 4 and 5 of R's new frame. (If there are more than three arguments, "getframe" executes the JSR @367 to store the additional arguments into R's frame; otherwise the JSR @367 is skipped.) The "getframe" routine returns, in AC0, the actual number of arguments passed to R. If R has declared a "numargs" variable, the first instruction in R stores AC0 into this variable.

After "getframe" is finished, the body of R is executed. R returns by executing JSR @366, with its result in AC0 if it is a function. This "return" routine must deallocate R's frame, restore Q's frame pointer to AC2, and return to Q at the location (+1) pointed to by word 1 of Q's frame.

For procedure calls which pass zero or one arguments, the above discussion applies as well; AC0 and/or AC1 are simply not loaded by Q, and are ignored by "getframe."

For procedure calls with exactly three arguments, AC0 and AC1 are loaded with the first two arguments as above, and the third argument is passed to R by Q in word 3 of Q's frame. In this case, in addition to the chores mentioned above, "getframe" copies this word to word 6 of R's new frame (word 6 is the location for putting the third argument). The code in Q for a call a=R(x,y,z) might look like:

```
LDA 0,x           //put arg1 in AC0
LDA 1,y           //put arg2 in AC1
LDA 3,z           //put arg3 in word 3 of
STA 3,3,2         //Q's frame
JSR @R            //call R
3                 //3 arguments to R
STA 0,a           //store result
```

(The code might be more complex that this if one or more of the arguments is not a simple variable.)

For procedure calls with N arguments (N>3), the calling sequence is more complicated. N+1 consecutive cells are reserved (as dynamic temps) in Q's frame, starting at word L of the frame. (L is not necessarily the same for every call.) Arguments 3 through N are stored by Q in cells L+3 through L+N of Q's frame; arguments 1 and 2 are loaded into AC0 and AC1; and the number L is stored in word 3 of Q's frame. (Words L, L+1 and L+2 in Q's frame are available as temps for "getframe.") So the code for a=R(z1,z2,z3,z4,z5) might look something like:

```
LDA 0,z3          //store args 3,4,5 in Q's frame
STA 0,L+3,2
LDA 0,z4
STA 0,L+4,2
LDA 0,z5
STA 0,L+5,2
LDA 0,KL          //KL contains the number L
STA 0,3,2         //pass offset of args to R
LDA 0,z1          //put args 1 and 2 in AC's
LDA 1,z2
JSR @R
5
STA 0,a
```

So for calls with more than three arguments, "getframe" must move arguments 3 through N from Q's frame into words 6 through 6+N-2 of the new frame for R. This is done by the "moveargs" routine (pointed to by location 367) after "getframe" has created the new frame. (The "moveargs" routine is used, rather than

having "getframe" itself move the arguments, for historical reasons. The "moveargs" routine, like "getframe," must return in AC0 the number of arguments passed to R.)

Nothing in the above description of procedure frames and procedure calls depends on where or how frame space is allocated by "getframe" and deallocated by "return." In addition, the code compiled by BCPL makes no assumptions about frame allocation; a BCPL procedure simply assumes that the standard four-instruction preface will set up its frame and that the standard return instruction will deallocate it and restore the state of the caller. By replacing the standard "getframe," "moveargs" and "return" routines (e.g., by changing locations 366, 367 and 370), the user can tailor frame allocation strategy to special needs.

10-3 . . . . . Frame Allocation on the Nova

The standard Nova BCPL "getframe" allocates frames on a stack which starts from the final PC value seen by BLDR and grows toward address #77777. When "getframe" allocates a new frame, it checks to see that the last word of the frame is not beyond the address contained in location 335; if it is, "getframe" prints a message indicating that the program has run out of frame space, and aborts execution. Location 335 is initialized to point at the highest memory address available (not used by DOS). Normally, all available memory is assumed to be devoted to frame space. However, by adjusting the contents of location 335, a program can reserve storage for itself (e.g., the statement @#335=@#335-#10000 reserves #10000 additional cells, starting at location @#335 (after the statement is executed)).

The page zero location 336 points to the location which will be the first word of the frame for the next procedure called. So when location 335 is adjusted, the program should check the contents of location 336 to see if the desired space is available: @#336 must be less than @#335.

SECTION 11

NOVA I/O and UTILITY ROUTINES

11-1 . . . . . Introduction

This section describes a number of routines which have been written to provide limited but useful runtime support for Nova BCPL programs. In many cases, the routines are very similar to the actual assembly-language DOS system call, or are obvious extensions of the DOS function. Routines have been written to do many I/O functions and a few string functions. Limited formatted I/O functions have been implemented using general string and integer conversion routines.

Before calling any of the I/O runtime routines, the routine initbcplio must be called to set up several global variables. The I/O errors are handled by the routine whose address is in syserror. This routine is normally ioerror, a routine which corrects some inadequacies of the DOS error-handling facility, and optionally prints procedure information. Input routines do not consider end of file to be an error and return this information through a byte count indicating how many bytes were actually read, or a special ASCII character. Errors may be captured by changing the routine in syserror to one of the user's routines or by setting syserrortrap to "false." If this is done, after an I/O routine is called, the location syserrorflag will be false if no error has occured, but otherwise will be true; syserrorvalue will have the error value from AC2 after the DOS system call. End of file will be shown as an error when this facility is used. For doing routine tasks, the default error routine will be adequate.

DOS strings are not compatible with BCPL strings. All the I/O routines accept BCPL strings and convert them to DOS strings when necessary, with the exception of readline and writeline (see description of those procedures).

The procedure descriptions will, in many cases carry a cross-reference note to the DOS manual of the form DOS:ch-pp. In general, all procedure arguments must be given; in a few specific cases, optional arguments are permitted -- these are indicated by brackets ([ ]). The DOS channel for an open file is an argument to many of the routines; it is always called "chno." When using routines in which the "chno" description is marked with an asterisk (*), if the value of "chno" given is -1, the system teletype will be used (via PCHAR and GCHAR DOS functions). Thus, for simple teletype I/O it is unnecessary to open a channel.

The routines are contained in the files IO1 and IO2. IOX is a file containing external definitions that can be included in a BCPL program with the "get" statement.

11-2 . . . . . Global Names

sysac
: The accumulators used for system calls to DOS. Not generally useful except inside the runtime routines.

syserrorflag
: If set after a system call, an error has occurred. This will be true independent of the state of syserrortrap. The value of the error will be in syserrorvalue until another error occurs.

syserrorvalue
: If syserror flag is set after a system call, this static contains the value of the error. This value is constant until another error occurs.

syserrortrap
If this static is set to true, the routine ioerror will print an appropriate error message and return to DOS CLI. If set to false, ioerror will simply return. If ioerror is called by the user program with a single parameter, ioerror is called by the user program with a single parameter, ioerror behaves as if syserrortrap were set to true. For more information see ioerror(syserrorvalue).

sysprintpc
If set to true, ioerror will print the addresses of the system procedure from the runtime I/O and the user procedure which caused the error. This is the variable which is set to true by initbcplio(2).

filenamelength
The maximum length of DOS filenames--manifest constant which may be used for allocating vectors to receive DOS file names.

11-3 . . . . . Procedures

nbytes = readcomcm(chno, string [, switches])

Purpose:	To read arguments and switches from the DOS command file, COM.CM
chno	DOS channel number, previously opened to file COM.CM
string	A BCPL vector for the name read from COM.CM (may be allocated with vec filenamelength).
switches	A 26 element boolean vector in which each element corresponds to the alphabetic character for the switch.
nbytes	The number of bytes actually read is returned.

initbcplio(mode)

Purpose:	To initialize various constants needed by the runtime I/O routines. Failure to invoke this routine will lead to system crashes at undefined times!
mode	1 - normal mode; error messages will be given normally. 2 - diagnostic mode; stack information will be printed if this mode is set. Mode 2 may also be invoked by setting sysprintpc to true.

char = readch(chno)

Purpose:	To read one 8 bit character from channel chno previously opened to a DOS file.
chno	* A DOS channel number 0-7.
char	The 8 bit character read from the channel.

writech(chno,char)

Purpose:	To write one 8 bit character from channel chno previously opened to a DOS file.
chno	* A DOS channel number 0-7.
char	The 8 bit character to be written.

rbytes = readseq(chno, bytepointer, nbytes) DOS:4-14

Purpose:	Read a number of bytes using the DOS .RDS command.
chno	A DOS channel number 0-7.
bytepointer	DOS byte pointer to the first byte involved in the transfer.
nbytes	Number of bytes to be read.
rbytes	Number of bytes actually read--must be used to detect end of file.

writeseq(chno, bytepointer, nbytes) DOS:4-18

Purpose:	Write a number of bytes using the DOS .WRS command.
chno	A DOS channel number 0-7.
bytepointer	DOS byte pointer to the first byte involved in the transfer.
nbytes	Number of bytes to be written.

nbytes = readline(chno, string[, true/false]) DOS:4-13

Purpose:	To read a string terminated by a carriage return from a DOS file.
chno	A DOS channel number 0-7.
string	A BCPL vector with enough space to receive the input string.
true/false	If true, the TRUE DOS readline function is executed. The .RDL function terminates on NULL as well as form feed, carriage return and end of file. One usually does not want to deal with this function. If false or absent, the NULL termination is removed.
nbytes	If 1, a terminator has been received. The last character in the string received is either carriage return or form feed (or NULL if the true .RDL) or carriage return followed by #377 if end of file.

writeline(chno, string) DOS:4-17

Purpose:	Write a string which MUST be terminated by a carriage return, null or form feed to the DOS channel previously opened. DOS interprets tabs, form feeds for certain devices.
chno	A DOS channel number 0-7.
string	A BCPL string or vector which must be terminated as specified for readline.

writestr(chno, string)

Purpose:	Write any BCPL string. A line feed is unconditionally issued following every carriage return character.
chno	* A DOS channel number 0-7.
string	A BCPL string or vector which must be terminated as specified above.

writezoct(chno, number)

Purpose:	Write a six digit unsigned octal number with leading zeroes.
chno	* A DOS channel number 0-7.
number	16 bit quantity.

writedec(chno, number[, space])

Purpose:	Write a signed decimal number with fixed or variable spacing.
chno	* A DOS channel number 0-7.
string	16 bit quantity.
space	Number of spaces to be used. If missing or zero, a variable number of spaces are used.

writeoct(chno, number[, space])

Purpose:	Write a signed octal number with fixed or variable spacing.
chno	* A DOS channel number 0-7.
number	16 btit quantity.
space	Number of spaces to be used. If missing or zero, a variable number of spaces are used.

writeform(chno, formatcode, data[, formatcode, data...])

Purpose:	Write a group of string or 16 bit data to the channel as specified by the formatcodes.
chno	* A DOS channel number 0-7.
formatcode	0 - data following is string data. 2-10 - data following is a 16 bit quantity to be displayed in that radix.

writevalue(chno, number, rdx[, space])

Purpose:	Write a 16 bit signed number in arbitrary radix (2-10) using fixed or variable spacing.
chno	* A DOS channel number 0-7.
number	A 16 bit signed quantity.
rdx	An arbitrary radix 2-10.
space	The number of spaces to be used. If the argument is missing or 0, a variable number of spaces will be used.

word = readbin(chno)

Purpose: Read a 16 bit quantity from the DOS channel. No end of file detection is provided except by capturing the error with syserrortrap.
chno A DOS channel number 0-7.
word A 16 bit quantity read from the file.

writebin(chno, word)
Purpose: Write a 16 bit quantity to the specified channel.
chno A DOS channel number 0-7.
word A 16 bit quantity to be written.

chno = open(name) DOS:4-10
Purpose: Open a DOS file to a channel selected by the runtime routines.
name Any BCPL string which is a legal DOS file name. Device specifier must be upper case, e.g., DP0--all other characters are translated to upper case.
chno A DOS channel number 0-7 returned specifying the channel number to be used.

chno = append(name) DOS:4-11
Purpose: Re-open a DOS file to a channel selected by the runtime routines. Writing will begin following the last character in the existing file.
name Any BCPL string which is a legal DOS file name. Device specifier must be upper case, e.g., DP0--all other characters are translated to upper case.
chno A DOS channel number 0-7 returned specifying the channel number to be used.

nbytes = curpos(chno)
Purpose: Return the current byte position of a DOS file.
chno A DOS channel 0-7.
nbytes Current byte pointer for the file.

setpos(chno, nbytes)
Purpose: Set the current byte position of a DOS file.
chno DOS channel 0-7.
nbytes Current byte pointer for the file.

curposdw(chno, doublewordvector)
Purpose: Return the current block and byte number of a DOS file in a BCPL vector to overcome the lack of double precision integers in BCPL.
chno A DOS channel 0-7.
doublewordvector A 2 word BCPL vector giving the block number in word 0 and the byte number in word 1.

setposdw(chno, doublewordvector)
Purpose: Set the current block and byte number of a DOS file in a BCPL vector to overcome the lack of double precision integers in BCPL.
chno A DOS channel 0-7.
doublewordvector A 2 word BCPL vector giving the block number in word 0 and the byte number in word 1.

createfile(name) DOS:4-6
Purpose: Create a DOS file.
name A legal DOS file name.

deletefile(name) DOS:4-7
Purpose: Create a DOS file.
name A legal DOS file name.

initdev(name) DOS:4-4
Purpose: Initialize a DOS device.
name A legal DOS device name.

directorydev(name) DOS:4-4

Purpose:	Change the default directory to the indicated device.
name	A legal DOS device name.

releasedev(name) DOS:4-5

Purpose:	Release a device.
name	A legal DOS device name.

renamefile(name,newname) DOS:4-7

Purpose:	Change the name of an existing DOS file.
name	A legal DOS file name.

close(chno) DOS:4-12

Purpose:	Close an I/O channel to further use until re-opened.
chno	A legal DOS channel number (0-7).

resetfiles() DOS:4-13

Purpose:	Close all I/O channels to further use until re-opened.

errvalue = systemcall(ac0, ac1, ac2, syscallname, err) DOS:4-1

Purpose:	Generate a DOS system call directly.
ac0	NOVA ac 0 to be passed as part of the system call.
ac1	NOVA ac 1.
ac2	NOVA ac 2.
syscallname	A name from the list of system calls contained in iox, generally, the DOS mnemonic preceded by "sys"--e.g., sysrdl for .RDL. These are manifest constants defined in IOX.
err	The BCPL procedure to be called in the event of an error return from the system call.
errvalue	The error value if an error occurs, otherwise -1. The error code is returned in global vector SYSAC!2 and in the global variables syserrorflag and syserrorvalue. If syserrorflag is set, syserrorvalue contains the value of the error. syserrorvalue will not be changed, but SYSAC!2 will be changed with every system call.

ioerror(syscallname, sysac) or (syserrorvalue)

Purpose:	Writes an error message to the teletype output device. Most messages are generated by DOS, but in a few cases, ioerror generates the correct message. If called with 2 parameters, the error value is taken from the vector specified by sysac and in some cases the name specified by sysac. If called with 1 parameter, the error value is taken to be the value of that parameter and if needed syserrorname will be used. If syserrortrap is set to false, this routine will simply return when called with TWO parameters. The routine is executed unconditionally if called with only one parameter.
syscallname	The DOS system call used to generate the error.
sysac	The system call accumulator vector.
syserrorvalue	The error value which may be given directly in lieu of the two above.

install(chno) DOS:4-5

Purpose:	Install a DOS on the default directory device.
chno	The DOS channel previously opened to the DOS to be installed.

chatr(chno, ac0) DOS:4-8

Purpose:	Change the attributes of a DOS file.
chno	A DOS channel previously opened to the file to be changed.
ac0	The value for ac0 as specified in the DOS manual for file attributes: R=#100000, S=#020000, P=#000002, W=#000001. WARNING: if #040000 (bit 1) is set and the file is permanent, it cannot be removed except by a full initialization of the directory!!!!!!!!

ac0 = getfileatr(chno) DOS:4-9

Purpose:	Returns the attributes of a DOS file.

chno A DOS channel previously opened to the file in question.
ac0 The word returned with meanings defined by the DOS manual.

incr = memavail() DOS:4-21
Purpose: Returns the amount of available memory for the user program.
incr The increment of available memory.

memincr(incr) DOS:4-21
Purpose: Change the amount of user available memory.
incr The increment of memory to be claimed.

dosexec(name, ac1) DOS:4-23
Purpose: Execute a DOS save file.
name The name of a DOS save file to be executed.
ac1 The value for ac1 as specified by the DOS manual. If missing, 0 will be used so that the current execution level is pushed to the disk and the next save file will be started at its normal starting address.

dosreturn() DOS:4-24
Purpose: Return control to DOS CLI.

dosereturn(ac2) DOS:4-24
Purpose: Return control to DOS giving an error code. The common error messages will be misprinted due to DOS assumptions about file names.
ac2 The error value to be returned.

dosbreak() DOS:4-25
Purpose: Create the file BREAK.SV. WARNING!!!!! All I/O channels must be closed with a resetfiles command if the file is to be re-executed.

word = strtovalue(string[, radix])
Purpose: Convert a signed string to a 16 bit integer in the specified radix.
string The BCPL string to be converted.
radix The radix of the conversion. If unspecified, 8 is assumed.
word A 16 bit word having the converted value.

valuetostr(word, string, radix[, space])
Purpose: Convert a 16 bit signed value to a signed string with no leading zeros having either fixed or variable spacing.
word The 16 bit value to be converted.
string A vector with enough space to hold the converted value. If fixed spacing is specified, overflow will cause more spaces to be used in this vector. The maximum number of spaces used depends on the radix and is 16 for radix 2, 6 for radices 8 and 10.
radix The conversion radix.
space The number of string spaces to be used. If zero or missing, variable space is assumed.

packstr(ustring, pstring)
Purpose: Change a BCPL string from unpacked format (one byte per word) to packed format (two bytes per word).
ustring A vector containing a BCPL unpacked string, one character per word, the first word specifying the length.
pstring A vector with enough room to receive the packed string.

unpackstr(pstring, ustring)
Purpose: Change a BCPL string from packed format (two bytes per word) to unpacked format (one byte per word).
pstring A BCPL string.

ustring — A vector with enough room for the BCPL unpacked string, one character per word, the first word specifying the length.

movestr(stringsrc, stringdest)
- Purpose: Move a BCPL string which may be in either packed or unpacked format.
- stringsrc: A BCPL string to be moved.
- stringdest: A vector with sufficient room to receive the source string.

byteptr = dostr(bcplstrig, dosstring)
- Purpose: Convert a BCPL string to a DOS string.
- bcplstring: A BCPL string to be converted.
- dosstring: A vector with sufficient space to receive the converted string. The only difference in the two formats is that DOS requires a null character at the end of many strings.
- byteptr: A DOS byte pointer to the first character of the DOS string.

word = lengthstr(string) Purpose: Return the length of a BCPL string.
- string: A BCPL string.
- word: The length of the string.

char = extractchar(string, index)
- Purpose: Extract a single character from a string at a specified index.
- string: A BCPL string.
- index: The index for the character. If out of range, garbage is returned.
- char: A 16 bit word containing the value of the character.

ans = extractstr(string1, string2, index, lengthstring1)
- Purpose: Extract string1 from string2 beginning at the specified index.
- string1: A vector of sufficient size to receive the extracted string.
- string2: The string from which the extraction is to be made.
- index: The beginning index for extraction; if the index goes out of the range of string2 at any time, the length of the extracted string will be adjusted accordingly.
- lengthstr1: The length of the string to be extracted.
- ans: The actual length of the extracted string.

lastbyteindex = imbedchar(char, string[, index])
- Purpose: Imbed a character into a vector containing a BCPL string. The existing character at that index is destroyed. If the index for the imbedded character is greater than the length of the string, the second string is filled with blanks up to the imbedded character. If no index is specified, the character will be appended.
- char: The character to be imbedded.
- string2: A vector or BCPL string in which the character is to be imbedded. If index extends the length of string2, string2 must be a vector large enough to hold the results.
- index: The index in string2 at which the character is to be imbedded.
- lastbyteindex: The last position of string2 which was modified.

lastbyteindex = imbedstr(string1, string2[, index])
- Purpose: Imbed string1 in string2. The existing sub-string at that index is destroyed. If the index for the imbedded string1 is greater than the length of the string2, string2 is filled with blanks up to the imbedded character. If no index is specified, string1 will be appended to string2.
- string1: The string to be imbedded.
- string2: A vector or BCPL string in which the first string is to be imbedded. If string1 extends the length of string2, string2 must be a vector large enough to hold the results.
- index: The index in string2 at which string1 is to be imbedded.
- lastbyteindex: The index of the last byte imbedded in string2.

index = searchstr(string1, string2[, startindex])

Purpose:	Search string1 for string2 at the specified starting index or at the start of string 1.
string1	The string to be searched.
string2	The string to be found.
startindex	The index in string1 at which to begin the search.
index	The index of the string if it is found; if not, then -1.

SECTION 12
APPENDICES

12-1 . . . . . BCPL Reserved Words

and	abort				
be	by	break	bit	byte	blank
case	compileif	compiletest			
default	do	docase			
eq	eqv	ext	endcase	external	
for	false	finish			
ge	gr	get	goto		
if	ifso	ifnot	into		
let	le	ls	lv	loop	
	logand	logor	lshift		
manifest					
ne	neg	nil	not	neqv	numargs
	newname				
or	offset				
rv	return	resultis	repeat	repeatwhile	
	rem	rshift	repeatuntil		
switchon	static	size	selecton	structure	
to	test	then	true	table	
unless	until				
vec	valof				
while	word	xor			

INDEX