The Cintpos Portable Operating System

by

Martin Richards

mr@cl.cam.ac.uk

http://www.cl.cam.ac.uk/~mr/

Computer Laboratory
University of Cambridge
September 15, 2005

Abstract

Development of the Tripos Portable Operating System was started in 1977 at Cambridge and the system was extensively used for many years for operating system and network research. It formed the basis of the operating system for the Commodore Amiga and was also used commercially by several companies. It is, indeed, still in use by at least one major manufacturer.

This manual describes a machine independent version of Tripos implemented using a slightly extended version of BCPL Cintcode. Its interpreter and interface with the host operating system are implemented in C and be easily modified and extended. It is designed for process control applications in which most devices are controlled using TCP/IP on a local network.

It provides a simple environment in which all Cintpos tasks run in the same address space and communicate with each other using packet based message passing. The Cintpos kernel is simple, effective and includes facilities to set breakpoints and perform single step execution of the cintcode translation of BCPL programs. The execution performance of Cintpos on modern processors is an order of magnitude faster than the native version of Tripos on the machines on which it originally ran.

This manual describes the internal structure of Cintpos, its command language and the standard commands. It is currently being developed to run under Linux but should in due course also run under any version of Microsoft Windows.

Keywords

Portable Operating Systems, Tasks, Threads, Coroutines, Tripos, BCPL, Cintcode.

Contents

Preface

History of Tripos and Cintpos.

The Cintpos System described in this manual is freely available from my home page on the Internet [Ric].

The implementation is reasonably machine independent and should run efficiently on many machines both now and in the indefinite future.

The topics covered by this manual are listed below.

- An overview of the Cintpos system.
- A summary of the BCPL
- Cintcode, the byte stream interpretive code used in this implementation.
- The Kernel data structure.
- A description of the Cintpos resident runtime library.
- The design and implementation of command language interpreter for the system.
- A description of the standalone cintcode debugger and the DEBUG task.
- The profiling and statistics gathering facilities offered by the system.

Chapter 1

The System Overview

A console session.

Chapter 2

The BCPL Language

A BCPL program is made up of one or more separately compiled sections, each consisting of a list of declarations that define the constants, static data and functions belonging to the section. Within functions it is possible to declare dynamic variables and vectors that exist only as long as they are required. The language is designed so that these dynamic quantities can be allocated space on a simple runtime stack. The addressing of these quantities is relative to the base of the stack frame belonging to the current function activation. For this to be efficient, dynamic vectors have sizes that are known at compile time. Functions may be called recursively and their arguments are called by value. The effect of call by reference can be achieved by passing pointers. Input and output and other system operations are provided by means of library functions.

The main syntactic components of BCPL are: expressions, commands, and declarations. These are described in the next few sections. In general, the purpose of an expression is to compute a value, while the purpose of a command is normally to change the value of one or more variables.

2.1 Lexical features

Some features in BCPL are implemented by the lexical analyser in the compiler. These include comments, `Get` directives and conditional compilation.

2.1.1 Comments

There are two form of comments. One starts with the symbol `//` and extends up to but not including the end-of-line character, and the other starts with the symbol `/*` and ends at a matching occurrence of `*/`. Comment brackets (`/*` and `*/` may be nested, and within such comments the lexical analyser is only looking

for `/*` and `*/`. Care is needed when commenting out fragments of program containing string constants. Comments are equivalent to white space and so may not occur in the middle of multi-character symbols such as identifiers or constants.

2.1.2 The `GET` Directive

A directive of the form `GET "filename"` is replaced by the contents of the named file. By convention, `GET` directives normally appear on separate lines. The header file is searched for first in the current directory, then in the directories specified by the `HDRPATH` shell variables.

2.1.3 Conditional Compilation

There is a simple mechanism, whose implementation takes fewer than 20 lines of code in the compiler's lexical analyser, that allow conditional skipping of lexical symbols. It uses directives of the following form:

`$$`*tag*
`$<`*tag*
`$>`*tag*

where *tag* is conditional compilation tag composed of letters, digits, dots and underlines. All tags are initially unset, but may be complemented using the `$$`*tag* directive. All the lexical tokens between `$<`*tag* and `$>`*tag* are skipped (treated as comments) unless the specified tag is set. The following example shows how this conditional compilation feature can be used.

```
$$Linux                     // Set the Linux conditional compilation tag

$<Linux                     // Include if the Linux tag is set
  $<WinNT $$WinNT $>WinNT   // Unset the WinNT tag if set
  writef("This was compiled for Linux")
$>Linux
$<WinNT                     // Include if the WinNT tag is set
  writef("This was compiled for Windows NT")
$>WinNT
```

2.1.4 Section Brackets

Historically BCPL used the symbols `$(` and `$)` to bracket commands and declarations. These symbols are called section brackets and are allowed to be followed by tags composed of letters, digits, dots and underlines. A tagged closing section

bracket is forced to match with its corresponding open section bracket by the automatic insertion of extra closing brackets as needed. Use of this mechanism is no longer recommended since it can lead to obscure programming errors. Recently BCPL has been extended to allow all untagged section brackets to be replaced by `{` and `}` as appropriate.

2.2 Expressions

Expressions are composed of names, constants and expression operators and may be grouped using parentheses. The precedence and associativity of the different expression constructs is given in Section 2.2.9. In the Cintcode implementation of BCPL all expressions yield values that are 32 bits long, but in some native code implementations this word length is 64 bits.

2.2.1 Names

Names are used to identify variables, functions, labels and manifest constants. Syntactically a name is any sequence of letters, digits, dots and underlines starting with a letter, except that the reserved words (such as `IF`, `WHILE`, `TABLE`) are not names.

2.2.2 Constants

Decimal numbers consist of a sequence of digits, while binary, octal or hexadecimal hexadecimal are represented, repectively, by `#b`, `#o` or `#x` followed by digits of the appropriate sort. The `o` may be omitted in octal numbers. Underlines may be inserted within numbers to improve their readability. The following are examples of valid numbers:

```
1234
1_234_456
#b_1011_1100_0110
#o377
#x_BC6
```

The constants `TRUE` and `FALSE` have values `-1` and `0`, respectively, which are the conventional BCPL representations of the two truth values. Whenever a boolean test is made, the value is compared with `FALSE` (`=0`).

A question mark (?) may be used as a constant with undefined value. It can be used in statements such as:

```
LET a, b, count = ?, ?, 0
sendpkt(P_notinuse, rdtask, ?, ?, Read, buf, size)
```

Character constants consist of a single character enclosed in single quotes ('). The character returns a value in the range 0 to 255 corresponding to its normal ASCII encoding.

Character (and string) constants may use the following escape sequences:

Escape	Replacement
*n	A newline (end-of-line) character.
*c	A carriage return character.
*p	A newpage (form-feed) character.
*s	A space character.
*b	A backspace character.
*t	A tab character.
*e	An escape character.
*"	"
*'	'
**	*
*x*hh*	The single character with number *hh* (two hexadecimal digits denoting an integer in the range [0,255]).
**ddd*	The single character with number *ddd* (three octal digits denoting an integer in the range [0,255]).
f..f	This sequence is ignored, where *f..f* stands for a sequence of one or more space, tab, newline and newpage characters.

A string constant consists of a sequence of zero or more characters enclosed within quotes ("). Both string and character constants use the same character escape mechanism described above. The value of a string is a pointer where the length and bytes of the string are packed. If s is a string then s%0 is its length and s%1 is its first character, see Section 2.2.6.

A static vector can be created using an expression of the following form: TABLE $K_0, \ldots, K_n$ where $K_0, \ldots, K_n$ are manifest constant expressions, see Section 2.2.10. The space for a static vector is allocated for the lifetime of the program and its elements are updatable.

2.2.3 Calls

The only difference between functions and routines is whether their calls return results. Functions calls are normally made in the context of an expression where a result is required, while routine calls are made in the context of a command where no result is required. If a function is called in the context of a command its result is thrown away, and if a routine is called in the context of an expression the result is undefined.

A call is syntactically an expression followed by a list of arguments enclosed in paretheses.

```
newline()
mk3(Mult, x, y)
writef("f(%n) = %n*n", i, f(i))
f(1,2,3)
(fntab!i)(p, @a)
```

The parentheses are required even if no arguments are given. The last example above illustrates a call in which the function is specified by an expression. Section 2.4.8 covers both procedure definition and procedure calls.

2.2.4 Method Calls

Method calls are designed to make an object oriented style of programming more convenient. They are syntactically similar to a function calls but uses a hash symbol (`#`) to separate the function specifier from its arguments. The expression:

$$E\texttt{\#(}E_1\texttt{,..,}E_n\texttt{)}$$

is defined to be equivalent to:

$$\texttt{(}E_1\texttt{!0!}E\texttt{)(}E_1\texttt{,..,}E_n\texttt{)}$$

Here, E_1 points to the fields of an object, with the convention that its zeroth field (E_1`!0`) is a pointer to the methods vector. Element E of this vector is applied to the given set of arguments. Normally, E is a manifest constant. An example program illustrating method calls can be found in `BCPL/bcplprogs/demos/objdemo.b` in the BCPL distribution system (see Chapter 10).

2.2.5 Prefixed Expression Operators

An expression of the form `!`E returns the contents of the memory word pointed to by the value of E.

An expression of the form `@`E returns a pointer to the word sized memory location specified by E. E can only be a variable name or an expression with leading operator `!`.

Expressions of the form: `+`E, `-`E, `ABS` E, `~`E and `NOT` E return the result of applying the given prefixed operator to the value of the expression E. The operator `+` returns the value unchanged, `-` returns the integer negation, `ABS` returns the absolute value, `~` and `NOT` return the bitwise complement of the value. By convention, `~` is used for bit patterns and `NOT` for truth values.

Expressions of the form: `SLCT` *len*`:`*shift*`:`*offset* pack the three constants *len*, *shift* and *offset* into a word. Such packed constants are used by the field selection operator `OF` described in the next section.

`SLCT` *shift*`:`*offset* means `SLCT 0:`*shift*`:`*offset*, and `SLCT` *offset* means `SLCT 0:0:`*offset*.

2.2.6 Infixed Expression Operators

An expression of the form E_1`!`E_2 evaluates E_1 and E_2 to yield respectively a pointer, p say, and an integer, n say. The value returned is the contents of the n^{th} word relative to p.

An expression of the form E_1`%`E_2 evaluates E_1 and E_2 to yield a pointer, p say, and an integer, n say. The expression returns a word sized result equal to the unsigned byte at position n relative to p.

An expression of the form K `OF` E accesses a field of consecutive bits in memory. K must be a manifest constant (see section 2.2.10) equal to `SLCT` *len*`:`*shift*`:`*offset* and E must yield a pointer, p say. The field is contained entirely in the word at position p+*offset*. It has a bit length of *len* and is *shift* bits from the right hand end of the word. A length of zero is interpreted as the longest length possible consistent with *shift* and the word length of the implementation. The operator `::` is a synonym of `OF`. Both may be used on right and left hand side of assignments statements but not as the operand of `@`. When used in a right hand context the selected field is shifted to the right hand end of the result with vacated positions, if any, filled with zeros. A shift to the left is performed when a field is updated. Suppose `p!3` holds the value `#x12345678`, then after the assignment:

```
(SLCT 12:8:3) OF p := 1 + (SLCT 8:20:3) OF p
```

the value of `p!3` is `#x12302478`.

An expression of the form E_1`<<`E_2 (or E_1`>>`E_2) evaluates E_1 and E_2 to yield a bit pattern, w say, and an integer, n say, and returns the result of shifting w to the left (or right) by n bit positions. Vacated positions are filled with zeroes. Negative shifts or those of more than the word length return 0.

Expressions of the form: E_1`*`E_2, E_1`/`E_2, E_1 `REM` E_2, E_1`+`E_2, E_1`-`E_2. E_1 `EQV` E_2 and E_1 `NEQV` E_2 return the result of applying the given operator to the two operands. The operators are, respectively, integer multiplication, integer division, remainder after integer division, integer addition, integer subtraction, bitwise equivalent and bitwise not equivalent (exclusive OR). `MOD` and `XOR` can be used as synonyms of `REM` and `NEQV`, respectively.

Expressions of the form: E_1`&`E_2 and E_1`|`E_2 return, respectively, the bitwise AND or OR of their operands unless the expression is being evaluated in a boolean context such as the condition in a while command, in which case the operands are tested from from left to right until the value of the condition is known.

An expression of the form: *E relop E relop ... relop E* where each *relop* is one of `=`, `~=`, `<=`, `>=`, `<` or `>` returns `TRUE` if all the individual relations are satisfied and `FALSE`, otherwise. The operands are evaluated from left to right, and evaluation stops as soon as the result can be determined. Operands may be evaluated more than once, so don't try `'0'<=rdch()<='9'`.

An expression of the form: E_1`->`E_2`,`E_3 first evaluates E_1 in a boolean context, and, if this yields `FALSE`, it returns the value of E_3, otherwise it returns the value of E_2.

2.2.7 Boolean Evaluation

Expressions that control the flow of execution in conditional constructs, such as if and while commands, are evaluated in a Boolean. This affects the treatment of the operators `NOT`, `&` and `|`. In a Boolean context, the operands of `&` and `|` are evaluated from left to right until the value of the condition is known, and `NOT` (or `~`) negates the condition.

2.2.8 `VALOF` Expressions

An expression of the form `VALOF` C, where C is a command, is evaluated by executing the command C. On encountering a command of the form `RESULTIS` E within C, execution terminates, returning the value of E as the result of the `VALOF` expression. Valof expressions are often used as the bodies of functions.

2.2.9 Expression Precedence

So that the separator semicolon (`;`) can be omitted at the end of any line, there is the restriction that infixed operators may not occur as the first token of a line. So, if the first token on a line is `!`, `+` or `-`, these must be regarded as prefixed operators.

The syntax of BCPL is specified by the diagrams in Appendix A, but a summany of the precendence of expression operators is given in table 2.1. The precedence values are in the range 0 to 9, with the higher values signifying greater binding power. The letters L and R denote the associativity of the operators. For instance, the dyadic operator `-` is left associative and so `a-b-c` is equivalent to `(v-i)-j`, while `b1->x,b2->y,z` is right associative and so is equivalent to `b1->x,(b2->y,z)`.

9 9L	Names, Literals, `?`, `TRUE`, `FALSE`, `(E)`, Function and method calls	
8L 7	`!  %  OF  ::` `!  @`	Dyadic Prefixed
6L 5	`*  /  REM MOD` `+  -  ABS`	Dyadic operators
4	`=  ~=  <=  >=  <  >`	Extended Relations
4L	`<<  >>`	

3
3L
2L
1L
	`~ NOT`
	`&`
	`|`
	`EQV NEQV XOR`
		Bitwise and Boolean operators

1R	`-> ,`	Conditional expression
0 0	`VALOF TABLE` `SLCT :`	Valof and Table expressions Field selector constant

Table 2.1: Operator precedence

Notice that these precedence values imply that

```
            ! f x  means  ! (f x)
            ! @ x  means  ! (@ x)
      ! v ! i ! j  means  ! ((v!i)!j)
      @ v ! i ! j  means  @ ((v!i)!j)
    x << 1+y >> 1  means  (x<<(1+y))>>1)
            ~ x!y  means  ~ (x!y)
            ~ x=y  means  ~ (x=y)
         NOT x=y   means  NOT (x=y)
b1-> x, b2 -> y,z  means  b1 -> x, (b2 -> y, z)
```

2.2.10 Manifest Constant Expressions

Manifest constant expressions can be evaluated at compile time. They may only consist of manifest constant names, numbers and character constants, `TRUE`, `FALSE`, `?`, the operators `REM`,`SLCT`, `*`, `/`, `+`, `-`, `ABS`, the relational operators, `<<`, `>>`, `NOT`, `~`, `&`, `|`, `EQV`, `NEQV`, and conditional expressions. Manifest expressions are used in `MANIFEST`, `GLOBAL` and `STATIC` declarations, the upper bound in vector declarations and the step length in `FOR` commands, and as the left hand operand of `::` and `OF`.

2.3 Commands

The primary purpose of commands is for updating variables, for input/output operations, and for controlling the flow of control.

2.3.1 Assignments

A command of the form L`:=`E udates the location specified by the expression L with the value of expression E. The following are some examples:

```
cg_x := 1000
v!i := x+1
!ptr := mk3(op, a, b)
str%k := ch
%strp := 'A'
```

Syntactically, L must be either a variable name or an expression whose leading operator is `!` or `%`. If it is a name, it must have been declared as a static or dynamic variable. If the name denotes a function, it is only updatable if the function has been declared to reside in the global vector. If L has leading operator

`!`, then its evaluation (given in Section 2.2.6) leads to a memory location which is the one that is updated by the assignment. If the `%` operator is used, the appropriate 8 bit location is updated by the least significant 8 bits of E.

A multiple assignment has the following form:

$$L_1\texttt{,}\ldots\texttt{,}L_n\ \texttt{:=}\ E_1\texttt{,}\ldots\texttt{,}E_n$$

This construct allows a single command to make several assignments without needing to be enclosed in section brackets. The assignments are done from left and is eqivalent to:

$$L_1\texttt{:=}E_1\ \texttt{;}\ldots\texttt{;}\ L_n\ \texttt{:=}\ E_n$$

2.3.2 Calls

Both function calls and method calls as described in sections 2.2.3 and 2.2.4 are allowed to be executed as commands. The only difference is that any results produced are thrown away.

2.3.3 Conditional Commands

The syntax of the three conditional commands is as follows:

`IF` E `DO` C_1
`UNLESS` E `DO` C_2
`TEST` E `THEN` C_1 `ELSE` C_2

where E denotes an expression and C_1 and C_2 denote commands. The symbols `DO` and `THEN` may be omitted whenever they are followed by a command keyword. To execute a conditional command, the expression E is first evaluated. If it yields a non zero value and C_1 is present then C_1 is executed. If it yields zero and C_2 is present, C_2 is executed.

2.3.4 Repetitive Commands

The syntax of the repetitive commands is as follows:

`WHILE` E `DO` C
`UNTIL` E `DO` C
C `REPEAT`

C `REPEATWHILE` E
C `REPEATUNTIL` E
`FOR` N = E_1 `TO` E_2 `DO` C
`FOR` N = E_1 `TO` E_2 `bY` K `DO` C

The symbol `DO` may be omitted whenever it is followed by a command keyword. The `WHILE` command repeatedly executes the command C as long as E is non-zero. The `UNTIL` command executes C until E is zero. The `REPEAT` command executes C indefinitely. The `REPEATWHILE` and `REPEATUNTIL` commands first execute C then behave like `WHILE` E `DO` C or `UNTIL` E `DO` C, respectively.

The `FOR` command first initialises its control variable (N) to the value of E_1, and evaluates the end limit E_2. Until N moves beyond the end limit, the command C is executed and N increment by the step length given by K which must be a manifest constant expression (see Section 2.2.10). If `BY` K is omitted `BY 1` is assumed. A `FOR` command starts a new dynamic scope and the control variable N is allocated a location within this new scope, as are all other dynamic variables and vectors within the `FOR` command.

2.3.5 `SWITCHON` command

A `SWITCHON` command has the following form:

`SWITCHON` E `INTO` { C_1 ; ... ; C_n }

where the commands C_1 to C_n may have labels of the form `DEFAULT:` or `CASE` K. E is evaluated and then a jump is made to the place in the body labelled by the matching `CASE` label. If no `CASE` label with the required value exists, then control goes to the `DEFAULT` label if it exists, otherwise execution continues from just after the switch.

2.3.6 Flow of Control

The following commands affect the flow of control.

`RESULTIS` E
`RETURN`
`ENDCASE`
`LOOP`
`BREAK`
`GOTO` E
`FINISH`

`RESULTIS` causes evaluation of the smallest textually enclosing `VALOF` expression to return with the value of E.

`RETURN` causes evaluation of the current routine to terminate.

`LOOP` causes a jump to the point just after the end of the body of the smallest textually enclosing repetitive command (see Section 2.3.4). For a `REPEAT` command, this will cause the body to be executed again. For a `FOR` command, it causes a jump to where the control variable is incremented, and for the `REPEATWHILE` and `REPEATUNTIL` commands, it causes a jump to the place where the controlling expression is re-evaluated.

`BREAK` causes a jump to the point just after the smallest enclosing repetitive command (see Section 2.3.4).

`ENDCASE` causes execution of the commands in the smallest enclosing `SWITCHON` command to complete.

The `GOTO` command jumps to the command whose label is the value of E. See Section 2.4.1 for details on how labels are declared. The destination of a `GOTO` must be within the currently executing function or routine.

`FINISH` only remains in BCPL for historical reasons. It is equivalent to the call `stop(0, 0)` which causes the current program to stop execution. See the description of `stop(code, res)` page 66.

2.3.7 Compound Commands

It is often useful to be able to execute commands in a sequence, and this can be done by writing the commands one after another, separated by semicolons and enclosed in section brackets. The syntax is as follows:

$$\{\ C_1\ ;\ldots;\ C_m\ \}$$

where C_1 to C_m are commands.

Any semicolon ocurring at the end of a line may be omitted. For this rule to work, infixed expression operators may never start a line (see Section 2.2.9).

2.3.8 Blocks

A block is similar to a compound command but may start with some declarations. The syntax is as follows:

$$\{\ D_1\ ;\ldots;\ D_n;\ C_1\ ;\ldots;\ C_m\ \}$$

where D_1 to D_n are delarations and C_1 to C_m are commands. The declarations are executed in sequence to initialise any variables declared. A name may be used on the right hand side of its own and succeeding declarations and the commands (the body) of the block.

2.4 Declarations

Each name used in a BCPL program must in the scope of its declaration. The scope of names declared at the outermost level of a program include the right hand side of its own declaration and all the remaining declarations in the section. The scope of names declared at the head of a block include the right hand side of its own declaration, the succeeding declarations and the body of the block. Such declarations are introduced by the keywords `MANIFEST`, `STATIC`, `GLOBAL` and `LET`. A name is also declared when it occurs as the control variable of a for loop. The scope of such a name is the body of the for loop.

2.4.1 Labels

The only other way to declare a name is as a label of the form N`:`. This may prefix a command or occur just before the closing section bracket of a compound command or block. The scope of a label is the body of the block or compound command in which it was declared.

2.4.2 Manifest Declarations

A `MANIFEST` declaration has the following form:

`MANIFEST` { N_1`=`K_1 `; . . . ;` N_n`=`K_n }

where N_1`, . . ,`N_n are names (see Section 2.2.1) and K_1`, . . ,`K_n are manifest constant expressions (see Section 2.2.10). Each name is declared to have the constant value specified by the corresponding manifest expression. If a value specification (`=`K_i) is omitted, the a value one larger than the previously defined manifest constant is implied, and if `=`K_1 is omitted, then `=0` is assumed. Thus, the declaration:

```
MANIFEST { A; B; C=10; D; E=C+100 }
```

declares `A`, `B`, `C`, `D` and `E` to have manifest values `0`, `1`, `10`, `11` and `110`, respectively.

2.4.3 Global Declarations

The global vector is a permanently allocated region of store that may be directly accessed by any (separately compiled) section of a program (see Section 2.5. It provides the main mechanism for linking together separately compiled sections. A GLOBAL declaration allows a names to be explicitly associated with elements of the global vector. The syntax is as follows:

GLOBAL { N_1:K_1 ;...; N_n:K_n }

where N_1,..,N_n are names (see Section 2.2.1) and K_1,..,K_n are manifest constant expressions (see Section 2.2.10).

Each constant specifies which global vector element is associated with each variable.

If a global number (:K_i) is omitted, the next global variable element is implied. If =K_1 is omitted, then =0 is assumed. Thus, the declaration:

```
GLOBAL { a, b:200, c, d:251 }
```

declares the variables a, b, c and d occupy positions 0, 200, 201 and 251 of the global vector, respectively.

2.4.4 Static Declarations

A STATIC declaration has the following form:

STATIC { N_1=K_1 ;...; N_n=K_n }

where N_1,..,N_n are names (see Section 2.2.1) and K_1,..,K_n are manifest constant expressions (see Section 2.2.10). Each name is declared to be a statically allocated variable initialised to the corresponding manifest expression. If a value specification (=K_i) is omitted, the a value one larger than the previously defined manifest constant is implied, and if =K_1 is omitted, then =0 is assumed. Thus, the declaration:

```
STATIC { A; B; C=10; D; E=C+100 }
```

declares A, B, C, D and E to be static variables having initial values 0, 1, 10, 11 and 110, respectively.

2.4.5 LET Declarations

LET declarations are used to declare local variables, vectors, functions and routines. The textual scope of names declared in a LET declaration is the right hand side of its own declaration (to allow the definition of recursive procedures), and subsequent declarations and the commands.

Local variable, vector and procedure declarations can be combined using the word AND. The only effect of this is to extend the scope of names declared forward to the word LET, thus allowing the declaration of mutually recursive procedures. AND serves no useful purpose for local variable and vector declarations.

2.4.6 Local Variable Declarations

A local variable declaration has the following form:

$$\texttt{LET}\ N_1\ \texttt{,...,}\ N_n\ \texttt{=}\ E_1\ \texttt{,...,}\ E_n$$

where N_1,..,N_n are names (see Section 2.2.1) and E_1,..,E_n are expressions. Each name, N_i, is allocated space in the current stack frame and is initialized with the value of E_i. Such variables are called dynamic variables since they are allocated when the declaration is executed and cease to exist when control leaves their scope.

The query expression (?) should be used on the right hand side when a variable does not need an initial value.

2.4.7 Local Vector Declarations

$$\texttt{LET}\ N\ \texttt{=}\ \texttt{VEC}\ K$$

where N is a name and K is a manifest constant expression. A location is allocated for N and initialized to a vector whose lower bound is 0 and whose upper bound is K. The variable N and the vector elements (N!0 to N!K) reside in the runtime stack and only continue to exist while control remains within the scope of the declaration.

2.4.8 Procedure Declarations

A procedure declaration has the following form:

$$\texttt{LET}\ N\ \texttt{(}\ N_1\ \texttt{,...,}\ N_n\ \texttt{)}\ \texttt{=}\ E$$
$$\texttt{LET}\ N\ \texttt{(}\ N_1\ \texttt{,...,}\ N_n\ \texttt{)}\ \texttt{BE}\ C$$

where N is the name of the function or routine being declared, $N_1, . . , N_n$ are its formal parameters. A function is defined using = and returns E as result. A routine is defined using BE and executes the command C withou returning a result.

Some example declarations are as follows:

```
LET wrpn(n) BE { IF n>9 DO wrpn(n/10)
                 wrch(n REM 10 + '0')
               }
LET gray(n) = n NEQV n>>1
LET next() = VALOF { c := c-1
                     RESULTIS !c
                   }
```

If a procedure is declared in the scope of a global variable with the same name then the global variable is given an initial value representing the procedure (see section 2.5).

A procedure defined using equals (=) it is called a function and yields a result, while a procedure defines by BE is called a routine and does not. If a function is invoked as a routine its result in thrown away, and if a routine is invoked as a function its result is undefined. Functions and routines are otherwise similar. See section 2.2.3 for information about the syntax of to function and routine calls.

The arguments of a procedure behave like named elements of a dynamic vector and so exist only for the lifetime of the procedure call. This vector has as many elements as there are formal parameters and they receive their initial values from the actual parameters at the moment of call. Procedures are variadic; that is, the number of actual parameters need not equal the number of formals. If there are too few actual parameters then the missing higher numbered ones are left uninitialized, and if there are too many actual parameters, the extra ones are evaluated but their values discarded. Notice that the i^{th} argument can be accessed by the expression (@v)!i, where v is the first argument. The scope of the formal parameters is the body of the procedure.

Procedure calls are cheap in both space and execution time, with a typical space overhead of three words of stack per call plus one word for each formal parameter. In the Cintcode implementation, the execution overhead is typically just one executed insruction for the call and one for the return.

There are two important restrictions concerning procedures. One is that a GOTO command cannot make a jump to a label not declared within the current procedure, although such non local jumps can be made using the library proce-

dures `level` and `longjump`, described on page 61. The other is that dynamic free variables are not permitted.

2.4.9 Dynamic Free Variables

Free variables of a procedure are those that are used but not declared in the procedure, and they are restricted to be either manifest constants, static variables, global variables, procedures or labels. This implies that they are not permitted to be dynamic variables (ie local variables of another procedure). There are several reasons for this restriction, including the need to be able to represent a procedure in a single word, the ability to provide a safe separate compilation facility with the related ability to assign procedures to variables. It also allows the procedure calling to be efficient. Programmers used to languages such as Algol or Pascal will find that they need to change their programming style somewhat; however, most experienced BCPL users agree that the restriction is well worthwhile. One should note that C adopted the same restriction, although in that language it is imposed by the simple expedient of insisting that all procedures are declared at the outermost level, thus making dynamic free variables syntactically impossible.

A style of programming that is often be used to avoid the dynamic free variable restriction is exemplified below.

```
GLOBAL { var:200 }

LET f1(...) BE
{ LET oldvar = var    // Save the current value of var
  var := ...          // Use var during the call of f1
  ...
  f2(...)             // var may be used in f2
  ...
  IF ... DO f1(...)   // f1 may be called recursively
  var := oldvar       // restore the original value of var
}

AND f2(...) BE        // f2 uses var as a free variable
{  ... var ...  }
```

2.5 Separate Compilation

Large BCPL programs can be split up into sections that can be compiled separately. When loaded into memory they can communicate with each other using

a special area of store called the *Global Vector*. This mechanism is simple and machine independent and was put into the language since linkage editors at the time were so primitive and machine dependent.

Variables residing in the global vector are declared by GLOBAL declarations (see section 2.4.3). Such variables can be shared between separately compiled sections. This mechanism is similar to the used of BLANK COMMON in Fortran, however there is an additional simple rule to permit access to procedures declared in different sections.

If the definition of a function or routine occurs within the scope of a global declaration for the same name, it provides the initial value for the corresponding global variable. Initialization of such global variables takes place at load time.

The three files shown in Table 2.1 form a simple example of how separate compilation can be organised.

File `demohdr`	File `demolib.b`	File `demomain.b`

`GET "libhdr"`
 	
`GLOBAL { f:200 }`
	`GET "demohdr"`
	
	`LET f(...)  = VALOF`
	`{ ...`
	`}`
		`GET "demohdr"`
		
		`LET start() BE`
		`{ ...`
		`  f(...)`
		`}`

Table 2.1 - Separate compilation example

When these sections are loaded, global 200 is initialized to the entry point of function `f` defined in `demolib.b` and so is can be called from the function `start` defined in `demomain.b`.

The header file, `libhdr`, contains the global declarations of all the resident library functions and routines making all these accessible to any section that started with: `GET "libhdr"`. The library is described in the next chapter. Global variable 1 is called `start` and is, by convention, the first function to be called when a program is run.

Automatic global initialisation also occurs if a label declared by colon (`:`) occurs in the scope of a global of the same name.

Although the global vector mechanism has disadvantages, particularly in the organisation of library packages, there are some compensating benefits arising from its extreme simplicity. One is that the output of the compiler is available directly for execution without the need for a link editing step. Sections may also be loaded and unloaded dynamically during the execution of a program

using the library functions `loadseg` and `unloadseg`, and so arbitrary overlaying schemes can be organised easily. An example of where this is used is in the implementation of the Command Language Interpreter described in Chapter 8. The global vector also allows for a simple but effective interactive debugging system without the need for compiler constructed symbol tables. Again, this was devised when machines were small and disc space was very limited; however, some of its advantages are still relevant today.

Chapter 3

The Design of Cintcode

The Cintcode abstract machine used to implementation of Cintpos has eight registers as shown in figure 3.1.

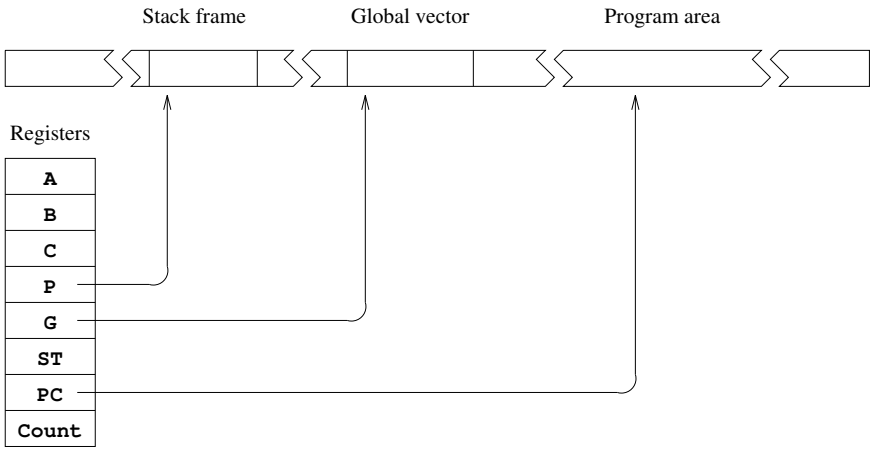


Figure 3.1: The Cintcode machine

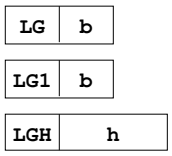
The registers `A` and `B` are used for expression evaluation, and `C` is used in byte subscription. `P` and `G` are pointers to the current stack frame and the global vector, respectively. `St` is zero when Cintcode interrupts are enabled, otherwise interrupts are disabled. `St=1` while executing code in the Cintpos kernel `KLIB`, `st==2` during the bootstrapping process and `st=3` when executing an interrupt service routine. `PC` points to the first byte of the next Cintcode instruction to execute. `Count` is a register used by the debugger. While it is positive, `Count` is decremented on each instruction execution, raising an exception (code 3) on reaching zero. When negative it normally causes a second (faster) interpreter to be used. The faster interpreter is usually implemented in assembly language and

provides few debugging aids. In particular it does not inspect or change the value of `Count`.

Cintcode encodes the most commonly occurring operations as single byte instructions, using multi-byte instructions for rarer operations. The first byte of an instruction is the function code. Operands of size 1, 2 or 4 bytes immediately follow some function bytes. The two instructions used to implement switches have inline data following the function byte. Cintcode modules also contains static data for strings, integers, tables and global initialisation data.

3.0.1 Global Variables

Global variables are referenced as frequently as locals and therefore have many function codes to handle them. The size of the global vector in most programs is less than 512, but Cintcode allows this to be as large are 65536 words. Each operation that refers to a global variable is provided with three related instructions. For instance, the instructions to load a global into register `A` are as follows:




```
B := A; A := G!b
B := A; A := G!(b+256)
B := A; A := G!h
```

Here, `b` and `h` are unsigned 8 and 16 bit values, respectively.

3.0.2 Composite Instructions

Cintcode contains many composite instructions, such as `AP3` which adds local `3` to the `A` register, and `L1P6` will load `v!1` into register `A`, assuming `v` is held in local `6`. Composite instructions improve the compactness and execution efficiency of Cintcode.

3.0.3 Relative Addressing

A relative addressing mechanism is used in conditional and unconditional jumps and the instructions: `LL`, `LLL`, `SL` and `LF`. All these instructions refer to locations within the code and are optimised for small relative distances. To simplify the generation, all relative addressing instructions are 2 bytes in length. The first being the function code and the second being an 8 bit relative address.

All relative addressing instructions have two forms: direct and indirect, depending on the least significant bit of the function byte. The details of both relative address calculations are shown in figure 3.2, using the instructions `J` and

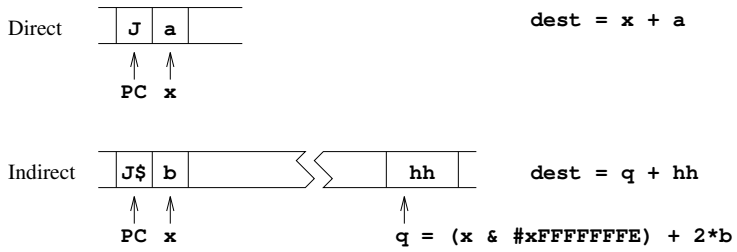


Figure 3.2: The relative addressing mechanism

J$ as examples. For the direct jump (J), the operand (a) is a signed byte in the range -128 to +127 which is added to the address (x) of the operand byte to give the destination address (dest). For the indirect jump, J$, the operand (b) is an unsigned byte in the range 0 to 255 which is doubled and added to the rounded version of x to give the address (q) of a 16 bit signed value hh which is added to q to give the destination address (dest).

The compiler places the resolving half word as late as possible to increase the chance that it can be shared by other relative addressing instructions to the same destination, as could happen when several ENDCASE statements occur in a large SWITCHON command. The use of a 16 bit resolving word places a slight restriction on the maximum size of relative references. Any Cintcode module of less than 64K bytes will have no problem.

3.1 The Cintcode Instruction Set

The resulting selection of function codes is shown in Table 3.1 and they are described in the sections that follow. In the remaining sections of this chapter the following conventions hold:

Symbol	Meaning
n	An integer encoded in the function byte.
Ln	The one byte operand of a relative addressing instruction.
b	An unsigned byte, range $0 \leq b \leq 255$.
h	An unsigned halfword, range $0 \leq h \leq 65535$.
w	A signed 32 bit word.
filler	Optional filler byte to round up to a 16 bit boundary.
`A`	The Cintcode `A` register.
`B`	The Cintcode `B` register.
`C`	The Cintcode `C` register.
`P`	The Cintcode `P` register.
`G`	The Cintcode `G` register.
`PC`	The Cintcode `PC` register.

3.1.1 Byte Ordering and Alignment

A Cintcode module is a vector of 32 bit words containing the compiled code and static data of a section of program. The first word of a module holds its size in words that is used as a relative address to the end of the module where the global initialisation data is placed. The last word of a module holds the highest referenced global number, and working back, there are pairs of words giving the global number and relative entry address of each global function or label defined in the module. A relative address of zero marks the end of the initialisation data. See section **??** for more details.

The compiler can generate code for either a big- or little-endian machine. These differ only in the byte ordering of bytes within words. For a little endian machine, the first byte of a 32 bit word is at the least significant end, and on a big-endian machine, it is the most significant byte. This affect the ordering of bytes in 2 and 4 byte immediate operands, 2 byte relative address resolving words, 4 byte static quantities and global initialisation data. Resolving words are aligned on 16 bit boundaries relative to the start of the module, and 4 byte static values are aligned on 32 bit boundaries. The 2 and 4 byte immediate operands are not aligned.

For efficiency reasons, the byte ordering is chosen to suit the machine on which the code is to be interpreted. The compiler option `OENDER` causes the BCPL compiler to compile code with the opposite endianess to that of the machine on which the compiler is running, see the description of the `bcpl` command on page 89.

	0	32	64	96	128	160	192	224
0	-	K	LLP	L	LP	SP	AP	A
1	-	KH	LLPH	LH	LPH	SPH	APH	AH
2	BRK	KW	LLPW	LW	LPW	SPW	APW	AW
3	K3	K3G	K3G1	K3GH	LP3	SP3	AP3	L0P3
4	K4	K4G	K4G1	K4GH	LP4	SP4	AP4	L0P4
5	K5	K5G	K5G1	K5GH	LP5	SP5	AP5	L0P5
6	K6	K6G	K6G1	K6GH	LP6	SP6	AP6	L0P6
7	K7	K7G	K7G1	K7GH	LP7	SP7	AP7	L0P7
8	K8	K8G	K8G1	K8GH	LP8	SP8	AP8	L0P8
9	K9	K9G	K9G1	K9GH	LP9	SP9	AP9	L0P9
10	K10	K10G	K10G1	K10GH	LP10	SP10	AP10	L0P10
11	K11	K11G	K11G1	K11GH	LP11	SP11	AP11	L0P11
12	LF	S0G	S0G1	S0GH	LP12	SP12	AP12	L0P12
13	LF$	L0G	L0G1	L0GH	LP13	SP13	XPBYT	S
14	LM	L1G	L1G1	L1GH	LP14	SP14	LMH	SH
15	LM1	L2G	L2G1	L2GH	LP15	SP15	BTC	MDIV
16	L0	LG	LG1	LGH	LP16	SP16	NOP	CHGC0
17	L1	SG	SG1	SGH	SYS	S1	A1	NEG
18	L2	LLG	LLG1	LLGH	SWB	S2	A2	NOT
19	L3	AG	AG1	AGH	SWL	S3	A3	L1P3
20	L4	MUL	ADD	RV	ST	S4	A4	L1P4
21	L5	DIV	SUB	RV1	ST1	XCH	A5	L1P5
22	L6	REM	LSH	RV2	ST2	GBYT	RVP3	L1P6
23	L7	XOR	RSH	RV3	ST3	PBYT	RVP4	L2P3
24	L8	SL	AND	RV4	STP3	ATC	RVP5	L2P4
25	L9	SL$	OR	RV5	STP4	ATB	RVP6	L2P5
26	L10	LL	LLL	RV6	STP5	J	RVP7	L3P3
27	FHOP	LL$	LLL$	RTN	GOTO	J$	ST0P3	L3P4
28	JEQ	JNE	JLS	JGR	JLE	JGE	ST0P4	L4P3
29	JEQ$	JNE$	JLS$	JGR$	JLE$	JGE$	ST1P3	L4P4
30	JEQ0	JNE0	JLS0	JGR0	JLE0	JGE0	ST1P4	-
31	JEQ0$	JNE0$	JLS0$	JGR0$	JLE0$	JGE0$	-	-

Table 3.1: The Cintcode function codes

3.1.2 Loading values

The following instructions are used to load constants, variables, the addresses of variables and function entry points. Notice that all loading instructions save the old value of register `A` in `B` before updating `A`. This simplifies the translation of dyadic expression operators.

`L`n	$0 \le n \le 10$	`B := A; A :=` n
`LM1`		`B := A; A := -1`
`L` b		`B := A; A :=` b
`LH` h		`B := A; A :=` h
`LMH` h		`B := A; A :=` $-h$
`LW` w		`B := A; A :=` w

These instructions load integer constants. Constants are in the range `-1` to `10` are the most common and have single byte instructions. The other cases use successively larger instructions.

`LP`n	$3 \le n \le 16$	`B := A; A := P!`n
`LP` b		`B := A; A := P!`b
`LPH` h		`B := A; A := P!`h
`LPW` w		`B := A; A := P!`w

These instructions load local variables and anonymous results addressed relative to `P`. Offsets in the range 3 to 16 are the most common and use single byte instructions. The other cases use successively larger instructions.

`LG` b	`B := A; A := G!`b
`LG1` b	`B := A; A := G!(`$b + 256$`)`
`LGH` h	`B := A; A := G!`h

`LG` loads the value of a global variable in the range 0 to 255, `LG1` loads globals in the range 256 to 511, and `LGH` can load globals up to 65535. Global numbers must be in the range 0 to 65535.

`LL` Ln	`B := A; A :=` variable Ln
`LL$` Ln	`B := A; A :=` variable Ln
`LF` Ln	`B := A; A :=` entry point Ln
`LF$` Ln	`B := A; A :=` entry point Ln

`LL` loads the value of a static variable and `LF` loads the entry address of a function, routine or label in the current module.

LLP b	B := A; A := @P!b
LLPH h	B := A; A := @P!h
LLPW w	B := A; A := @P!w
LLG b	B := A; A := @G!b
LLG1 b	B := A; A := @G!($b+256$)
LLGH h	B := A; A := @G!h
LLL Ln	B := A; A := @(variable Ln)
LLL$ Ln	B := A; A := @(variable Ln)

These instructions load the BCPL pointers to local, global and static variables.

3.1.3 Indirect Load

GBYT		A := B%A
RV		A := A!0
RVn	$1 \le n \le 6$	A := A!n
RVPn	$3 \le n \le 7$	A := P!n!A
L0Pn	$3 \le n \le 12$	B := A; A := P!n!0
L1Pn	$3 \le n \le 6$	B := A; A := P!n!1
L2Pn	$3 \le n \le 5$	B := A; A := P!n!2
L3Pn	$3 \le n \le 4$	B := A; A := P!n!3
L4Pn	$3 \le n \le 4$	B := A; A := P!n!4
LnG b	$0 \le n \le 2$	B := A; A := G!b!n
LnG1 b	$0 \le n \le 2$	B := A; A := G!(b+256)!n
LnGH h	$0 \le n \le 2$	B := A; A := G!h!n

These instructions are used in the implementation of byte and word indirection operators % and ! in right hand contexts.

3.1.4 Expression Operators

NEQ	A := -A
ABS	A := ABS A
NOT	A := ~A

These instructions implement the three monadic expression operators.

MUL	A := B * A
DIV	A := B / A
REM	A := B REM A
ADD	A := B + A
SUB	A := B - A
LSH	A := B << A
RSH	A := B >> A

AND	A := B & A
OR	A := B | A
XOR	A := B NEQV A

These instructions provide for all the normal arithmetic and bit pattern dyadic operators. The instructions DIV and REM generate exception 5 if the divisor is zero. Evaluation of relational operators in non conditional contexts involve conditional jumps and the FHOP instruction, see page 33. Addition is the most frequently used arithmetic operation and so there are various special instructions to improve its efficiency.

An	$1 \leq n \leq 5$	A := A + n
Sn	$1 \leq n \leq 4$	A := A - n
A b		A := A + b
AH h		A := A + h
AW w		A := A + w
S b		A := A - b
SH h		A := A - h

These instructions implement addition and subtraction by constant integer amounts. There are single byte instructions for incrementing by 1 to 5 and decremented by 1 to 4. For other values longer instructions are available.

APn	$3 \leq n \leq 12$	A := A + P!n
AP b		A := A + P!b
APH h		A := A + P!h
APW w		A := A + P!w
AG b		A := A + G!b
AG1 b		A := A + G!(b+1)
AGH h		A := A + G!b

These instructions allow local and global variables to be added to A. Special instructions for addition by static variables are not provided, and subtraction by a variable is not common enough to warrant special treatment.

3.1.5 Simple Assignment

SPn	$3 \leq n \leq 16$	P!n := A
SP b		P!b := A
SPH h		P!h := A
SPW w		P!w := A
SG b		G!b := A
SG1 b		G!(b+256) := A

SGH h		G!h := A
SL Ln		variable Ln := A
SL$ Ln		variable Ln := A

These instructions are used in the compilation of assignments to named local, global and static variables. The SP instructions are also used to save anonymous results and to layout function arguments.

3.1.6 Indirect Assignment

PBYT		B%A := C
XPBYT		A%B := C
ST		A!0 := B
STn	$1 \leq n \leq 3$	A!n := B
STOPn	$3 \leq n \leq 4$	P!n!0 := A
ST1Pn	$3 \leq n \leq 4$	P!n!1 := A
STPn	$3 \leq n \leq 5$	P!n!A := B
SOG b		G!b!0 := A
SOG1 b		G!(b+256)!0 := A
SOGH h		G!h!0 := A

These instructions are used in assignments in which % or ! appear as the leading operator on the left hand side.

3.1.7 Procedure calls

At the moment a function or routine is called the state of the stack is as shown in figure 3.3. At the entry point of a function or routine the first argument, if any, will be in register A and in memory P!3.

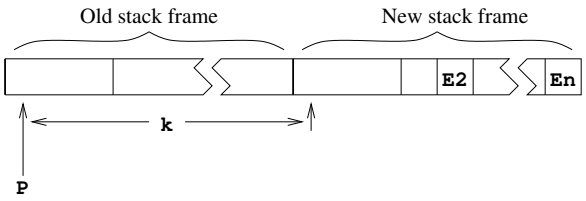


Figure 3.3: The moment of calling E(E1,E2,...En)

Kn	$3 \leq n \leq 11$
K b	
KH h	
KW w	

These instructions call the function or routine whose entry point is in `A` and whose first argument (if any) is in `B`. The new stack frame at position `k` relative to `P` where `k` is n, b, h or w depending on which instruction is used. The effect of these instructions is as follows:

```
P!k := P    // Save the old P pointer
P   := P+k  // Set its new value
P!1 := PC   // Save the return address
PC  := A    // Set PC to the entry point
P!2 := PC   // Save it in the stack for debugging
A   := B    // Put the first argument in A
P!3 := A    // Save it in the stack
```

As can be seen, three words of link information (the old `P` pointer, the return address and entry address) are stored in the base of the new stack frame.

KnG b	$3 \leq n \leq 11$
KnG1 b	$3 \leq n \leq 11$
KnGH h	$3 \leq n \leq 11$

These instructions deal with the common situation where the entry point of the function is in the global vector and the stack increment is in the range 3 to 11. The global number `gn` is b, $b+256$ or h depending on which function code is used and stack increment `k` is n. The first argument (if any) is in `A`. The effect of these instructions is as follows:

```
P!k := P    // Save the old P pointer
P   := P+k  // Set its new value
P!1 := PC   // Save the return address
PC  := G!gn // Set the new PC value from the global value
P!2 := PC   // Save it in the stack for debugging
P!3 := A    // Save the first argument in the stack
```

`RTN`

This instruction causes a return from the current function or routine using the previous `P` pointer and the return address held in `P!0` and `P!1`. The effect of the instruction is as follows:

```
PC  := P!1 // Set PC to the return address
P   := P!0 // Restore the old P pointer
```

When returning from a function the result will be in `A`.

3.1.8 Flow of Control and Relations

The following instructions are used in the compilation of conditional and unconditional jumps, and relational expressions. The symbol *rel* denotes `EQ`, `NE`, `LS`, `GR`, `LE` or `GE` indicating the relation being tested.

`J` *Ln*	`PC :=` *Ln*
`J$` *Ln*	`PC :=` *Ln*
`J`*rel* *Ln*	`IF B` *rel* `A DO PC :=` *Ln*
`J`*rel*`$` *Ln*	`IF B` *rel* `A DO PC :=` *Ln*
`J`*rel*`0` *Ln*	`IF A` *rel* `0 DO PC :=` *Ln*
`J`*rel*`0$` *Ln*	`IF A` *rel* `0 DO PC :=` *Ln*

The destinations of these jump instructions are computed using the relative addressing mechanism described in section 3.0.3. Notice than when the comparison is with zero, `A` holds the left operand of the relation.

`GOTO`	`PC := A`

This instruction is only used in the compilation of the `GOTO` command.

`FHOP`	`A := 0; PC := PC+1`

The `FHOP` instruction is only used in the compilation of relational expressions in non conditional contexts as in the compilation. The assignment: `x := y < z` is typically compiled as follows:

`LP4`	Load `y`
`LP5`	Load `z`
`JLS 2`	Jump to the `LM1` instruction if `y<z`
`FHOP`	A := FALSE; and hop over the `LM1` instruction
`LM1`	A := TRUE
`SP3`	Store in `x`

3.1.9 Switch Instructions

The instructions are used to implement switches are `SWL` and `SWB`, switching on the value held in `A`. They both assume that all case constants are in the range 0 to 65535, with the compiler taking appropriate action when this constraint is not satisfied.

`SWL` *filler n dlab* $L_0 \ldots L_{n-1}$

This instruction is used when there are sufficient case constants all within a small enough range. It performs the jump by selecting an element from a vector

of 16 bit resolving half words. The quantities n, *dlab*, and L_0 to L_{n-1} are 16 bit half words, aligned on 16 bit boundaries by the option filler byte. If `A` is in the range 0 to $n-1$ it uses the appropriate resolving half word $L_{\texttt{A}}$, otherwise it uses the resolving half word *dlab* to jump to the default label. See Section 3.0.3 for details on how resolving half words are interpreted.

`SWB` *filler* n *dlab* K_1 L_1 ... K_n L_n

This instruction is used when the range of case constants is too large for `SWL` to be economical. It performs the jump using a binary chop strategy. The quantities n, *dlab*, K_1 to K_n and L_1 to L_n are 16 bit half words aligned on 16 bit boundaries by the option filler byte. This instruction successively tests `A` with the case constants in the balanced binary tree given in the instruction. The tree is structured in a way similar to that used in heapsort with the children of the node at position i at positions $2i$ and $2i+1$. References to nodes beyond n are treated as null pointers. Within this tree, K_i is greater than all case constants in the tree rooted at position $2i$, and less than those in the tree at $2i+1$. The search starts at position 1 and continues until a matching case constant is found or a null pointer is reached. If `A` is equal to some K_i then `PC` is set using the resolving half word L_i, otherwise it uses the resolving half word *dlab* to jump to the default label. See Section 3.0.3 for details on how resolving half words are interpreted.

The use of this structure is particularly good for the hand written machine code interpreter for the Pentium where there are rather few central registers. Cunning use can be made of the add with carry instruction (`adcl`). In the following fragment of code, `%esi` points to n, `%eax` holds i and `A` is held in `%eab`. There is a test elsewhere to ensure that `A` is in the range 0 to 65535.

```
swb1:  cmpw (%esi,%eax,4),%bx ; { compare A with Ki
       je swb3                ;   Jump if A=Ki
       adcl                   ;   IF A>Ki THEN i := 2i
                              ;           ELSE i := 2i+1
       cmpw (%esi),%ax        ;
       jle swb1               ; } REPEATWHILE i<=n
```

The compiler ensures that the tree always has at least 7 nodes allowing the code to be further improved by preceding this loop with two copies of:

```
       cmpw (%esi,%eax,4),%bx ;   compare Ki with A
       je swb3                ;   Jump if match found
       adcl                   ;   IF A>Ki THEN i := 2i
                              ;           ELSE i := 2i+1
```

The above code is a great improvement on any straightforward implementation of the standard binary chop mechanism.

3.1.10 Miscellaneous

```
XCH             Exchange A and B
ATB             B := A
ATC             C := A
BTC             C := B
```

These instructions are used move values between register `A`, `B` and `C`.

`NOP`

This instruction has no effect.

`SYS`

This instruction is used in body of the hand written library routine `sys`. It provides an escape mechanism that allows for operations no directly available to Cintcode. These operations include input/output and control of the running environment. They are all described in Section 6.2.1.

Otherwise, it performs a system operation returning the result in `A`. In the C implementation of the interpreter this is done by the following code:

```
c = dosys(p, g);
```

`MDIV`

This instruction is used as the one and only instruction in the body of the hand written library routine `muldiv`, see Section 6.3.14. It divides `P!5` into the double length product of `P!3` and `P!4` placing the result in `A` and the remainder in the global variable `result2`. It then performs a function return (`RTN`). Its effect is as follows:

```
A            := <the result>
G!Gn_result2 := <the remainder>
PC           := P!1              // PC      := P!1
P            := P!0              // P       := P!0
```

`CHGCO`

This instruction is used in the implementation of coroutines. It is the one and only instruction in the body of the hand written library routine `chgco`. Its effect, which is rather subtle, is as follows:

```
G!Gn_currco!0 := P!0     // !currco := !P
PC            := P!1     // PC      := P!1
G!Gn_currco   := P!4     // currco  := cptr
P             := P!4!0   // P       := !cptr
```

`BRK`

This instruction is used by the debugger in the implementation of break points. It causes the interpreter to return with exception code 2.

3.1.11 Undefined Instructions

These instructions have function codes 0, 1, 232, 254 and 255, and they each cause the interpreter to return with exception code 1.

3.1.12 Corruption of `B`

To improve the efficiency of some hand written machine code interpreters, the following instructions are permitted to corrupt the value held in `B`:

`K KH KW K`*n* `K`*n*`G K`*n*`G1 K`*n*`GH`
`SWL SWB MDIV CHGCO`

All other instructions either set `B` explicitly or leave its value unchanged.

3.1.13 Exceptions

When an exception occurs, the interpreter saves the Cintcode registers in its register vector and yields the exception number as result. For exceptions caused by non existent instructions, BRK, DIV or REM the program counter is left pointing to the offending instruction. For more details see the description of `sys(1,` *regs*`)` on page 48.

Chapter 4

The Kernel Data Structures

Proper understanding of how Cintpos works requires knowledge of all the structures used by the kernel. All these structures are visible to any program running under Cintpos and their fields may be conveniently accessed using symbolic offsets declared in `libhdr`. The various kernel structures are as follows.

4.1 The root node

The fundamental structure that gives access to all other kernel structures is called the rootnode. It is positioned at location 100, but normally referenced using the manifest constant `rootnode`. Its fields are as follows:

Rootnode offset	Purpose
rtn.tasktab	Pointer to the task table
rtn.devtab	Pointer to the device table
rtn.tcblist	Pointer to the highest priority TCB
rtn.crntask	Pointer to the currently executing TCB
rtn.blklist	Pointer to the list of memory blocks
rtn.tallyv	Pointer to the tally vector
rtn.clkintson	TRUE if clock interrupts are enabled
rtn.lastch	The latest charcter typed on the keyboard
rtn.insadebug	TRUE if currently in sadebug
rtn.clkwq	List of packet sent to the clock device
rtn.membase	Pointer to the start of Cintcode memory
rtn.memsize	Size of Cintcode memory in words
rtn.info	Pointer to the info structure
rtn.sys	Entry address of the `sys` function
rtn.blib	List of the `BLIB` modules
rtn.boot	Pointer to the `BOOT` module
rtn.klib	Pointer to the `KLIB` modules
rtn.keyboard	The SCB for main keyboard input
rtn.screen	The SCB for main screen output
-	Other system dependent fields
	after this point
rtn.upb	The last field used.

4.2 The task table

The task table is pointed to by `rootnode!rtn.tasktab`. Its zeroth entry holds the upper bound of the table and each other entry is either zero or holds a pointer to the task control block (TCB) of a task. Tasks are identified by small positive integers. If a task with identified `id` exists, its TCB will be pointed to by `rootnode!rtn.tasktab!rtn.tasktab!id`.

4.3 Task control blocks

Each task in the system has a task control block (TCB) which contains information relating to that task. It fields are as follows:

TCB offset	Purpose
tcb.link	Link to the next TCB (or zero)
tcb.procid	Not used
tcb.taskid	The id of the task
tcb.pri	The task's priority, a positive number
tcb.wkq	List of packets sent to this task
tcb.state	The state of the task, a 4 bit integer
tcb.flags	The flags field
tcb.stsiz	The task's stack size
tcb.seglist	Pointer to the vector of segment lists
tcb.segnames	Not used
tcb.gbase	Either zero or the base of the global vector
tcb.sbase	Either zero or the base of the stack
tcb.timlist	Not used
tcb.iolist	Used only in pvstask/vobroot.bpl
-	Unused
-	Unused
tcb.a	Dump of A
tcb.b	Dump of B
tcb.c	Dump of C
tcb.p	Dump of P
tcb.g	Dump of G
tcb.intson	Dump of Intson
tcb.pc	Dump of PC
tcb.count	Dump of Count
tcb.upb	The last field used.

4.4 Global vectors

Each active task has a global vector whose zeroth element is the upper bound of the global vector. This size is normally 1000. Every other global element is initialised to `globword(=#xEFEF0000)+`n where n is the global number. If such a value is used as a function the interpreter will generate a fault. The globals corresponding to entry points in the modules specified by the segment lists are then set.

4.5 Stacks

Each active task has a stack used to hold arguments, locals and anonymous result during the evaluation of (nested) function calls. The stack pointed to by the TCB field sbase is the root coroutine stack for the task. A task may create other coroutine stacks during execution. The fields of a stack are shown below.

Stack offset	Purpose
co.pptr	The resumption P pointer if suspended
co.parent	-1 if root, 0 if orphan, or parent if active
co.list	Pointer to the next coroutine
co.fn	The coroutine's main function
co.size	The size of the coroutine's stack
co.c	Coroutine system work space

4.6 Memory blocks

Memory may be allocated and freed using `getvec` and `freevec`. The space is taken from a list of blocks pointed to by the `blklist` field of the root node. The zeroth word of each block in the block list contains its size, which must be even, and a one bit flag (the least significant bit) to indicate whether the block is currently allocated or free. A flag bit of 0 means the block is allocated, and 1 means it is free. The last block in the chain is marked with a zero indicating an allocated block of size zero. The functions `getvec` and `freevec` invoke C functions to do the allocation/deallocation. This is done using the `sys` function. Such modification of the block list is only permitted by the Cintcode interpreter thread. Other threads of the C program such as ones handling I/O devices may not change the block list.

If this simple mechanism proves to be too slow it can easily be improved.

4.7 Device control blocks

Cintpos has a device table pointed to by the `devtab` field of the root node. Its zeroth entry contains its upper bound and each other entry is either zero or points to a device control block (DCB). DCBs are used to control communication between Cintpos and external devices such as disc filing systems or serial lines connected via TCP/IP ports. The structure of a DCB is as follows.

DCB offset	Purpose
Dcb_type	Unused – once the driver code
Dcb_devid	The device id, a negative number
Dcb_wkq	List of packets sent to the device
Dcb_op	The function to call when the device receives a packet
Dcb_threadp	M/C address of location holding the pthread id
Dcb_cvp	M/C address of the pthread conditional variable
Dcb_intson	TRUE if the device may generate interrupts
Dcb_irq	TRUE if the device is ready to interrupt

The type field indicates the type of the device. The device types currently available are:

Type	Description
Devt_clk	The clock device
Devt_ttyin	The keyboard device
Devt_ttyout	The screen device
Devt_fileop	The main filing system device
Devt_tcpdev	Devices to handle TCP/IP connections

Devices are implemented using one Posix thread and one condition variable per device. At the lowest level the device is controlled by calls of `sys(Sys_devcom,` *com*`,` *arg*`)` where *com* can be `Devc_create`, `Devc_destroy`, `Devc_start`, `Devc_stop` or `Devc_setintson`. However, the user normally used `createdev` and `deletedev` to create and delete devices. Interaction with a device is normally done by sending it a packet using `qpkt` and waiting for it to reply using `taskwait`.

Chapter 5

Cintpos startup and initialisation

Although most of Cintpos is implemented in BCPL, its main program and cinterpreter are implemented in C. The function `main` is defined in `cintpos.c` and is the first code to execute. It reads the optional command line arguments....

More to come here

Chapter 6

The Resident Library

This chapter describes the resident library functions, routines and manifest constants that are declared in the standard library header file `libhdr` and defined in either `SYSLIB` or `sys/BLIB.b`.

6.1 The Standard Header File `libhdr`

The file `libhdr` contains the global and manifest declarations of all the procedures, variables and constants belonging to the standard resident library. Global variables 0 to 199 are reserved for use by the system. Of these, globals 0 to 99 mainly belong to BLIB and SYSLIB, while those between 133 and 149 belong to the Command Language Interpreter described in Chapter 8. The first global available to the user is numbered 200. For convenience this is declared as the manifest constant `ug`.

The size of the global vector is held in `globsize` (global 0) and, by convention, the global `result2` is used by some functions to return a second result. Global variable 1 is called `start` and is special since it must be defined by the user because it is the first function to be called when a program is run. Global 4 (called `clihook` is also special since it is useful when debugging programs, see page 65.

6.1.1 Architecture Constants

This constant `B2Wsh` holds the shift required to convert a BCPL pointer into a byte address. On 32 bit machines it is set to 2 while on some 64 bit implementations it is set to 3. The constant `bytesperword` is defined to be `1<<B2Wsh` and indicates how many bytes can be packed into a word. The constant `bitsperbyte` is set to 8 indicating the size of a byte. The constants `minint` and `maxint` hold

the largest negative and positive numbers that can be represented by a word in this implementation.

6.1.2 Other Manifest Constants

The constants `t_hunk`, `t_bhunk` and `t_end` are used in the representation of Cintcode Object Modules. Constants whose names start with `co_` are used in the implementation of the coroutine mechanism (see Section 6.3.12), and constants starting with `rtn_` denote offsets in the rootnode (see Section 6.3.1). The constants `InitObj`(=0) and `CloseObj`(=1) are the positions in the methods vector of the routines to initialise and close an object. See `mkobj` described on page 64.

6.2 SYSLIB

This module contains the definitions of the functions `sys`, `chgco` and `muldiv`. These functions are hand written in Cintcode because they need to execute the Cintcode instructions (`SYS`, `CHGCO` and `MDIV`) that cannot be generated by the BCPL compiler. The instruction `SYS` (see Section 6.2.1) provides an interface with the host operating system, `CHGCO` is used in the implementation of coroutines (see Section 6.3.12), and `MDIV` performs the operation required by `muldiv`(see Section 6.3.14).

To understand the `sys` function and the corresponding `SYS` Cintcode instruction it is necessary to know how the Cintcode System in implemented. It is mainly implemented in C and can be found in the files `sysc/cintpos.c`, `sysc/cinterp.c`, `sysc/kblib.c` and `sysc/nrastlib.c` with a header file `sysc/cinterp.h` that contains architecture dependent declarations. The file `sysc/cintpos.c` contains the main program and the the definition of `dosys` which provide access to I/O primitives and other system functions. The file `sysc/cinterp.c` is the C implementation of the Cintcode interpreter but there is usually a second more efficient interpreter such as `sysasm/LINUX/cintasm.s`. `cintasm` is faster than the C version since it has fewer built-in debugging aids and because it is carefully hand written in assembly language taking advantage of knowledge not available to a C compiler. Both `cinterp` and `cintasm` are callable from C and both receive a pointer to the base of the Cintcode memory vector and the position within it of where a dump of the Cintcode registers are held. Both will interpreters sequentially execute Cintcode instructions before returning with an integer result indicating why interpretation has ceased. The possible results are given below in the description of `sys(Sys_interpret,...)`.

The BCPL function sys provides an interface between BCPL and dosys, and can also control which version of the interpreter is used.

6.2.1 The BCPL sys Function

The sys function is defined by hand in SYSLIB it body is just the Cintcode instructions SYS followed by RTN. When SYS is encountered by the interpreter, it normally just calls dosys passing the BCPL P and G pointers as arguments. But when sys(Sys_quit,code) is executed, the interpreter saves the Cintcode registers in a vector and returns with the result code to the (C) program that called this invocation of the interpreter. This is normally used to exit from the Cintcode system, but can also be used to return from recursive invocations of the interpreter (see sys(Sys_interpret,regs) below).

A typical call of sys is as follows:

res := sys(*op*,...)

The action performed by this call depends on *op*. Some actions concern the management of the interpreter, some are concerned with input and output, while others provide access to functions implemented in C.

6.2.2 Interpreter Management Sys Functions

The Cintcode System normally has two resident interpreters. One is called cinterp and is implemented in C and the other is called cintasm which is normally implemented by hand in assembly language. The assembly language version runs faster but but provides fewer debugging and statistics gathering aids. It is possible to select dynamically which interpreter is running by setting a value in the Cintcode count register. A positive value causes cinterp to run. Each time cinterp executes a Cintcode instruction it decrements count and returns with an error code of 3 when count reaches zero. If the value in count is -1, cintasm is invoked and runs without changing count. With some debugging versions of cintasm, setting count to -2 causes cintasm to execute just one instruction and then return with error code 10. This feature assists the debugging of a new versions of cintasm. The count register can be set by means of the sys(Sys_setcount,...) call described below.

A second version of the BCPL Cintcode system called rasterp. It uses just one interpreter implemented in C, and can be called upon to generate raster image data. For more information, see the description of sys(Sys_setraster,...) on page 52 and the raster and rast2ps commands on page 93.

oldcount `:= sys(Sys_setcount,` *newcount*`)`

Under normal conditions when the interpreter is running under the control of code in `BOOT`, this sets the Cintcode count register to *newcount* and returns the previous value. If the count register is now less than `0`, interpretation continues using the fast interpreter (`cintasm`), but if it is greater than `0`, interpretation continues using the slow interpreter (`cinterp`). The code to make this selection can be found in the execution loop occuring at the end of the function `start` defined in `sys/BOOT.b`.

For some (debugging) versions of `cintasm` a count of `-2` causes the (fast) interpreter to execute just one Cintcode instruction before returning with an error code of `10`.

The `sys(Sys_setcount,...)` function is used by the CLI command `interpreter` to select which interpreter is to be used (see Section 8.2). It is also used by the `instrcount` function defined in `BLIB.b` (see page 66).

`sys(Sys_quit,` *code*`)`

This will cause a return from the the interpreter yielding *code* as the return code. A *code* of zero denotes successful completion and, if invoked at the outermost level, causes the BCPL Cintcode System to terminate.

res `:= sys(Sys_interpret,` *regs*`)`

This function enters the Cintcode interpreter recursively with the Cintcode registers set to values specified in the vector *regs*. The elements of *regs* are as follows:

regs`!0`	A register	– work register
regs`!1`	B register	– work register
regs`!2`	C register	– work register
regs`!3`	P register	– the stack frame pointer
regs`!4`	G register	– the base of the global vector
regs`!5`	ST register	– the status register
regs`!6`	PC register	– the program counter
regs`!7`	Count register	– see below

When `cinterp` is active, the count register is decremented every time an instruction is interpreted. When the count goes negative the interpreter saves the registers and returns with a result (=3) to indicate what has happened. If the count register is positive it indicates how many Cintcode instructions should be executed before the interpreter returns. A count of `-1` is treated as infinity and causes the fast interpreter `cintasm` to be used.

Either interpreter returns when a fault, such as division by zero, occurs or when a call of `sys(Sys_quit,...)` or `sys(Sys_setcount,...)` is made. When returning, the current state of the Cintcode registers is saved back into *regs*. The returned result is either the second argument of `sys(Sys_quit,...)` or one of the built-in return codes in the following table:

-1	Re-enter the interpreter with a new value in the the count register
0	Normal successful completion (by convention)
1	Non existent Cintcode instruction
2	`BRK` instruction encountered
3	Count has reached zero
4	PC set to a negative value
5	Division by zero
10	Single step interrupt from the fast interpreter (debugging)
11	The watched location has just been updated.

`sys(Sys_saveregs, regs)`
`sys(Sys_rti, regs)`
`sys(Sys_setst, val)`

These calls are normally only made in the Cintpos kernel and the interrupt service routine. `Sys_saveregs` causes the current state of the cintcode registers to be saved in the vector `regs`. `Sys_rti` will set the Cintcode registers, except for `count`, to the values held in the vector `regs`. `Sys_setst` will set the Cintcode register `st` to the given value. It is primarily used in the kernel to enable and disable interrupts.

`sys(Sys_watch,` *addr*`)` On some systems this will cause the interpreter to watch the given address. When the watched value changes, fault 11 will be generated causing the standalone debugger to be entered. A zero address disables the watch feature.

`sys(Sys_tracing, b)`
`sys(Sys_tally, flag)`

When running under `cinterp`, the calls `sys(Sys_tracing,TRUE)` and `sys(Sys_tracing,FALSE)` provide a primitive trace facility to aid debugging the `cinterp` itself and are not normally of use to ordinary users. When running under `cinterp`, the calls `sys(Sys_tally,TRUE)` and `sys(Sys_tally,FALSE)` provide the profiling facility that was used to obtain execution statistics. It uses the tally vector to hold frequency counts of Cintcode instructions executed. When

tallying is enabled, the i^{th} element of the tally vector is incremented every time the instruction at location i of the Cintcode memory is executed. The size of the tally vector can be specified when entering the Cintpos system. This size is stored in its zeroth element. The tally vector is held in the `rtn.tallyv` field of the rootnode. Note this profiling facility is only available when running under `cinterp`. Statistics of the execution of a program can be gathered and analysed using the CLI command `stats` (see Section 8.2).

6.2.3 Primitive I/O Operations

ch `:= sys(Sys_sardch)`

Return the next character from the keyboard. The character is echoed to standard output (normally the screen).

`sys(Sys_sawrch,` *ch*`)`

Send character *ch* to the standard output (normally the screen). the character linefeed is transmitted as carriage return followed by linefeed, and the output is flushed.

n `:= sys(Sys_read,` *fp*`,` *buf*`,` *len*`)`

Read upto *len* bytes from the file specified by the file pointer *fp* into the byte buffer *buf*. The file pointer must have been created by a call of `sys(Sys_openread,...)` or `sys(Sys_openreadwrite,...)`. The number of bytes actually read is returned as the result. A result of zero indicate end of file, and a negative value indicates an error.

n `:= sys(Sys_write,` *fp*`,` *buf*`,` *len*`)`

Write *len* bytes to the file specified by the file pointer *fp* from the byte buffer *buf*. The file pointer must have been created by a call of `sys(Sys_openwrite,...)` or `sys(Sys_openread,...)`. The result is the number of bytes transferred, or zero if there was an error.

fp `:= sys(Sys_openread,` *name*`)`

This opens for reading the file whose name is given by the string *name*. It returns `0` if the file cannot be opened, otherwise it returns the file pointer for the opened file. See page 61 for information about the treatment of file names.

fp `:= sys(Sys_openwrite,` *name*`)`

This opens for writing the file whose name is given by the string *name*. It

returns 0 if the file cannot be opened, otherwise it returns the file pointer for the opened file. See page 61 for information about the treatment of file names.

fp := sys(Sys_openreadwrite, *name*)

This opens for reading and writing the file whose name is given by the string *name*. It returns 0 if the file cannot be opened, otherwise it returns the file pointer for the opened file. See page 61 for information about the treatment of file names.

res := sys(Sys_close, *fp*)

This closes the file whose file pointer is fp. It returns zero if successful.

res := sys(Sys_deletefile, *name*)

This deletes the file whose name is given by name. See page 61 for information about the treatment of file names.

res := sys(Sys_renamefile, *old*, *new*)

This renames file *old* to *new*. It returns zero if successful.

res := sys(Sys_getvec, *upb*)

This allocates a vector whose lower bound is zero and whose upper bound is *upb*. It returns zero if the request cannot be satified.

sys(Sys_freevec, *ptr*)

If *ptr* is zero it does nothing, otherwise it returns the space pointed to by *ptr* that must have previously been allocated by sys(Sys_getvec,...).

res := sys(Sys_loadseg, *name*)

This loads a Cintcode module from file *name*. It returns a pointer to the loaded module if successful. Otherwise it returns zero.

res := sys(Sys_globin, *seg*)

This initializes the global variables defined in the loaded module pointed to by *seg*. It returns zero is there is an error.

res := sys(Sys_unloadseg, *seg*)

This unloads the the loaded module given by *seg*. If *seg* is zero it does nothing.

res := sys(Sys_muldiv, *a*, *b*, *c*)

This invokes the C implementation of muldiv. It returns the result of dividing *c* into the double length product of *a* and *b*. It sets result2 to the remainder.

res := sys(Sys_intflag)

This returns TRUE if the program is being interrupted by the user. On many systems this is not implemented and just returns FALSE.

res := sys(Sys_setraster, *n*, *arg*)

This invokes the setraster primitive used in the rastering version of the Cintcode interpreter (rasterp). It is used to control the collection of data for time-memory raster images. If *n*=3, it returns 0 if rastering is available and -1 otherwise. If *n*=2, the memory granularity is set to *arg* bytes per pixel. The default is 12. If *n*=1, the number of Cintcode instructions per raster line is set to *arg*. The default is 1000.

If *n* is zero and *arg* is non-zero then rastering is activated to send its output to file *name* (the rastering data file). Raster information is collected for the duration of the next CLI command. If *n* and *arg* are both zero the rastering data file is closed.

The raster data file is an ASCII file that encodes the raster lines using run length encoding. Typical output is as follows:

K1000 S12	1000 instructions per raster line, 12 bytes per pixel
W10B3W1345B1N	10 white, 3 black, 1345 white, 1 black, newline
W13B3W12B2N	etc
...	

See the CLI commands raster and rast2ps on page 93 for more information on how to use the rastering facility.

res := sys(Sys_cputime)

This returns the CPU time in milliseconds since the Cintcode system was entered.

res := sys(Sys_filemodtime, *name*)

This returns time of last modification of the file given by *name*.

res := sys(Sys_setprefix, *prefix*)

This is primarily a function for the Windows CE version of the BCPL Cintcode System for which there is no current working directory machanism. It sets the prefix that is prepended to all future relative file names. See Sections 6.3.6 and the CLI prefix command described on page 93.

str := sys(Sys_getprefix)

This returns the current prefix string. See sys(Sys_setprefix,...) above.

res := sys(Sys_graphics,...)

This is currently only useful on the Windows CE version of the BCPL Cintcode system. It performs an operation on the graphics window. The graphics window is a fixed size array of 8-bit pixels which can be written to and whose visibility can be switched on and off.

res := sys(Sys_chready)

This returns TRUE if a character is ready to be read from the keyboard. This function is currently only available in the Windows CE implementation.

res := sys(Sys_seek, *fd*, *pos*)

This sets the current position of the file with descriptor *fd* to *pos*.

res := sys(Sys_tell, *fd*)

This returns the current position in the file whose file descriptor is *fd*.

res := sys(Sys_waitirq)

This will suspend the execution of the interpreter until woken up again by a signal from a device. This call is only used in the IDLE task.

res := sys(Sys_devcom, *dcb*, *com*, *arg*)

This will execute the device command *com* for the device with specified DCB. If the command needs an argument it may be passed in *arg*.

res := sys(Sys_ftime, *tv*)

This set the current real time in *tv*!0 to *tv*!4. See the source of cintpos.c for details.

res := sys(Sys_usleep, *usecs*)

This will cause the interpreter to sleep for the specified number of microseconds.

res := sys(Sys_filesize, *fd*)

This returns the size in bytes of the file whose file descriptor is *fd*. A negative result indicates an error.

6.3 BLIB

BLIB contains the main part of the standard library. It is implemented in BCPL and its source code can be found in sys/BLIB.b.

6.3.1 Initialization

When the Cintcode System is started, a region of store is allocated for the Cintcode memory. This is where Cintode stacks, global vectors, program code and system data are placed. It contains a vector called the *rootnode* that allows running programs to locate components of the system data structure. The global variable `rootnode` holds a pointer to the rootnode and there are manifest constants (define in `libhdr`) to ease access to its various elements. For instance, the pointer to the start of the memory block chain can be obtained by evaluating `rootnode!rtn_blklist`. Ten elements are defined in the rootnode as shown below.

Position	Expression	Value
0	`rootnode!rtn.tasktab`	Vector of tasks.
1	`rootnode!rtn.devtab`	Vector of devices.
2	`rootnode!rtn.tasktab`	List of tasks in decreasing priority order.
3	`rootnode!rtn.tasktab`	The current task.
4	`rootnode!rtn.blklist`	The start of the chain of memory blocks.
5	`rootnode!rtn.tallyv`	The tally vector.
6	`rootnode!rtn.clkintson`	Clock interrupts on/off flag.
7	`rootnode!rtn.lastch`	For sadebug polling input.
8	`rootnode!rtn.insadebug`	Looked at by the ttyin device.
9	`rootnode!rtn.bptaddr`	Vector of breakpoint addresses.
10	`rootnode!rtn.bptinstr`	Vector of breakpoint instructions.
11	`rootnode!rtn.clwkq`	The clock work queue.
12	`rootnode!rtn.membase`	Pointer to the start of the Cintcode memory.
13	`rootnode!rtn.memsize`	The size of the Cintcode memory in words.
15	`rootnode!rtn.sys`	The BCPL `sys` function.
16	`rootnode!rtn.boot`	The `BOOT` code segment.
17	`rootnode!rtn.klib`	The `KLIB` code segments.
18	`rootnode!rtn.blib`	The `BLIB` code segments.
19	`rootnode!rtn.keyboard`	The stream control block for the keyboard.
20	`rootnode!rtn.screen`	The stream control block for the screen.
21	`rootnode!rtn.vecstatsv`	Memory allocation statistics.
22	`rootnode!rtn.dumpflag`	=TRUE to dump memory on fault.
23	`rootnode!rtn.envlist`	List of logical name-value pairs.
24	`rootnode!rtn.abortcode`	Latest reason for leaving the interpreter.

6.3.2 Space Allocation

Allocation and release of space is performed by `getvec` and `freevec`, which use a first fit algorithm based on a list of blocks chained together in memory order. Word zero of each block in the chain contains a flag in its least significant bit indicating whether the block is allocated or free(0=allocated, 1=free). The rest of the word is an even number giving the size of the block in words. A pointer to the first block in the chain is held in the `rootnode!rtn.membase`. The size of blocks are rounded up to an even number of words and four words of data are added at the end which are used by `freevec` can check that the block has not been misused.

`freevec(`*v*`)`

This returns vector *v* to free store. It does nothing if $v = 0$, and for non zero *v* it checks the four words at the end of *v* to see if the vector looks like one that was allocated by `getvec`. If it detects an error it outputs a message and calls `abort(999)`. It currently calls `safreevec`.

v `:= getvec(`*upb*`)`

This allocates a vector with upper bound *upb* from the first large enough free block on the block list. If no such block exists it returns zero. In the current implementation it just calls `sagetvec`.

6.3.3 Standalone Functions

`safreevec(`*v*`)`

This returns vector *v* to free store by calling: `sys(Sys_freevec,` *v*`)`.

v `:= sagetvec(`*upb*`)`

This allocates a vector with upper bound *upb* by calling: `sys(Sys_getvec,` *upb*`)`.

ch `:= sardch()`

This function returns the next character from the keyboard as soon as it is available, echoing the character to the screen. It is implemented by means of `sys(10)`, described above.

`sawrch(`*ch*`)`

The call `sawrch(`*ch*`)` outputs the character *ch* to the screen. It is implemented by means of `sys(11,`*ch*`)`, described above.

`sawritef(`*format*`,`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`,`*l*`,`*m*`,`*n*`,`*o*`,`*p*`,`*q*`,`*r*`,`*s*`,`*t*`,`*u*`,`*v*`,`*w*`,`*x*`,`*y*`,`*z*`)`

This function has the same effect as `writef` described below but uses `sawrch` rather than `wrch` to perform the writing. Output thus goes straight to the standard output stream independent of which output stream is currently selected.

seg `:= saloadseg(`*name*`)`

This loads the named segment of code, returning the segment if successful or zero on failure. It calls: It calls `sys(Sys_loadseg,...)`.

`saunloadseg(`*seg*`)`

This unloads a previously loaded code segment. It calls `sys(Sys_unloadseg,...)`.

seg `:= saglobin(`*seg*`)`

This initialises the globals of a previously loaded code segment. It calls `sys(Sys_globin,...)`.

`sadeletefile(`*name*`)`

This deletes a named file. It directly calls `sys(Sys_deletefile,...)`.

`sarenamefile(`*oldname*`,` *newname*`)`

This renames a file. It directly calls `sys(Sys_renamefile,...)`.

6.3.4 Stream Functions

`endread()`

The call `endread()` closes the currently selected input stream. If there is an error it aborts (with code 190).

`endstream(`*scb*`)`

This call closes stream *scb*. If there is an error it aborts (with code 190).

`endwrite()`

This routine closes the currently selected output stream. If there is an error it aborts with code 189 (trouble with writing) or code 191 (trouble with closing).

scb `:= findinput(`*name*`)`

The call `findinput(`*name*`)` opens an input stream. If *name* is the string `"*"` then it opens the standard input stream which is normally from the keyboard, otherwise *name* is taken to be a device or file name. If the stream cannot be

opened the result is zero. See Section 6.3.6 for information about the treatment of filenames.

scb := findoutput(*name*)

This function opens an output stream specified by the device or file name *name*. If *name* is the string "*" then it opens the standard output stream which is normally to the screen. If the stream cannot be opened, the result is zero. See Section 6.3.6 for information about the treatment of filenames.

scb := input()

This returns the currently selected input stream.

scb := output()

This function returns the SCB of the currently selected output stream.

scb := pathfindinput(*name*, *pathname*)

The call pathfindinput(*name*, *pathname*) opens an input stream. If *name* is the string "*" then input comes from the keyboard, otherwise *name* is taken to be a filename. If the stream cannot be opened the file directories specified by the shell variable *pathname* are searched. If the file is not found in any of these directories, the result is zero. The shell variable BCPLPATH is used by the Command Language Intepreter (see Chapter 8) when searching for commands, and by the BCPL compiler when processing GET directives.

selectinput(*scb*)

The call selectinput(*scb*) selects *scb* as the currently selected input stream. It aborts (with code 186) if *scb* is not an input stream.

selectoutput(*scb*)

This routine selects *scb* as the currently selected output stream. It aborts (with code 187) if scb is not an output stream.

settimeout(*scb*, *timeout*)

This set the timeout value for the specified stream (*scb*) to the specified value (*timeout*). A timeout od zero means that there is no time. A positive timeout specifies the maximum time in milli-seconds that an input/output operation may take, and a negative timemout puts the stream into polling mode where input/output operations return immediately either successfully or with an indication that the required operation could not be performed immediately

6.3.5 Input and Output Functions

n `:= readn()`

This reads an optionally signed decimal integer from the currently selected input stream. Leading spaces, tabs and newlines are ignored. If the number is syntactically correct, `readn` returns its value with `result2` set to zero, otherwise it returns zero with `result2` set to `-1`. In either case, it uses `unrdch` to replace the terminating character.

`wrch(`*ch*`)`

This routine writes the character *ch* to the currently selected output stream. If output is to the screen, *ch* is transmitted immediately. It aborts (with code 189) if there is a write failure.

ch `:= rdch()`

This reads the next character from the currently selected input stream. If the stream is exhausted, it returns the special value `endstreamch`(=-1). Input from the keyboard is buffered until the ENTER (or RETURN) key is pressed to allow simple line editing in which the backspace and rubout keys may be used to delete the most recent character typed. For streams that aloow for timeouts (currently only TCP streams), if the timeout value is zero, `rdch` waits until a character is available or the stream is exhausted. If the timeout value is positive, `rdch` waits until a character is available or the stream is exhausted, but if the timeout period is exceeded it returns `timeoutch`(=-2). If the timeout value is negative, `rdch` returns immediately with either the next character if available, or `enstreamch`(=-1) if the stream is exhasted, or it returns `pollingch`(=-3) if no character is currently available. See `settimeout` described above.

flag `:= unrdch()`

The call `unrdch()` attempts to step the current input stream back by one character position. It returns `TRUE` if successful, and `FALSE` otherwise. A call of `unrdch` will always succeeds the first time after a call of `rdch`. It is useful in functions such as `readn` (described below) where single character lookahead is necessary.

`newline()`
`newpage()`

These two routines simply output the newline character (`'\n'`) or the newpage (form-feed) character (`'\p'`), respectively, to the currently selected output stream.

`writed(`*n*`, `*d*`)`
`writeu(`*n*`, `*d*`)`
`writen(`*n*`)`

These routines output the integer *n* in decimal to the currently selected output stream. For `writed` and `writeu`, the output is padded with leading spaces to fill a field width of *d* characters. If `writen` is used or if *d* is too small, the number is written without padding. If `writeu` is used, *n* is regarded as an unsigned integer.

`writehex(`*n*`, `*d*`)`
`writeoct(`*n*`, `*d*`)`
`writebin(`*n*`, `*d*`)`

These routines output, repectively, the least significant *d* hexadecimal, octal or binary digits of the integer *n* to the currently selected output stream.

`writes(`*str*`)`
`writet(`*str*`, `*d*`)`

These routines output the string *str* to the currently selected output stream. If `writet` is used, trailing spaces are added to fill a field width of `d` characters. In either case if *str* does not look like a string it is replaced by `##Bad string##` rather than causing a memory fault.

`writef(`*format*`,`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`,`*l*`,`*m*`,`*n*`,`*o*`,`*p*`,`*q*`,`*r*`,`*s*`,`*t*`,`*u*`,`*v*`,`*w*`,`*x*`,`*y*`,`*z*`)`

The first argument (*format*) is a string that is copied character by character to the currently selected output stream until a substitution item such as `%s` or `%i5` is encountered when a value (usually the next argument) is output in the specified format. The substitution items are given in table 6.1. If *format* does not look like a string it is replaced by `"##Bad format##"` rather than causing a memory fault.

When a field width (denoted by *n* in the table) is required, it is specified by a single character, with 0 to 9 being represented by the corresponding digit and 10 to 35 represented by the letters `A` to `Z` or `a` to `z`. Format and field width characters are case insensitive.

As an illustrations of the use of the `%p` substitution item, the following code:

```
FOR i = 0 TO 2 DO writef("There %p\is\are\%- %n item%-%ps*n", i)
```

will write:

```
There are 0 items
There is 1 item
There are 2 items
```

Item	Substitution
`%s`	Write the next argument as a string using `writes`.
`%t`n	Write the next argument as a left justified string in a field width of n characters using `writet`.
`%c`	Write the next argument as a character using `wrch`.
`%b`n	Write the next argument as a binary number in a field width of n characters using `writebin`.
`%o`n	Write the next argument as an octal number in a field width of n characters using `writeoct`.
`%x`n	Write the next argument as a hexadecimal number in a field width of n characters using `writehex`.
`%i`n	Write the next argument as a decimal number in a field width of n characters using `writed`.
`%n`	Write the next argument as a decimal number in its natural field width using `writen`.
`%u`n	Write the next argument as an unsigned decimal number in a field width of n characters using `writeu`.
`%+`	Step forward over the next argument value.
`%-`	Step backward over the previous argument value.
`%p`\$s$\$p$\	If the next argument has value is 1 write the text s otherwise write p. In either case the argument pointer is advanced by one position.
`%p`c	Unless the next argument value is 1 write the character c which is normally an 's', and advance the argument pointer one position. The character c may not be a backslash.
`%f`	Treat the next argument value as a format string, advancing the next argument pointer appropriately.
`%m`	Find the message format corresponding to the next argument value, and use it in a way similar to `%f` above.
`%%`	Write the character `%`.

Figure 6.1: `writef` substitution items

The `%m` substitution item is provided to simplify the generation of messages in different languages. Each message has a message number which is converted into a format string by the function `get.text` which is normally overridden by a definition in the application program. The default definition of `get.text` in

sysb/BLIB.b always yields the string: "<mess:%-%n>".

The implementation of writef (in sysb/BLIB.b) is a good example of how a variadic function can be defined.

6.3.6 The Filing System

BCPL uses the filing system of the host machine and so such details as the maximum length of filenames or what characters they may contain are machine dependent. However, within a file name the characters slash (/) and backslash (\) are regarded as a file separators and are converted into the appropriate separator for the operating system being used. For Unix systems this is a slash, for MS-DOS, WINDOWS and OS/2 it is a backslash, and on the Apple Macintosh it is a colon. Thus, under MS-DOS, findoutput can be given a file name such as "tmp/RASTER" and it will be treated as if the name "tmp\RASTER" had been given. This somewhat ad hoc feature greatly improves portability between systems.

A file name prefix feature is available primarily for systems such as Windows CE where there is no concept of a current working directory. The system maintains a prefix that is prepended to any non absolute file name before it is passed to the operating system. A file name is absolute if it starts with a slash or backslash or, on Windows systems, if it starts with a letter followed by a colon. A separator is placed between the prefix and the given file name.

The current prefix can be inspected and changed using the calls: sys(32,*prefix*) and sys(33), or the CLI command prefix described on page 93.

6.3.7 File Deletion and Renaming

flag := deletefile(*name*)
flag := renamefile(*oldname*, *newname*)

The call deletefile(*name*) deletes the file with the given name. It returns TRUE if the deletion was successful, and FALSE otherwise. The call renamefile(*oldname*, *newname*) renames the file *oldname* as file *newname*, deleting *newname* if necessary. Both *oldname* and *newname* are strings. The function returns TRUE if the renaming was successful, and FALSE otherwise.

6.3.8 Non Local Jumps

P := level()
longjump(*P*, *L*)

The call `level()` returns the current stack frame pointer for use in a later call of `longjump`. The call `longjump(`P`, `L`)` causes execution to resume at label L in the body of a procedure that owns the stack frame given by P. Jumps to labels within the current procedure can be performed using the `GOTO` command, so `level` and `longjump` are only needed for non local jumps.

6.3.9 Command Arguments

This implementation of BCPL incorporates a command language interpreter which is described in Chapter 8. Most commands require argument and these are most easily read using the functions: `rditem`, `rdargs`, `findarg` and `str2numb`.

kind `:= rditem(`*v*`, `*upb*`)`

The function `rditem` reads a command argument from the currently selected input stream. After ignoring leading spaces and tab characters, it packs the item into the vector *v* whose upper bound is *upb* and returns an integer describing the kind of item read. Table 6.2 gives the kinds of item that can be read and corresponding item codes.

Example items	Kind of item	Item code
`;`		4
carriage return		3
`"from"` `"\ntwo words\n"`	Quoted string	2
`abc` `123-45*6`	Unquoted string	1
end-of-stream	Terminator	0
	An error	-1

Figure 6.2: `rditem` results

It is possible to include newline characters within quoted strings using the escape sequence `*n`.

res `:= rdargs(`*keys*`, `*argv*`, `*upb*`)`

The first argument (*keys*) is a string specifying a list of argument keywords with possible qualifiers. The second and third arguments provide a vector (*argv*) with a given upper bound (*upb*) in which the decoded arguments are to be placed.

If `rdargs` is successful, it returns the number of words used in *argv* to represent the decoded command arguments, and, on failure, it returns zero.

Command arguments are read from the currently selected input stream using a decoding mechanism that permits both positional and keyed arguments to be freely mixed. A typical use of `rdargs` occurs in the source of the `input` command as follows:

```
UNLESS rdargs("FROM/A,TO/K,N/S", argv, 50) DO
{ writef "Bad arguments for INPUT\n"
  ...
}
```

In this example, there are three possible arguments and their values will be placed in the first three elements of `argv`. The first argument has keyword `FROM` and must receive a value because of the qualifier `/A`. The second has keyword `TO` and its qualifier `/K` insists that, if the argument is given, it must be introduced by its keyword. The third argument has the qualifier `/S` indicating that it is a switch that can be turned on by the presence of its keyword. If an argument is supplied, the corresponding element of `argv` will be set to `-1`, if it is a switch argument, otherwise it will be set to a string containing the characters of the argument value. The elements of `argv` corresponding to unset arguments are cleared. Table 6.3 shows the values in placed in `argv` and the result when the call:

`rdargs("FROM/A,TO=AS/K,N/S", argv, 50)`

is given various argument strings. This example illustrates that keyword synonyms can be defined using `=` within the key string. Positional arguments are those not introduced by keywords. When one is encountered, it becomes the value of the lowest numbered unset non-switch argument.

Arguments	argv!0	argv!1	argv!2	Result
abc TO xyz	"abc"	"xyz"	0	>0
to xyz from abc	"abc"	"xyz"	0	>0
as xyz abc n	"abc"	"xyz"	-1	>0
abc xyz	-	-	-	=0
"from" to "to"	"from"	"to"	0	>0

Figure 6.3: `rdargs("FROM/A,TO=AS/K,N/S", argv, 50)`

n `:= findarg(`*keys*`,` *item*`)`

The function `findarg` was primarily designed for use by `rdargs` but since it

is sometimes useful on its own, it is publicly available. Its first argument, *keys*, is a string of keys of the form used by `rdargs` and item is a string. If the result is positive, it is the argument number of the keyword that matches item, otherwise the result is `-1`.

n `:= str2numb(`*str*`)`

This function converts the string *str* into an integer. Characters other than `0` to `9` and `-` are ignored.

n `:= randno(`*upb*`)`

This function returns a random integer in the range `1` to `upb`. Its implementation is as follows:

```
STATIC { seed=12345 }

LET randno(upb) = VALOF
{ seed := seed*2147001325 + 715136305
  RETURN ABS(seed/3) REM upb + 1
}
```

oldseed `:= setseed(`*newseed*`)`

The current seed can be set to *newseed* by the call `setseed(`*newseed*`)`. This function returns the previous seed value.

obj `:= mkobj(`*upb*, *fns*, *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*`)`

This function creates and initialises an object. Its definition is as follows:

```
LET mkobj(upb, fns, a, b, c, d, e, f, g, h, i, j, k) = VALOF
{ LET obj = getvec(upb)
  UNLESS obj=0 DO
  { !obj := fns
    InitObj#(obj, @a) // Send the init message to the object
  }
  RETURN obj
}
```

As can be seen, it allocates a vector for the fields of the object, initialises its zeroth element to point to the methods vector and calls the initialisation method that is expected to be in element `InitObj` of *fns*. The result is a pointer to the initialised fields vector. If it fails, it returns zero. As can be seen the initialisation method receives a vector with up to 11 arguments.

6.3.10 Program Loading and Control

In this implementation, the BCPL compiler generates a file of hexadecimal numbers for the compiled code. For instance the compiled form of the `logout` command:

```
SECTION "logout"
GET "libhdr"
LET start() BE abort(0)
```

is

```
000003E8 0000000C
0000000C 0000FDDF 474F4C07 2054554F 0000DFDF
61747307 20207472 7B1F2310 00000000 00000001
0000001C 0000001F
```

The first two words indicate the presence of a "hunk" of code of size 12(`000000C`) words which then follow. The first word of the hunk (`000000C`), is again the length. The next two words (`0000FDDF` and `474F4C07`) contain the `SECTION` name `"logout"`. These are followed by the two words `0000DFDF` and `61747307` which identify the procedure name `"start"`. The body of `start` is compiled into one word (`7BF1F2310`) which correspond to the Cintcode instruction:

`L0`	Load A with 0
`K3G 31`	Call the function in global 31, incrementing the stack by 3
`RTN`	

The remaining 4 words contains global initialisation data indicating that global 1 is to be set to the entry point at position 28 (`0000001C`) relative to the start of the hunk, and that the highest referenced global number is 31 (`0000001F`).

code `:= start(`*a1*`, `*a2*`, `*a3*`, `*a4*`)`

This function is, by convention, the main procedure of a program. If it is called from the command language interpreter (see section 8), its first argument is zero and its result should be the command completion code; however, if it is the main procedure of a module run by `callseg`, defined below, then it can take up to 4 arguments and its result is up to the user. By convention, a command completion code of zero indicates successful completion and larger numbers indicate errors of ever greater severity

`clihook()`

This procedure is defined in BLIB and simply calls `start`. Its purpose is to

assist debugging by providing a place to set a breakpoint in the command language interpreter (CLI) just before a command in entered. It is also permissible for the user to override the standard definition of clihook with a private version.

`stop(`*code*`)`

This function is provided to stop the execution of the current command running under control of the CLI. Its argument *code* is the command completion code.

count`:= instrcount(`*fn*`,`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`)`

This procedure returns the number of Cintcode instruction executed when evaluating the call: *fn*`(`*a*`,`*b*`,`*c*`,`*d*`,`*e*`,`*f*`,`*g*`,`*h*`,`*i*`,`*j*`,`*k*`)`.

Counting starts from the first instruction of the body of *fn* and ends when its final `RTN` instruction is executed. Thus when `f` was defined by `LET f(x) = 2*x+1`, the call `instrcount(f, 10)` returns 4 since its body executes the four instructions: `L2; MUL; A1; RTN`.

`abort(`*code*`)`

This procedure causes an exit from the current activation of the interpreter, returning *code* as the fault code. If *code* is zero execution leaves the Cintcode system altogether, if *code* is `-1` execution resumes using the faster version of the interpreter (`cintasm`). If *code* is positive, under normal conditions, the interactive debugger is entered.

flag `:= intflag()`

This function provides a machine dependent test to determine whether the user is asking to interrupt the normal execution of a program. On the Apple Macintosh, `flag` will be set to `TRUE` only if the COMMAND, OPTION and SHIFT keys are simultaneously pressed.

segl `:= loadseg(`*name*`)`

This function loads the compiled program into memory from the specified file name. It returns the list of loaded program modules if loading was successful and zero otherwise. It does not initialise the global variables defined in the program.

res `:= globin(`*segl*`)`

This function initialises the global variables defined in the list of program modules given by its argument *segl*. It returns zero if the global vector was too small, otherwise it returns *segl*.

`unloadseg(`*segl*`)`

This routine unloads the list of loaded program modules given by *segl*.

res := callseg(*name*,*a1*,*a2*,*a3*,*a4*)

This function loads the compiled program from the file *name*, initialises its global variables and calls start with the four arguments *a1*,...,*a4*. It returns the result of this call, after unloading the program.

6.3.11 Character Handling

ch := capitalch(*ch*)

This function converts lowercase letters to uppercase, leaving other characters unchanged.

res := compch(*ch1*, *ch2*)

This function compares two characters ignoring case. It yields -1 (+1) if *ch1* is earlier (later) in the collating sequence than *ch2*, and 0 if they are equal.

res := compstring(*s1*, *s2*)

This function compares two strings ignoring case. It yields -1 (+1) if *s1* is earlier (later) in the collating sequence than *s2*, and 0 if they the strings are equal.

6.3.12 Coroutines

BCPL uses a stack to hold function arguments, local variables and anonymous results, and it uses the global vector and static variables to hold non-local quantities. It is sometimes convenient to have separate runtime stacks so that different parts of the program can run in pseudo parallelism. The coroutine mechanism provides this facility.

In this implementation, they have distinct stacks but share the same global vector, and it is natural to represent a coroutine by a pointer to its stack. At the base of each stack there are six words of system information as shown in figure 6.4.

The resumption point is the P-pointer belonging to the procedure that caused the suspension of the coroutine. It becomes the value of the P-pointer when the coroutine next resumes execution. The parent link points to the coroutine that called this one, or is zero if the coroutine not active. The outermost coroutine (or *root coroutine*) is marked by the special value -1 in its parent link. As a debugging aid, all coroutines are chained together in a list held in the global

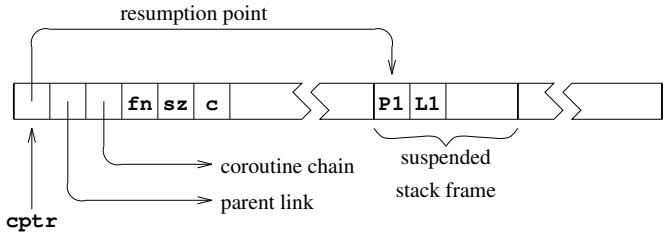


Figure 6.4: A coroutine stack

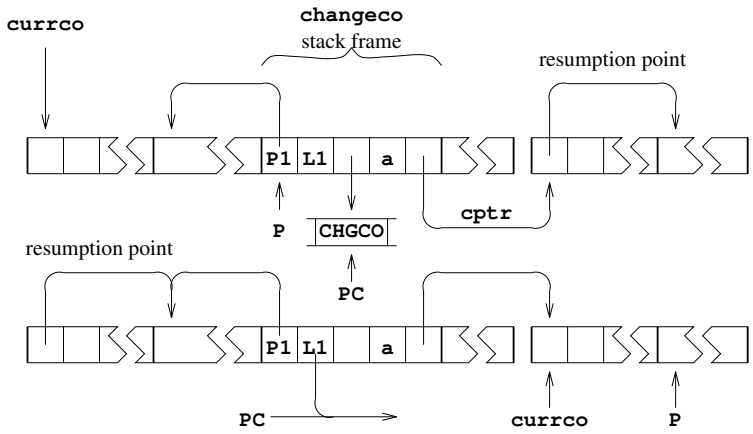


Figure 6.5: The effect of `changeco(a, cptr)`

`colist`. The values `fn` and `sz` hold the main function of the coroutine and its stack size, and `c` is a private variable used by the coroutine mechanism.

At any time just one coroutine (the *current coroutine*) has control, and all the others are said to be suspended. The current coroutine is held in the global variable `currco`, and the P pointer points to a stack frame within its stack. Passing control from one coroutine to another involves saving the resumption point in the current coroutine, and setting new values for the program counter (PC), the P pointer and `currco`. This is done by `changeco(a,cptr)` as shown in figure 6.5. The function `changeco` is defined by hand in `SYSLIB` and its body consists of the single instruction `CHGCO` and as can be seen its effect is somewhat subtle. The only uses of `changeco` are in the definitions of `createco`, `callco`, `cowait` and `resumeco`, and its effect, in each case, is explained below. The only functions that can cause coroutine suspension are `callco`, `cowait` and `resumeco`.

res `:= callco(`*cptr*`,` *arg*`)`

This call suspends the current coroutine and transfers control to the coroutine pointed to by *cptr*. It does this by resuming execution of the function that caused its suspension, which then immediately returns yielding *arg* as result. When

`callco(`*cptr*`,`*arg*`)` next receives control it yields the result it is given.

res `:= cowait(`*arg*`)`

This call suspends the current coroutine and returns control to its parent by resuming execution of the function that caused its suspension, yielding *arg* as result. When `cowait(`*arg*`)` next receives control it yields the result it is given.

res `:= resumeco(`*cptr*`,` *arg*`)`

The effect of `resumeco` is almost identical to that of `callco`, differing only in the treatment of the parent. With `resumeco` the parent of the calling coroutine becomes the parent of the called coroutine, leaving the calling coroutine suspended and without a parent. Systematic use of `resumeco` reduces the number of coroutines having parents and hence allows greater freedom in organising the flow of control between coroutines.

cptr `:= createco(`*fn*`,` *size*`)`

This function creates a new coroutine leaving it suspended in the call of `cowait` in the following loop.

```
c := fn(cowait(c)) REPEAT
```

When control is next transfered to the new coroutine, the value passed becomes the result of `cowait` and hence the argument of `fn`. If `fn(..)` returns normally, its result is assigned to `c` which is returned to the parent coroutine by the repeated call of `cowait`. Thus, if `fn` is simple, a call of the coroutine will convert the value passed, `val` say, into `fn(val)`. However, in general, `fn` may contain calls of `callco`, `cowait` or `resumeco`, and so the situation is not always quite so simple.

In detail, the implementation of `createco` uses `getvec` to allocate a vector with upper bound `size+6` and initialises its first seven locations ready for the call of `changeco(0,c)` that follows. The state just after this call is shown in figure 6.6. Notice that `cowait(c)` is about to be executed in the environment of the new coroutine, and that this call will cause a return from `createco` in the original coroutine, passing back a pointer to the new coroutine as a result.

`deleteco(`*cptr*`)`

This call takes a coroutine pointer as argument and, after checking that the corresponding coroutine has no parent, deletes it by returning its stack to free store.

cptr `:= initco(`*fn*`,` *size*`,` *a*`,...)`

This function is defined in BLIB to provide a convenient method of creating

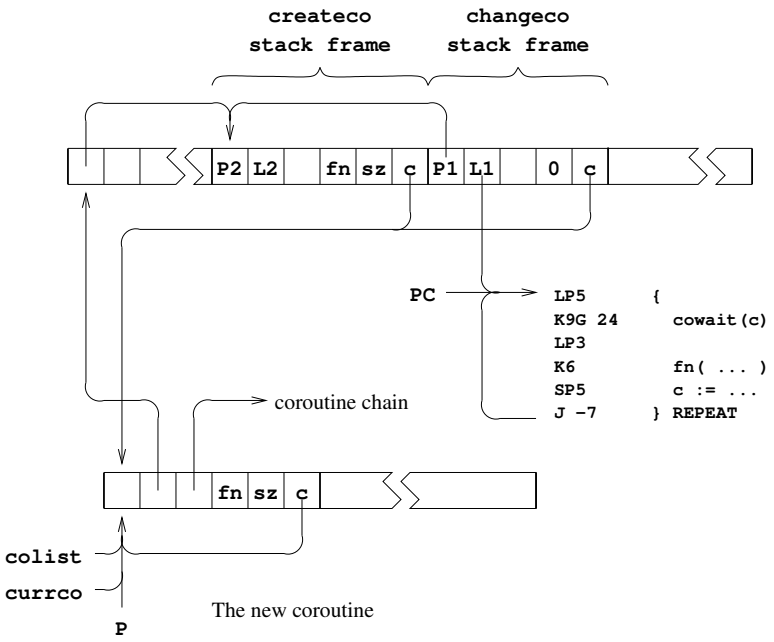


Figure 6.6: The state just after `changeco(0,c)` in `createco`

and intialising coroutines. It definition is as follows:

```
LET initco(fn, size, a, b, c, d, e, f, g, h, i, j, k) = VALOF
{ LET cptr = createco(fn, size)
  UNLESS cptr=0 DO callco(cptr, @a)
  RESULTIS cptr
}
```

A coroutine with main function *fn* and given size is created and, if successful, it is initialised by `callco(`*cptr*`, @`*a*`)`. Thus, *fn* should expect a vector containing up to 11 values. Once the coroutine has initialised itself, it should return control to `initco` by means of a call of `cowait`. Examples of the use of `initco` can be found in the example that follows.

6.3.13 Hamming's Problem

A following problem permits a neat solution involving coroutines.

> Generate the sequence `1,2,3,4,5,6,8,9,10,12,...` of all numbers divisible by no primes other than 2, 3, or 5".

This problem is attributed to R.W.Hamming. The solution given here shows how data can flow round a network of coroutines. It is illustrated in figure 6.7 in which each box represents a coroutine and the edges represent `callco/cowait` connections. The end of a connection corresponding to `callco` is marked by `c`, and end

corresponding to `cowait` is marked by `w`. The arrows on the connections show the direction in which data moves. Notice that, in `tee1`, `callco` is sometimes used for input and sometimes for output.

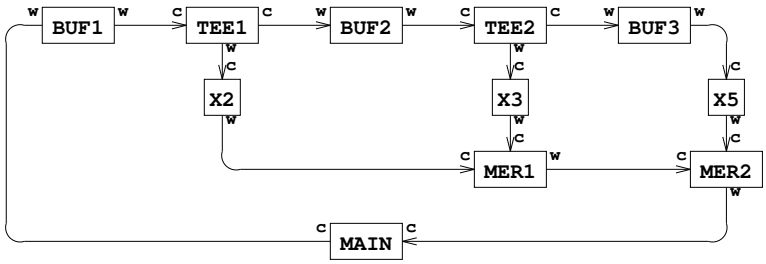


Figure 6.7: Coroutine data flow

The coroutine `buf1` controls a queue of integers. Non-zero values can be inserted into the queue using `callco(buf1,val)`, and values can be extracted using `callco(buf1,0)`. The coroutines `buf2` and `buf3` are similar. The coroutine `tee1` is connected to `buf1` and `buf2` and is designed so that `callco(tee1)` will yield a value extracted from `buf1`, after sending a copy of it to `buf2`. `tee2` similarly takes values from `buf2` passing them to `buf3` and `x3`. Values passing through `x2`, `x3` and `x5` are multiplied by 2, 3 and 5, repectively. `mer1` merges two monotonically increasing streams of numbers produced by `x2` and `x3`. The resulting stream is then merged by `mer2` with the stream produced by `x5`. The stream produced by `mer2` is the required Hamming sequence, each value of which is printed by `main` and then inserted into `buf1`.

The BCPL code for this solution is as follows:

```
GET "libhdr"

LET buf(args) BE    // Body of BUF1, BUF2 and BUF3
{ LET p, q, val = 0, 0, 0
  LET v = VEC 200

  { val := cowait(val)
    TEST val=0 THEN { IF p=q DO writef("Buffer empty*n")
                      val := v!(q REM 201)
                      q := q+1
                    }
               ELSE { IF p=q+201 DO writef("Buffer full*n")
                      v!(p REM 201) := val
                      p := p+1
                    }
  } REPEAT
}
```

```
LET tee(args) BE    // Body of TEE1 and TEE2
{ LET in, out = args!0, args!1
  cowait()          // End of initialisation.

  { LET val = callco(in, 0)
    callco(out, val)
    cowait(val)
  } REPEAT
}

AND mul(args) BE    // Body of X2, X3 and X5
{ LET k, in = args!0, args!1
  cowait()          // End of initialisation.

  cowait(k * callco(in, 0)) REPEAT
}

LET merge(args) BE  // Body of MER1 and MER2
{ LET inx, iny = args!0, args!1
  LET x, y, min = 0, 0, 0
  cowait()          // End of initialisation

  { IF x=min DO x := callco(inx, 0)
    IF y=min DO y := callco(iny, 0)
    min := x<y -> x, y
    cowait(min)
  } REPEAT
}

LET start() = VALOF
{ LET BUF1 = initco(buf,   500)
  LET BUF2 = initco(buf,   500)
  LET BUF3 = initco(buf,   500)
  LET TEE1 = initco(tee,   100, BUF1, BUF2)
  LET TEE2 = initco(tee,   100, BUF2, BUF3)
  LET X2   = initco(mul,   100,    2, TEE1)
  LET X3   = initco(mul,   100,    3, TEE2)
  LET X5   = initco(mul,   100,    5, BUF3)
  LET MER1 = initco(merge, 100,   X2,   X3)
  LET MER2 = initco(merge, 100, MER1,   X5)

  LET val = 1
  FOR i = 1 TO 100 DO { writef(" %i6", val)
                        IF i REM 10 = 0 DO newline()
                        callco(BUF1, val)
                        val := callco(MER2)
                      }

  deleteco(BUF1); deleteco(BUF2); deleteco(BUF3)
  deleteco(TEE1); deleteco(TEE2)
  deleteco(X2); deleteco(X3); deleteco(X5)
  deleteco(MER1); deleteco(MER2)
  RESULTIS 0
}
```

6.3.14 Scaled Arithmetic

The library function `muldiv` makes full precision scaled arithmetic convenient.

res `:= muldiv(`*a*`, `*b*`, `*c*`)`

The result is the value obtained by dividing *c* into the double length product of *a* and *b*. The remainder of this division is left in the global variable `result2`. The result is undefined if it is too large to fit into a single length word or if *c* is zero. In this implementation, the result is also undefined if any of *a*, *b* or *c* is the largest negative integer. As an example, the function defined below calculates the cosine of the angle between two unit vectors in three dimensions using scaled integers to represent numbers with 6 digits after the decimal point.

```
MANIFEST { Unit=1000000 } // Scaling factor for numbers of the
                          // form ddd.dddddd

FUN inprod(v, w) = muldiv(v!0, w!0, Unit) +
                   muldiv(v!1, w!1, Unit) +
                   muldiv(v!2, w!2, Unit)
```

On some processors, such as the Pentium, `muldiv` can be encoded very efficiently in assembly language.

Chapter 7

Streams

As has been seen in the previous chapter, streams provide a convenient method of obtaining device independent input and output. Information needed to process a stream is held in its stream control block (SCB) whose fields are given in the following table.

Field	Purpose
`scb.id`	Specifies the direction: in, out or inout
`scb.type`	Type of stream: terminal, file, mailbox, etc
`scb.task`	The task, if any, to handle this stream
`scb.buf`	The buffer for this stream
`scb.pos`	Position in `buf` of next character to transfer
`scb.end`	Position of end of valid data in `buf`
`scb.rdfn`	Function to refill the buffer
`scb.wrfn`	Function to write out the buffer
`scb.endfn`	Function to close a stream
`scb.block`	The block number in a disc file
`scb.write`	True when data in `buf` need to be written out
`scb.bufend`	The size of `buf` in bytes
`scb.lblock`	The number of the last block of a disc file
`scb.ldata`	The number of data bytes in the last block of a disc file
`scb.blength`	The size in bytes of a disc block
`scb.reclen`	The number of bytes in a record (for some files)
`scb.fd`	The file descriptor for this stream

Throughout this section `id`, `type`, `task`, `buf`, `pos` and so on will be used represent to the values of the corresponding SCB fields.

The field `id` has value `id.inscb`,`id.outscb` or `id.inoutscb` specifying respectively whether the stream is for input, output or both.

The type field holds the type of the stream. The possible values are given in the following table.

Type	Purpose
`scbt.file`	A disc file
`scbt.ram`	A stream entirely held in RAM
`scbt.console`	A stream controlled by the console handler
`scbt.mbx`	A mailbox stream
`scbt.tcp`	A stream using TCP/IP

Console and TCP streams have negative types for which output to the device is triggered by the newline character `'*n'` and certain other control characters (`'*p'`, `'*c'`, `'*e'`). The use of `'*e'` to flush output is deprecated and will, in due course, be replaced by calls of `flushstream(scb)`.

The task field is either zero or specifies the task that controls the device associated with the stream. Currently there are four resident stream tasks: `COHAND` for the keyboard and screen, `FH0` for the main disc filing system, `MBXHAND` for mailbox streams, and `TCPHAND` for TCP/IP streams.

The field `buf` is either zero or points to a buffer of length `buflen` bytes. The buffer is often, but not always, allocated by `getvec` when the stream was opened, and is freed in a stream dependent way when closing the stream by calling the stream function in `endfn`.

The fields `rdfn`, `wrfn` and `endfn` contain either zero or stream dependent functions to put data into the buffer, extract data from the buffer, or close down the stream, respectively. These functions are set up at the time the stream is opened. They each take the stream control block as argument and return `TRUE` if the operation was successful. On error, the result is `FALSE` and an error code is placed in `result2`. For input streams, a false result with `result2=0` indicates that end of file has been reached. These functions typically update several SCB fields, typically including `pos` and `end`.

The bytes in a stream buffer between `buf%0 ... buf%(end-1)` represent valid data that has either been recently read in from a device, or is being prepared for output to a device. If `0<=pos<end` then `buf%pos` refers to the location of the next byte to be read (by `rdch`) or written (by `wrch`). If `pos>=end`, then either `end` must be advanced, or more data must be read from or written to the device. Input streams can be stepped back using `unrdch` provided `pos` is not zero.

For output streams, data is normally appended to the end of the file and both `pos` and `end` advanced together. These pointers cannot become larger than `buflen`, the size of the buffer. When the buffer becomes full, data is extracted

from it and the pointers reset. If the file position is changed (using `point`) then output may be placed in the middle of the file and only `pos` is advanced.

For disc files, if the buffer currently holds data from the last block of a file then `block=lblock` and `end` points just beyond the last byte of the file. The number of bytes in the last disc block of a file is held in `ldata`. As a file is being extended `end` is typically larger than `ldata` and only synchronised when the last block is written to disc. Normally `ldata` needs correcting when the last block of a file is written to disc.

For input operations, if `pos>=end` the stream is either exhausted or more data must be read into the buffer. For any stream, the condition `end=-1` indicates that no more data can be transferred to or from the stream. Reading from such a stream will yield `endstreamch`.

The functions `note` and `point` can be used with any stream but only have any effect on streams connected to disc files. The call `note(scb, posv)` will place `block` and `pos` in the first two elements of `posv`. The call `point(scb, posv)` will set `block` and `pos` for stream `scb` from the first two elements of `posv`. This sometimes causes the current block to be written and the buffer reloaded with the contents of another disc block. Clearly, subsequent read or write operations will use (and change) the newly specified position.

Some disc files are treated as a sequence of fixed length records. For such files the record length is held in `reclen` which can be set by a call of `setrecordlength`. The functions `recordnote` and `recordpoint` can be used to control the position in a record file.

The following SCB fields are (probably) obsolete and will be removed.

Field	Purpose
`scb.devtype`	Used only in termroot.bpl
`scb.fab`	File access block (not used)
`scb.rab`	Record access block (not used)
`scb.name`	Name of stream

7.1 Implementation of `rdch` and `unrdch`

Characters are read using `rdch` whose definition is as follows:

```
LET rdch() = VALOF
{ LET pos = cis!scb.pos
  IF pos < cis!scb.end DO { LET ch = cis!scb.buf%pos
                            cis!scb.pos := pos+1
                            RESULTIS ch
                          }
  UNLESS replenish(cis) RESULTIS endstreamch
} REPEAT
```

If a byte is available in the buffer, it is returned after incrementing `pos`, otherwise an attempt is made to replenish the buffer by calling `replenish(cis)`. If this fails `endstreamch` is returned, otherwise `rdch` is re-entered and this time it will be successful.

After a call of `rdch` that it did not yield `endstreamch`, `pos>1` and the stream will be capable able to be backed up by at least one position using `unrdch` whose definition is as follows.

```
AND unrdch() = VALOF
{ LET pos = cis!scb.pos
  IF pos<=0 RESULTIS FALSE
  cis!scb.pos := pos-1
  RESULTIS TRUE
}
```

This function in used in functions like `readn` when a character must be put back into the input stream. A call of `unrdch` returns `TRUE` if the backup was successful and `FALSE` otherwise. Repeated calls of `unrdch` can backup the stream to the start the current buffer, but this is not often useful.

The function `replenish` is only called from `rdch` when `pos>=end`. Its definition is as follows:

```
AND replenish(scb) = VALOF
{ LET rdfn = scb!scb.rdfn
  result2 := 0
  IF rdfn & rdfn(scb) RESULTIS TRUE
  scb!scb.pos, scb!scb.end := 0, 0
  RESULTIS FALSE
}
```

It attempts to replenish the buffer by calling the function held in `rdfn`, but if `rdfn=0` the stream is taken to be exhausted. The result is `TRUE` if the buffer was successfully replenished with at least one new byte. It otherwise returns `FALSE` and a error code in `result2`. The condition `result2=0` indicates that the stream was exhausted. A non zero value indicate an error.

7.2 Implementation of wrch

Bytes can be written to the currently selected output stream by calling wrch whose definition is as follows.

```
AND wrch(ch) = VALOF
{ LET pos    = cos!scb.pos

  IF pos >= scb!scb.bufend DO
  { UNLESS deplete(scb} RESULTIS FALSE
    pos := cos!scb.pos
  }

  cos!scb.buf%pos := ch
  pos := pos+1
  cos!scb.pos := pos
  IF cos!scb.end < pos DO cos!scb.end := pos
  cos!scb.write := TRUE

  UNLESS cos!scb.type<0 & ch<'*s' RESULTIS TRUE

  IF ch='*n' | ch='*e' | ch='*p' | ch='*c' RESULTIS deplete(cos)
  RESULTIS TRUE
}
```

If pos>=bufend the buffer must be depleted before the write operation can be done. If this is successful there will be room in the buffer for at least on byte. The byte can now be placed in the buffer and pos advanced. It is often necessary to advance end as well. The write flag is then set to indicate that new data is present. For interactive streams certain control characters, typically '*n', trigger depletion of the completed line. The result is TRUE if wrch is successful, and FALSE otherwise.

The function deplete is only called from wrch and closestream. It causes the data between buf%0 and buf%(end-1) to be written out. The definition of deplete is as follows.

```
AND deplete(scb) = VALOF
{ LET wrfn = scb!scb.wrfn
  IF wrfn & wrfn(scb) RESULTIS TRUE
  scb!scb.pos, scb!scb.end, scb!scb.write := 0, 0, FALSE
  RESULTIS FALSE
}
```

If the write function wrfn exists and its call is successful, deplete returns TRUE. Otherwise, the buffered data is thrown away and the buffer pointers reset.

7.3 Console streams

A console stream has type `scbt.console` and is typically connected to a keyboard or a screen. Communication is controlled by the console handler task `COHAND`. Console streams are interactive causing data transmission to be triggered by `'*n'` and certain control characters. Data is can only be read from a console input stream when a complete line is available.

At any time `COHAND` is connected to a particular client task called the client. As it builds a line for the client, `COHAND` performs simple line editing and other control operations as shown in the following table.

Control sequence	Meaning
backspace or rubout	Remove latest character from current line
ctrl-a	Cause the client task to enter hold state
ctrl-b	Set flag bit 0 in the client task
ctrl-c	Exit from the Cintpos system
ctrl-d	Set flag bit 3 in the client task
ctrl-e	Set flag bit 4 in the client task
`@f`	Throw away all lines pending for the client task
`@l`	Throw away the line currently being built
`@q`	Insert an end of file mark
`@z`	Disable echoing of the current line
`@`*ooo*	Insert character with octal code 000
`@x`*hh*	Insert character with hex code *hh*
`@s`*dd*	Make task *dd* the current client
`@t`*dd*	Make task *dd* the current client and disable output from other tasks

The maximum line length for console input is 128 characters, and output lines have a limit of 1024. When a console stream is created, it has no buffer and the fields `pos`, `end`, `buf` and `buflen` are all set to zero. Stream buffers are allocated and deallocated on demand on a line by line basis. For console input streams, successive calls of `unrdch` can 'unread' to the start of the current line.

7.4 File streams

A file stream has type `scbt.file` and is controlled by the file handler task `FH0`. When such a stream is opened the fields `blength` and `buflen` are set to the block size which is typically 4096 bytes, and a buffer of this size allocated. For input and inout streams the first disc block of the file is then read into the buffer, and

block and pos set to 1 and 0, respectively. The fields lblock and ldata are also set to reflect the size of the file. For output streams, the file size is initially zero and so block=lblock=1 and pos=end=ldata=0. For any file stream, write is initially FALSE, and only changes when new data is placed into the buffer.

All low level file operations are performed within FH0. These include openning and closing files, setting the read/write position of a file, and reading and writing data between a disc block and the buffer. Reading a disc block is performed by the call: fh0getbuf(scb) and writing by the call: fh0putbuf(scb). They both use block to calculate where to position the file before the read or write operation. The low level file positioning in fh0getbuf and fh0putbuf is performed by the following code.

```
UNLESS sys(Sys_seek, scb!scb.fd, offset) RESULTIS FALSE
```

Both fh0getbuf and fh0putbuf return FALSE if there is a position failure. If fh0putbuf is writing to the last block of a file, ldata is updated appropriately.

The user can discover and set the position in a file stream using the calls note(scb, posv) and point(scb, posv) where scb is the pointer to the stream control block and posv is a two element vector representing the position within the stream. When note is called it sends a packet to FH0 containing scb and posv. This, in turn, calls fh0note which simply copies the current values of block and pos into the first two elements of posv. Its definition is as follows.

```
AND fh0note(scb, posv) = VALOF
{ posv!0 := scb!scb.block
  posv!1 := scb!scb.pos
  RESULTIS TRUE
}
```

Setting the stream position is more complicated since it may involve writing the buffer out to the current disc block and reading data from another block.

Again, `point` sends a packet to `FH0` which in turn calls `fh0point` to do the work.

```
AND fh0point(scb, posv) = VALOF
{ LET blkno  = posv!0
  LET pos    = posv!1
  LET id     = scb!scb.id
  LET block  = scb!scb.block
  LET lblock = scb!scb.lblock
  LET end    = scb!scb.end

  // Check the stream is suitable
  UNLESS scb!scb.type=scbt.file &
         (id=id.inscb | id=id.inoutscb) RESULTIS FALSE

  IF pos=0 & blkno=lblock+1 DO blkno, pos := lblock, buflen

  IF blkno<=0 |              // Perform safety check
     blkno>lblock |
     blkno=lblock & pos > (block=lblock->end, scb!scb.ldata) DO abort(5001)

  IF blkno=block DO { scb!scb.pos := pos; RESULTIS TRUE }

  // The move is to a different block,
  // so check whether this block must be written
  IF scb!scb.write UNLESS fh0putbuf(scb) DO abort(5001)

  scb!scb.block := blkno
  UNLESS fh0getbuf(scb) DO abort(5001)

  // Safety check
  UNLESS end=buflen |
         blkno=lblock & end=scb!scb.ldata DO abort(5001)

  scb!scb.pos := pos
  RESULTIS TRUE
}
```

In this implementation, it is only permissable to set the stream pointer in disc file streams that allow read operations. The stream pointer is also restricted to be between the start and end of the file. If the new position is within the current block the it is only necessary to change the `pos` field in the SCB. If the position is to a different block it may be necessary to write the current block to disc befor reading the new block in. Note that `fh0point` cautiously performs several safety checks.

7.5 Mailbox streams

Mailbox streams are based loosely on similar streams available the VAX VMS system. A mailbox is a globally named FIFO queue of variable length records that typically consist of text terminated by newlines. Cintpos tasks can write and

read such records to a mailbox on a first come first served basis. In the current implementation records cannot exceed 1024 bytes in length and the maximum size of a mailbox FIFO buffer is 4096 bytes. Writing a record to a mailbox is blocked if there is insufficient room in the buffer for it, and reading is blocked if the mailbox is empty.

Mailboxes have names starting with `"MBX:"`, for example: `"MBX:box1"`. The functions `findoutput` and `findinput` are used to connect streams to mailboxes. On the first such call an new empty mailbox will be created. If the named mailbox already exists, it will be attached to the newly created stream. A count is maintained of how many open streams are connected to a mailbox. If the count reaches zero and the mailbox is empty, it is deleted.

The mailbox mechanism is implemented using a handler task called `MBXHAND`. This has an interface with the stream system very similar to that used by `COHAND` and `FH0`. `MBXHAND` maintains a simple linked list of existing mailbox. Each mailbox has a control block in the list with the following fields.

Field	Purpose
`mbx.link`	Zero or a pointer to the next mailbox
`mbx.rp`	Position of the next record to read
`mbx.wp`	Position of the next record to write
`mbx.refcount`	Count of the number of streams referring to this mailbox
`mbx.rdq`	List of blocked read request packets
`mbx.wrq`	List of blocked write request packets
`mbx.namebase`	Base of space to hold the mailbox name
`mbx.bufbase`	Base of the 4096 byte FIFO buffer

The FIFO is implemented as a circular buffer using `rp` and `wp` as the pointers to the next bytes to read or write. The FIFO is empty when `rp=wp`, and full when `rp+4095` equals `wr` modulo 4096. The buffer always contains a queue of complete records.

If `MBXHAND` receives a write request packet whose record is too large to currently fit into the FIFO then the packet is placed at the end of `wrq`. Also, any write packet received when `wrq` is non empty is blocked and appended to the list, even though there may be room for it record in the FIFO. This preserves the necessary ordering of the records. When a read request extracts a record from the FIFO, the first packet on `wrq` is inspected to see if its record will now fit. Note that one read of a large record may causes several writes to complete.

Similarly, blocked read request packets are held in `rdq` in the order in which they were received. They are released as write requests are received.

The current implementation limits the total length of a mailbox name to less than 31 characters.

7.6 TCP streams

TCP streams may be opened using `findinput` and `findoutput` using stream names starting with `TCP:` or `NET:`. Such names contain host and port names separated by a colon. Some examples are as follows:

```
TCP:spice:echo
TCP:127.0.0.1:9050
TCP::9000
NET:shep.cl.cam.ac.uk:9200
```

As can be seen, both hosts and ports may be specified by name or numerically. If the port is omitted, port 9000 is used by default. If the host is specified, both `findinput` and `findoutput` will try to establish as TCP connection with the given port on the specified machine. The stream will not be opened if the connection is refused. If the host is omitted, the the stream will be opened but not be able to transfer data until a connection via the specified local port is established by another machine. Read and write requests will be blocked until this happens.

`TCPHAND` maintains a list of connection identified by the IP address and port number. If stream is opened with an IP address and port number that already exists in the list, then its SCB will reference the existing connecting. Multiple input and output streams may thus use the same connection. Sequences of read and write requests using different streams but the same connection will be processed in the order in which they are received by `TCPHAND`. A connection is made when the first stream refering to it is opened, and delete when the last stream refering to it is closed.

Streams with names starting with `TCP:` are interactive having data transmission typically triggered by `'*n'`. Such streams are normally used for communication with slow serial devices such as terminals and printers. For applications requiring rapid transmission of large volumes of data it is preferable to use streams with names starting with `NET:` since these typically transmit data in blocks of 4096 bytes. For both kind of streams, output can be triggered using `flushstream` if necessary.

The use of several streams attached to the same connection is analogous to the mailbox mechanism and is normally only useful with interactive streams.

Both `TCP:` and `NET:` streams are implemented using the same handler task called `TCPHAND`. This task has an interface with the stream system similar to

that used by COHAND and FHO. TCPHAND maintains a simple linked list of existing tcp connection. Each each connection has a control block in the list with the following fields.

Field	Purpose
tcp.link	Zero or a pointer to the next connection
tcp.ipaddr	The IP address of this connection
tcp.port	The post associated with this connection
tcp.refcount	Count of the number of streams referring to this mailbox
tcp.rdq	List of blocked read request packets
tcp.wrq	List of blocked write request packets
tcp.sock	Zero or the socket for this connection
tcp.namebase	Base of space to hold the stream name

7.7 RAM streams

A RAM stream is one which is only used for input and for which all the data is preloaded into its buffer. The length of the buffer is held in buflen and end, and pos is initially zero. The stream function rdfn is set to zero, so when all the buffered data has been read rdch will yield endstreamch. There is no need for an end function (endfn) since the buffer is appended to the stream control block and so freeing the SCB also frees the buffer. Although wrfn should never be used, it is also set to zero. A typical use of a RAM stream is in the implementation of the run command. One could also have been used in the implementation of the c command.

Chapter 8

The Command Language

The Command Language Interpreter (CLI) is a simple interactive interface between the user and the system. It is implemented in BCPL and its source code can be found in `cintcode/sys/CLI.b`. It loads and executes previously compiled programs that are held either in the current directory or a directory specified by the shell variable `BCPLPATH`. The source of the system provided commands can be found in `cintcode/com`. These commands are described in below in Section 8.2. Since any compiled program can be regarded as a command, the command language can be extended easily by the user.

8.1 Bootstrapping

When the Cintcode System is started, control is passed to the interpreter which, after a few initial checks, allocates vectors for the memory of the cintcode abstract machine and the tally vector available for statistics gathering. The cintcode memory is initialised suitably for sub-allocation by `getvec`, which is then used to allocate space for the root node, the initial stack and the initial global vector. The initial state shown in figure 8.1 is completed by loading the object modules `SYSLIB`, `BLIB` and `BOOT`, and initialising the root node, the stack and global vector. Interpretation of cintcode instructions now begins with the Cintcode register `PC`, `P` and `G` set as shown in the figure, and `Count` set to `-1`. The other registers are cleared. The first Cintcode instruction to be executed is the first instruction of the body of the routine `start` defined in `sys/BOOT.b`. Since no return link has been stored into the stack, this call of `start` must not attempt to return in the normal way; however, its execution can still be terminated using `sys(Sys_quit,0)`.

The global vector and stack shown in figure 8.1 are used by `start` and form the running environment both during initialization and while running the debugger.

The CLI, on the other hand, is provided with a new stack and a separate global vector, thus allowing the debugger to use its own globals freely without interfering with the command language interpreter or running commands. The global vector of 1000 words is allocated for `CLI` and this is shared by the `CLI` program and its running commands. The stack, on the other hand, is used exclusively by the command language interpreter since it creates a coroutine for each command it runs.

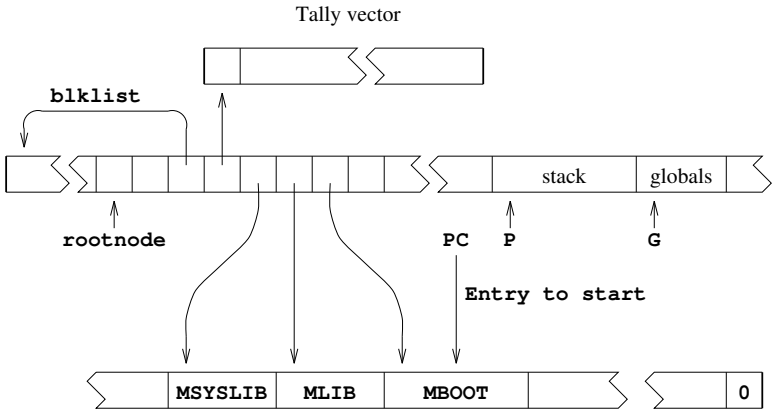


Figure 8.1: The initial state

Control is passed to the `CLI` by means of the call `sys(Sys_interpret,regs)` which recursively enters the intepreter from an initial Cintcode state specified by the vector `regs` in which that `P` and `G` are set to point to the bases of a new stack and a new global vector for `CLI`, respectively, PC is the location of the first instruction of `startcli`, and `count` is set to `-1`. This call of `sys(Sys_interpret,regs)` is embedded in the loop shown below that occurs at the end of the body of `start`.

```
{ LET res = sys(Sys\_interpret, regs)  // Call the interpreter
  IF res=0 DO sys(Sys\_quit, 0)
  debug(res)                 // Enter the debugger
} REPEAT
```

At the moment `sys(Sys_interpret,regs)` is first called, only `globsize`, `sys` and `rootnode` have been set in the CLI global vector and so the body of `startcli` must be coded with care to avoid calling global functions before their entry points have been placed in the global vector. Thus, for instance, instead of calling `globin` to initialise the globals defined in `SYSLIB` and `BLIB`, the following code is used:

```
sys(Sys\_globin, rootnode!rtn_syslib)
sys(Sys\_globin, rootnode!rtn_blib)
```

If a fault occurs during the execution of `CLI` or a command that it is running, the call of `sys(1,regs)` will return with the fault code and `regs` will hold the

dumped Cintcode registers. A result of zero, signifying successful completion, causes execution of the Cintcode system to terminate; however, if a non zero result is returned, the debugger is entered by means of the call `debug(res)`. Note that the Cintcode registers are available to the debugger since `regs` is a global variable. When `debug` returns, the `REPEAT`-loop ensures that the command language interpreter is re-entered. The debugger is briefly described in section 9.

On entry to `startcli`, the coroutine environment is initialised by setting `currco` and `colist` to point to the base of the current stack which is then setup as the root coroutine. The remaining globals are then initialised and the standard input and output streams opened before loading the `CLI` program by means of the following statement:

```
rootnode!rtn_cli := globin(loadseg("CLI"))
```

The command language interpreter is now entered by the call `start()`.

8.2 Commands

This section describes the commands whose source code can be found in `cintcode/com`. Each command is introduced by its name and `rdargs` argument format string.

`abort NUMBER`

The command: `abort` n calls the `BLIB` function `abort` with argument n. If $n = 0$, this will cause a successful return from the Cintcode system. If n is non zero, the interactive debugger is entered with fault code n. The default value for n is `99`. A brief description of the debugger is given in section 9.

```
bcpl    FROM/A,TO/K,VER/K,SIZE/K,TREE/S,NONAMES/S,
            D1/S,D2/S,OENDER/S,EQCASES/S,BIN/S:
```

This invokes the BCPL compiler. The `FROM` argument specified the name of the file to compile. If the `TO` argument is given, the compiler generates code to the specified file. Without the `TO` argument the compiler will just output the OCODE intermediate form to the file `OCODE`. This is used for compiler debugging and cross compilation. The `VER` argument redirects the standard output to a named file. The `SIZE` argument specified the size of the compiler's work space. The default is 40000 words. If the `NONAMES` switch is given the compiler will not include section and function names in the compiled code. These are only useful for debugging. The switches `D1` and `D2` control compiler debugging output. `D1` causes a readable form of the compiled cintcode to be output. `D2` causes a trace of the internal

working of the codegenerator to be output. `D1` and `D2` together causes a slightly more detailed trace of the internal working of the codegenerator to be output. `OENDER` causes code to be generated for a machine with the opposite endianess of the machine on which the compiler is running. `EQCASES` causes all identifiers to be converted to uppercase during compilation. This allows very old BCPL programs to be compiled. `BIN` causes the target cintcode to be in binary rather than ASCII encoded hexadecimal. This is primarily for Windows CE machines where reducing the size of code modules may be important.

`bcplxref FROM/A,TO/K,PAT/K`

This command outputs a cross reference listing of the program given by the `FROM` argument. This consists of a list of all identifiers used in the program each having a list of line numbers where the identifier was used and a letter indicating how the identifier was declared. The letters have the following meanings:

V	Local variable
P	Function or Routine
L	Label
G	Global
M	Manifest
S	Static
F	FOR loop variable

The `TO` argument can be used to redirect the output to a file, and the `PAT` argument supplies a pattern to restrict which names are to be cross referenced. Within a pattern an asterisk will match any sequence of characters, so the pattern `a*b*` will match identifiers such as `ab`, `axxb`or `axbyy`. Upper and lower case letters are equated.

`c` *command-file arguments*

The `c` command allows a file of commands to be executed as though they had just been typed in. The argument *command-file* gives the name of the file containing the command sequence.

Unless explicitly changed, the characters '`=`', '`<`', '`>`', '`$`' and '`.`' have special meanings within a command command. A dot '`.`' at the start of a line starts a directive which can specify the command command's argument format, or replace one of the special character with an alternative. There are six possible directives as follows:

.KEY or .K	*str*	argument format string
.DEFAULT or .DEF	*key value*	give *key* a default value
.BRA	*ch*	use *ch* instead of <
.KET	*ch*	use *ch* instead of >
.DOLLAR	*ch*	use *ch* instead of $
.DOT	*ch*	use *ch* instead of .

All directives must occur at the start of the command file. The .KEY directive specifies a format string of the form used by rdargs (see page 63) that describes what arguments can follow the command file name. The .DEFAULT directive specifies the default value that a specified key should have if the corresponding argument was omitted. The remaining directives allow the special characters to be changed.

The command sequence occurs after all the directives and may contain items of the form <*key*$*value*> or <*key*> where *key* is one of the keys in the format string and *value* is a default value. Such items are textually replaced by its corresponding argument or a default value. If $*value* is present, this overrides (for this item only) any default that might have been given by a .DEFAULT directive.

casech FROM/A,TO/A,DICT/K,U/S,L/S,A/S

This command systematically processes a BCPL program converting all reserved words to upper case and changing all identifiers to upper case (U), lower case (L, or in the form given by a specified dictionary (DICT).

checksum FROM/A,TO/K

This command calculates a check sum for the file specified by the FROM argument, sending the result to the file specified by the TO argument.

delete ,,,,,,,,,

This command will delete up to ten given files.

detab FROM/A,TO/K,SEP/K

This command copies the file give by the FROM argument to the file given by the TO argument replacing all tab characters by spaces. The tabs are separated by a distance specified by the SEP argument. The default is 8.

echo TEXT,N/S

This command will output its first argument TEXT, if given. The text will be followed by a newline unless the switch N is set.

`edit FROM/A,TO,WITH/K,VER/K,OPT/K`

This command is meant to provide a simple line editor. It used to run on the Tripos Portable Operating System but has not been modified to run on this system.

`fail CODE`

This command just returns to the CLI with a completion code given by `CODE`. The default code is 20.

`input TO/A,TERM/K`

This command will copy text from the current input sending it the the file specified by the `AS` argument. The input is terminated by a line starting with `/*` or the value of the `TERM` argument if given.

`interpreter FAST/S,SLOW/S`

This command allows the user to select the fast (`cintasm`) or the slow (`cinterp`) version of the interpreter. If no arguments are given the fast one is selected. It is implemented using `sys(0,-1)` or `sys(0,-2)` as described on page 48.

`join ,,,,,,,,,,,,,,,AS/A/K,CHARS/S`

This command will concatenate several files sending the result to the file specified by the `AS` argument. If the `CHARS` switch is given the files are treated as text files, otherwise they are copied in binary.

`logout`

This command causes an exit from the BCPL Cintcode System, typical returning to an operating system shell.

`map BLOCKS/S,NAMES/S,CODE/S,MAPSTORE/S,TO/K,PIC/S`

This command outputs the state of the Cintcode memory in a form that depends on the arguments given. The output goes to the screen unless a filename is given using the `TO` keyword.

`nlconv FILE,TOUNIX/S,TODOS/S,Q/S`

Thus command replaces the specified file with one in which line endings have been replaced by those appropriate for the destination system which is specified by the switches `TOUNIX` (the default) or Windows systems (`TODOS`). The `Q` argument quietens the command.

`prefix PREFIX,UNSET/S`

If the first argument is given, it becomes the current prefix string. If UNSET is specified, the prefix string is unset, and if no argument is given the current prefix is output. This command is implemented using `sys(32,`*prefix*`)` and `sys(33)` described on page 52. See also Section 6.3.6.

`preload ,,,,,,,,,`

This command will preload up to 10 commands into the Cintcode memory. Without arguments it outputs the list of preloaded commands. Preloading improves the efficiency of command execution and is also useful in conjunction with the `stats` command, see below.

`procode FROM,TO/K`

This command converts an OCODE (intermediate code) file specified by `FROM` to a more readable form. If `FROM` is missing it reads from the file `OCODE`. If the `TO` argument is missing it send the result to the screen.

`prompt PROMPT`

This command allows the user to change the prompt string. The prompt is output by the CLI using code of the form:

`writef(`*prompt*`,` *msecs*`)`

where *prompt* is the prompt format string and *msecs* is the time in milliseconds used by the previous command. The default prompt format is: `"%d> "`.

`raster COUNT,SCALE,TO/K,HELP/S`

This command controls the collection of rastering information but only works when the BCPL Cintcode system is running under the rastering interpreter `rasterp`. The implementation uses `sys(27,...)` calls that are described on page 52. If `raster` is given an argument it activates the rastering mechanism. Once rastering is activated information will be written to a raster data file for the duration of the next CLI command. The format of this file is also outlined on page 52.

The `COUNT` argument allows the user to specify how many Cintcode instructions to obey for each raster line. The default is 1000. The `SCALE` argument gives the raster line granularity in bytes per pixel. The default being 12. The `TO` argument specifies the name of the raster data file to be written. The default file name is `RASTER`.

If `raster` is called without any arguments, it closes the raster data file. The raster data file can be processed and converted to Postscript using the `rast2ps`

command described below. Typical use of the `raster` command is the following script:

```
raster count 1000 scale 12 to RASTER
bcpl com/bcpl.b to junk
raster
rast2ps fh 18000000 mh 301000
```

This will create the Postscript file `RASTER.ps` for the BCPL compiler compiling itself, similar to that shown in Figure 8.2.

```
rast2ps  FROM,SCALE,TO/K,ML,MH,MG,FL,FH,FG,
              DPI/K,INCL/K,A4/S,A3/S,A2/S,A1/S,A0/S
```

This command converts a raster data file (written using the `raster` command described above) into a postscript file suitable for printing. There are parameters to control the region to convert, the output paper size and other parameters. It is also possible to include annotations in the resulting picture.

The `FROM` parameter specifies the name of the raster data file. `RASTER` is the default. `SCALE` specifies a magnification as a percentage. The default is 80. The `TO` parameter specifies the name of the postscript file to be generated. `RASTER.ps` is the default. The parameters `ML` and `MH` specify the low and high limits of the address space to be processed. `MG` specifies the separation of the grid line on the memory axis. The defaults are `ML=0 MH=300100` and `MG=100000`. The units are in bytes. The parameters `FL` and `FH` specify the low and high limits of the instruction count axis to be processed. `FG` specifies the separation of the grid line on the memory axis. The defaults are `FL=0 FH=20000000` and `FG=1000000`. `DPI` specifies the approximate number of dots per inch used by the output device. The default is 300. `A`*n* specifies the output page size. The default is `A4`. The `INCL` parameter specifies the name of a file to be copied into the postscript file. The default is `psincl`. This file allows annotations to be made in the picture. The file `cintcode/psincl` was used to annotate the memory time graph shown in Figure 8.2. This file contains lines such as:

```
F2 setfont
(SYN) 1.1 35 2 PDL
(TRN) 8.1 30 1.7 PUL
(CG) 15.3 36 2.1 PUR
(GET Stream) 0.45 270 1.7 PUL
...
(OCODE Buffer) 13.9 245 2 PDR
% 8.5 150 MVT (HELLO WORLD) SC
F3 setfont
(Self Compilation of the Cintcode BCPL Compiler) TITLE
```

The postscript macros `PDL`, `PUL`, `PUR` and `PDR` draw arrows with specified labels, byte address, instruction count and arrow lengths. The arrow directions

are respectively: down left, Up left, up right and down right. The macro `MVT` moves to the specified position in the graph and `SC` draws a string centered at that position. The `TITLE` macro draws the graph title and `F2` and `F3` are fonts suitable for the labels and title. The resulting postscript file can, of course, be further edited by hand.

`rename FROM/A,TO=AS/A/K`

This will rename the file given by `FROM` to that specified by the `AS` argument.

`stack SIZE`

The command `stack` n causes the size of the coroutine stack allocated for subsequent commands to be n words long. If called without an argument `stack` outputs the current setting.

`stats TO/K,PROFILE/S,ANALYSIS/S`

This command controls the tallying facility which counts the execution of individual Cintcode instructions. If no arguments are given, `stats` turns on tallying by clearing the tally vector and causing tallying to be enabled for the next command to be executed. Subsequent commands are not tallied, making it possible to process the tally vector while it is in a static state. Typical usage of the `stats` command is illustrated below:

`preload queens`	Preload the program to study
`stats`	Enable stats gathering on next command
`queens`	Execute the command to study
`interpreter`	Select the fast interpreter (`cintasm`) `stats` automatically selects the slow one
`stats to STATS`	Send instruction frequencies to file
	or
`stats profile to PROFILE`	Send detailed profile info to file
	or
`stats analysis to ANALYSIS`	Generate statistical analysis to file

`type FROM/A,TO,N/S`

This command will output the file given by the `FROM` argument, sending it to the screen unless the `TO` argument is given. The switch argument `N` causes line numbers to be added.

`typehex FROM/A,TO/K`

This will output the file specified by `FROM` in hexadecimal and send the result to the `TO` file if this argument is given.

`unpreload ,,,,,,,,,,ALL/S`

This command will remove preloaded commands from the Cintcode memory. The `ALL` switch will cause all preloaded commands to be removed.

8.3 The implementation of `CLI`, `newcli` and `run`

The main command language interpreter runs as task 1 and is implemented by the program whose source is `CLI.b`. This code is shared by the CLI tasks created by the commands `newcli` and `run`. When a CLI task is started it initialises itself by calling `cli.init(parm.pkt)` where `parm.pkt` is the startup packet for the task. Every CLI task has a segment list structure of size 4, with `seglist!1` containing the KLIB segments, `seglist!2` the segmemts belonging to BLIB, seglist!3 is the segment containing the function `cli.init` and `seglist!4` contains the compiled form of `CLI.b`.

For the main CLI task, the definition of `cli.init` is in a separate file `CLI_INIT.b` whose compiled form is placed in `seglist!3`. The commands `newcli` and `run` have their own definition of `cli.init` embedded in the same segment as their `start` functions. They can thus both be executed as commands as well as being placed in `seglist!3` of a CLI task to provide its initialisation code.

During the activation of a task, its global vector is initialised by the following code (in `KLIB`).

```
seglist := tcb.seglist!tcb
FOR i = 1 TO seglist!0 DO sys(Sys_globin(seglist!i)
```

For the CLI tasks created by `newcli` and `run`, the order in which the segment lists are initialised is important. They both put their own code in `seglist!3` (so that `cli.init` is defined) and the compilation of `CLI.b` in `seglist!4`. The definition of `start` in `CLI.b` thus overrides any definition of `start` in `seglist!3`.

In `CLI.b`, the code to call `cli.init` is as follows:

```
{ LET f = cli.init(parm.pkt)
  IF f DO f(result2)
}
```

If the returned result is non zero, it must be a function that can be applied to `result2`. This mechanism is used by `CLI_INIT.b` and `newcli.b` to invoke `unloadseg(seglist!3)` after `cli.init` has completed.

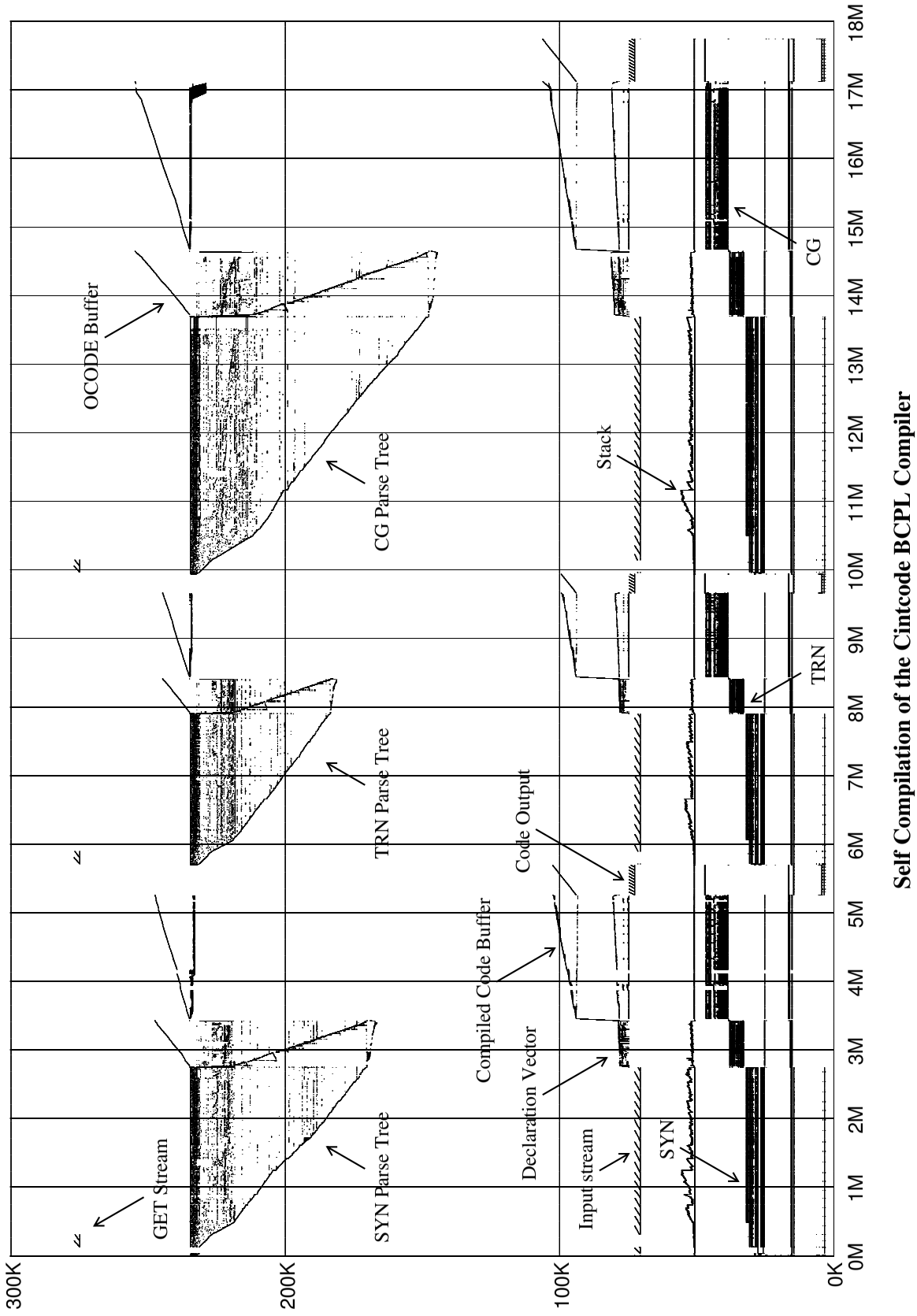


Figure 8.2: Self compilation memory-time graph

Chapter 9

The Standalone Debugger

When Cintpos starts up, the first BCPL code to be executed is the function `start` in `BOOT`. This runs using the global vector and stack it was given. It initialises BOOT's own coroutine environment

This initialises the system and creates a running environment for the kernel. This includes the creation of four tasks. Task 1 is an interactive command language interpreter (`CLI`). Task 2 is an interactive debugging task. Task 3 is the console handler that controls interaction between the other tasks and the main keyboard and screen attached to the computer. Task 4 is the file handler controlling access to the main filing system.

Once the kernel data structure and these tasks have been setup, control is passed to the scheduler by means of a recursive call to the Cincode interpreter by means of the call `sys(Sys_interpret, klibregs)`. This causes normal execution of Cintpos to start, including the processing of asynchronous interrupts. This continues until the Cintpos system is explicitly terminated or until a fault is encountered. On encountering a fault, control returns from the recursive invocation of the interpreter and resumes in the original `BOOT` environment. The Cintcode registers at the time of the fault will have been saved in `klibregs`, and the fault code stored in the `BOOT` variable `res`. If `res` is zero `BOOT` returns to the host operating system shell, if it is `-1` it immediately re-enters the interpreter, possibly using a different interpreter if register `count` has changed. Otherwise, the standalone debugger in entered.

The standalone debugger can read and update Cintpos memory, set and clear breakpoints and perform single step execution. The single step execution facility allows for single step execution of user programs, kernel code and even interrupt routines. For this to be useful, it is commonly necessary to disable most interrupts. The way the debugger behaves depends on the value held in the Cintcode status register `st`. There are four cases as follows:

`st=0`

Execution is in user mode with interrupts enabled, running code belonging to the current task (held in `rootnode!rtn.crntask`). The task will have a stack `rootnode!rtn.crntask!rtn.sbase` and a global vector `rootnode!rtn.crntask!rtn.gbase`). If the fault occurred early on during task activation, the stack and global vectors may not be fully initialised and there may be no proper coroutine environment. During this activation stage the field tcb.active is `FALSE` only being set to `TRUE` when task activation is complete. If the field `tcb.active` in a task's TCB is `TRUE`, the globals and coroutine environment can be assumed to be properly setup. When `tcb.active is FALSE`, some debugging commands will either be disallowed or have unexpected results.

`st=1`

Execution is performing some kernel operation involving the kernel data structure with interrupts disabled. All rootnode fields are set, but may be temporarily inconsistent while the kernel operation is being executed.

`st=2`

Execution is within `BOOT` which is typically assumed to be debugged. The rootnode fields cannot be assumed to be set.

`st=3`

Execution is in an interrupt routine. Interrupt routines each have their own stack but share the kernel's global vector and run with interrupts disabled. While in an interrupt routine, the user level registers belonging to the current task are held in `saveregs`. Execution of the interrupt routine terminated either directly by the call `sys(Sys_rti, saveregs)` or indirectly via a call of `srchwk`. Interrupt routines typically use `movepkt` to return packets to their tasks.

9.1 Entering the Standalone Debugger

As stated above, the standalone debugger can be entered as a result of a fault such as division by zero, but can also be entered explicitly using the CLI `abort`

command, as follows:

```
560> abort

!! ABORT 99: User requested
a1#
```

The standalone debugger's prompt, in this case `a1#`, consists of a letter indicating the current status of the code being debugged, followed by the number of the currently selected task and ending with a `#` to indicate that the standalone debugger (not the DEBUG task) has been entered. The status codes are as follows:

Code	Meaning
`a`	The selected task is fully active, `st=0`
`d`	The selected task is becoming active or dying, `st=0`
`k`	Obeying code in KLIB, `st=1`
`b`	Obeying code in BOOT, `st=2`
`i`	Executing the interrupt service routine, `st=3`

9.2 Debugger Commands

A brief description of the available debug commands can be displayed using the query (`?`) command.

```
a1# ?
?          Print list of debug commands
Gn Pn Rn Vn                Variables
G  P  R  V                 Pointers
n #b101 #o377 #x7FF 'c     Constants
*e /e %e +e -e |e &e       Dyadic operators
< > !                      Postfixed operators
SGn SPn SRn SVn SWn SAn    Store current value
Sn                         Select task n
S.                         Select current task
H          Hold/Release selected task
K          Enable/Disable clock interrupts
=          Print current value
Tn         Print n consecutive locations
$c         Set print style C, D, F, B, O, S, U or X
I          Print current instruction
N          Print next instruction
Q          Quit -- leave the cintpos system
M          Set/Reset memory watch address
B 0Bn eBn  List, Unset or Set breakpoints
C          Continue execution
X          Set breakpoint 9 at start of clihook
Z          Set breakpoint 9 at return of current fn
\          Execute one instruction
.          Move to current coroutine
,          Move down one stack frame
;          Move to parent coroutine
[          Move to first coroutine
]          Move to next coroutine
a1#
```

The debugger has a current value that can be loaded, modified and displayed. For example:

```
a1# 12                                  Set the current value to 12
a1# -2                                  Subtract 2
a1# *3                                  Multiply by 3
a1# =          30                       Display the current value
a1# <                                   Shift left one place
a1# =          60                       Display the current value
a1# 12 -2 *3 < =          60            Do it all on one line
a1#
```

Six areas of memory, namely: the global vector, the current stack frame, the Cintcode register dump, 10 scratch variables, the TCB of the currently selected task and the entire cintpos memory are easily accessed using the letters `G`, `P`, `R`, `V`, `W` and `A`, respectively.

```
a1# 10sv1 11sv2                    Put 10 and 11 in variables 1 and 2
a1# vt5                            Display the first 5 variables

V    0:          0          10          11          0          0
a1#
a1# v1*50+v2=          511         A calculation using variables
a1# g0=       1000                 Display global zero (globsize)
a1# g=       28089                 Display the address of global zero
a1# ! =       1000                 Indirect and display
a1# gt10                           Display the first 10 globals

G    0:       1000     start       stop        sys         clihook
G    5:      #G005#    changec       48673       48673     srchwk
a1#
```

Notice that values that appear to be entry points display the first 7 characters of the function's name. Other display styles can be specified by the commands `$C`, `$D`, `$F`, `$B`, `$O`, `$S`, `$U` or `$X`. These styles are respectively: characters, decimal number, in function style (the default), binary, octal, string, unsigned decimal and hexadecimal.

It is possible to display Cintcode instructions using the commands `I` and `N`. For example:

```
a1# g4=    clihook                 Get the entry to clihook
a1# i   24844:    K4G  1           Call global 1, incrementing P by 4
a1# n   24846:    RTN               Return from the function
a1#
```

A breakpoint can be set at the first instruction of `clihook` and the debugged program re-entered by the following:

```
a1# g4=    clihook                 Get the entry to clihook
a1# b9                             Set break point 9
a1# c                              Resume execution
1>
```

The `X` command could have been used since it is a shorthand for `G4B9C`. The function `clihook` is defined in `BLIB` and is called whenever a CLI command is invoked. For example:

```
1> echo ABC                        Invoke the echo command

!! BPT 9:      clihook             Breakpoint 9 reached
   A=          0 B=          0    24844:    K4G  1
a1#
```

Notice that the values of the Cintcode registers `A` and `B` are displayed, followed by the program counter `PC` and the Cintcode instruction at that point. Single step execution is possible, for example:

```
a1# \A=          0 B=          0    80700:    LLP  4
a1# \A=      48687 B=          0    80702:    SP3
a1# \A=      48687 B=          0    80703:     SP  89
a1# \A=      48687 B=          0    80705:      L  80
a1# \A=         80 B=      48687    80707:     SP  90
a1# \A=         80 B=      48687    80709:    LLL  80736
a1# \A=      20184 B=         80    80711:     LG  78
a1# \A=    rdargs  B=      20184    80713:      K  85
a1# \A=      20184 B=      20184    19932:    LP4
a1#
```

At this point the first instruction of `rdargs` is about to be executed. Its return address is in `P1`, so a breakpoint can be set to catch the return, as follows:

```
a1# p1b8
a1# c

!! BPT 8:        80715
     A=     40689 B=          0    80715:   JNE0  80718
a1#
```

A breakpoint can be set at the start of `sys`, as follows:

```
a1# g56b1           Set breakpoint 1 in wrch
a1# b               Display the currently set of breakpoints
1:      wrch
8:        80715
9:      clihook
a1# 0b8 0b9         Unset breakpoints 8 and 9
a1# b               Display the remaining breakpoint
1:      wrch
a1#
```

The next three calls of `wrch` will write the characters `ABC`. The following example steps through these and stops at the moment the newline character is about to be written.

```
a1# c

!! BPT 1:      wrch
   A=         65 B=      48694    17992:    LG  53
a1# c

!! BPT 1:      wrch
   A=         66 B=      48694    17992:    LG  53
a1# c

!! BPT 1:      wrch
   A=         67 B=      48694    17992:    LG  53
a1# c

!! BPT 1:      wrch
   A=         10 B=      48694    17992:    LG  53
a1#
```

The state of the runtime stack is now inspected and normal execution after removing the remaining breakpoint.

```
a1# .   48679:  Active coroutine     clihook   Size  1000  Hwm   152
        48811:    wrch             10          83      195204       43090
a1# ,   48801:    write.f       20191       48799       48800           1
a1# ,   48744:    writef        20191       48694          80       48695
a1# ,   48689:    start         48693       48694  1128415492           0
a1# ,   48685:    clihook           0      194740
a1# , Base of stack
a1# 0b1c                  Clear breakpoint 1 and resume
ABC
1>
```

Notice that the characters `ABC` are buffered and not written to the screen until the newline character is written.

The following debugging commands allow the coroutine structure to be explored.

Command	Effect
.	Select and display the current coroutine of the currently selected task
,	Select and display its next stack frame
;	Select and display the parent coroutine
[	Select and display the first coroutine belonging to the selected task
]	Select and display the next coroutine

Finally, the command `Q` causes a return from the Cintcode system.

Chapter 10

Installation

The following needs change.

The implementation of BCPL described in this report is is available free via my Home Page [Ric] to individuals for private use and to academic institutions. If you install the system, please send me a message (to `mr@cl.cam.ac.uk`) so I can keep a record of who is interested in it.

This implementation is designed to be machine independent being based on an interpreter written in C. There are, however, hand written assembly language versions of the interpreter for several architectures (including i386, MIPS, ALPHA and Hitachi SH3). For Windows 95/98/NT and Windows CE there are precompiled `.exe` files, but for all the other architectures it is necessary to rebuild the system.

The simplest installation is for Linux machines.

10.1 Linux Installation

This section describes how to install the BCPL Cintcode System on an IBM PC running Linux.

- First create a directory named `BCPL` and copy either `bcpl.targz` or `bcpl.zip` into it. They are available (free) via my home page [Ric] and both contain the same set of packed files and directories.
- Either unpack `bcpl.targz` by:

```
tar zxvf bcpl.targz
```

  or unpack `bcpl.zip` using:

```
unzip -v bcpl.zip
```

This will create the directories `cintcode`, `bcplprogs` and `natbcpl`. The directory `cintcode` contains all the source files of the BCPL Cintcode System, `bcplprogs` contains a collection of demonstration programs, and `natbcpl` contains a version of BCPL that compiles into native code (for Intel and ALPHA machines.

- Now change directory to `cintcode`.

```
cd cintcode
```

- Re-build enter the BCPL system:

```
make
```

This should generate output that ends with:

```
BCPL Cintcode System
0>
```

indicating that the Command Langage Interpreter has been successfully entered.

- It is now necessary to recompile all the system software and commands. This is done by typing:

```
c compsys
```

- Now try out a few commands, eg:

```
echo hello
bcpl com/echo.b to junk
junk hello
map pic
logout
```

- The BCPL programs that are part of the system are: `BOOT.b`, `BLIB.b` and `CLI.b`. These reside in `BCPL/cintcode/sys` and can be compiled by the following commands (in the BCPL Cintcode System).

```
c bs BOOT
c bs BLIB
c bs CLI
```

  The standard commands are in `BCPL/cintcode/com` may be compiled using `bc`.

```
c bc echo
c bc abort
c bc logout
c bc stack
c bc map
c bc prompt
```

  Read the documentation in `cintcode/doc` and any `README` files you can find. A log of recent changes can be found in `cintcode/doc/changes`. A postscript version of the current version of this BCPL manual is available from my home page. There is a demonstration script of commands in `cintcode/doc/notes`.

- In order to use the BCPL Cintcode System from another directory it is necessary to define the shell variable `BCPLPATH` to be the absolute file name of the cintcode directory and add this directory to your `PATH`, before entering the BCPL Cintcode system. On Linux, this can be done by:

```
export BCPLPATH=/home/mr/distribution/BCPL/cintcode
export PATH=$PATH:$BCPLPATH
```

  The shell variable `BCPLPATH` is used when loading Cintcode object modules, reading BCPL header files and files read by the `c` command.

- To compile and run a demo program such as `bcplprogs/demos/queens.b`:

```
cd ../bcplprogs/demos
cinterp
c b queens
queens
```

10.2 Command Line Arguments

The command `cintpos` that invokes the Cintcode interpreter can be given arguments to control memory allocation. These are:

- `-m` n Set the cintcode memory size to $1000n$ words
- `-t` n Set the tally vector size to $1000n$ words
- `-h` Output some help information

10.3 Installation on Other Machines

Not yet available.

Bibliography

[Ric] M. Richards. *My WWW Home Page*. www.cl.cam.ac.uk/users/mr/.

Appendix A

BCPL Syntax Diagrams

The syntax of BCPL is specified using the transition diagrams given in figures A.1, A.2, A.3 and A.4. Within the diagrams the syntactic categories *program*, *section*, *declaration*, *command* and $expression_n$ are represented by the rounded boxes: —(program)—, —(section)—, —(D)—, —(C)— and —(En)—, respectively.

The rectangular boxes are called test boxes and can only be traversed if the condition labelling the box matches the current input. When the label is a token, as in —[WHILE]— and —[:=]—, it must match the next input token for the test to succeed. The test box —[eof]— is only satisfied if the end of file has been reached. Sometimes the test box contains a side condition, as in —[REM n<6]—, in which case the side condition must also be satisfied. The only other test boxes are —[is call]— and —[is name]— which are only satified if the most recently read expression is syntactically a function call or a name, respectively. By setting n successively from 0 to 8 in the definition of the category —(En)—, we obtain the definitions of —(E0)— to —(E8)—. Starting from the definition of —(program)—, we can construct an infinite transition diagram containing only test boxes by simply replacing all rounded boxes by their definitions, recursively. The parsing algorithm searches through this infinite diagram for a path with the same sequence of tokens as the program being parsed. In order to eliminate ambiguities, the left hand branch at a branch point is tried first. Notice how this rule causes the command

```
IF i>10 DO i := i/2 REPEATUNTIL i<5
```

to be equivalent to

```
IF i>10 DO { i := i/2 REPEATUNTIL i<5 }
```

and not

```
{ IF i>10 DO i := i/2 } REPEATUNTIL i<5
```

A useful property of these diagrams is that, once a test box has been successfully traversed, previous branching decisions need not be reconsidered and so the parser need never backtrack.

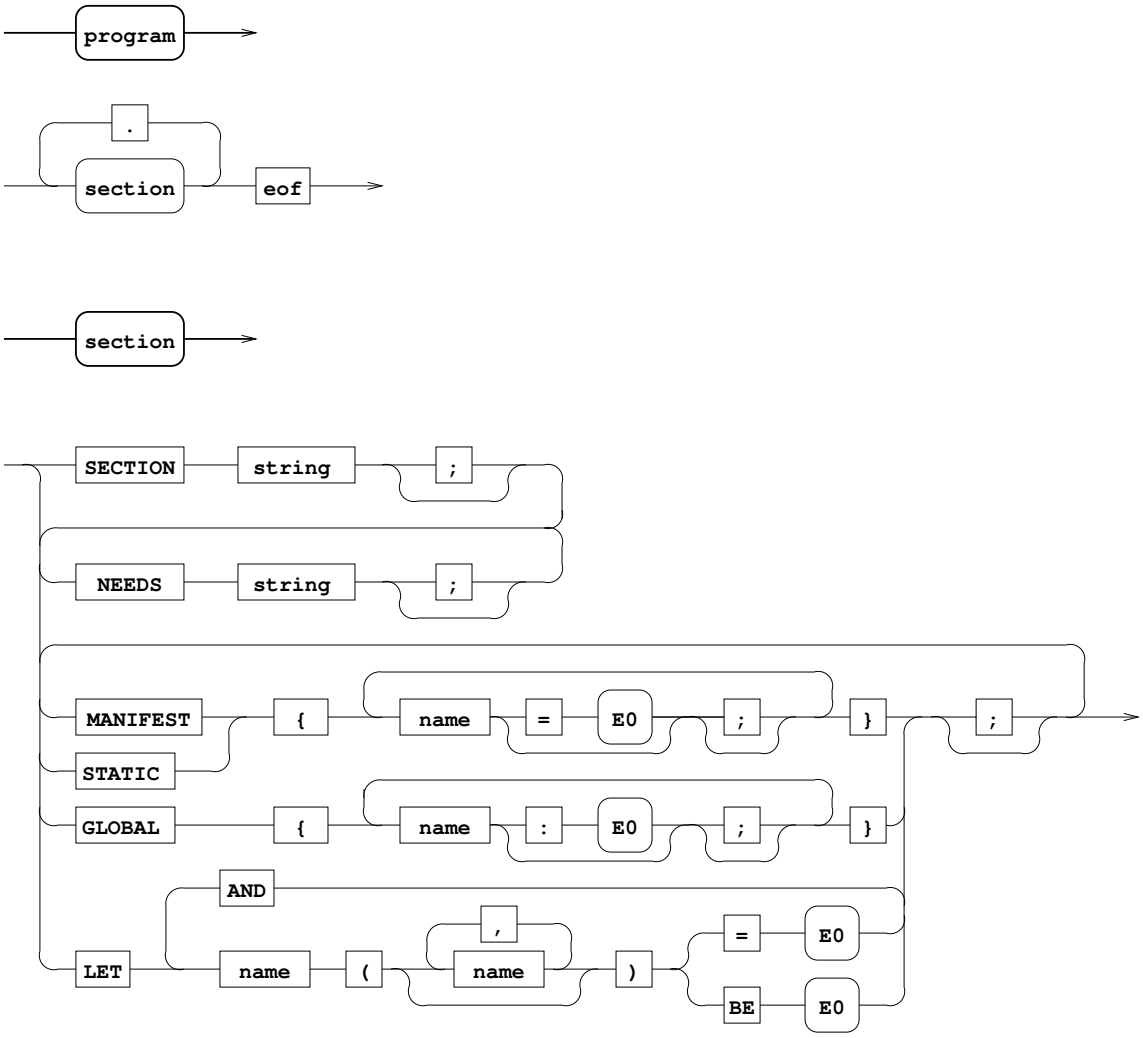


Figure A.1: Program, Section

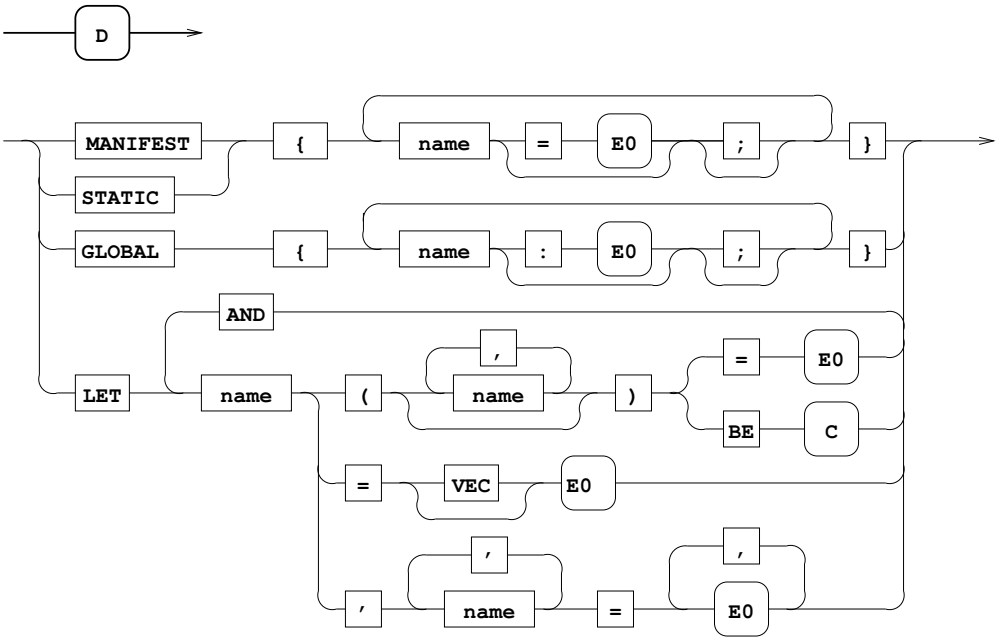


Figure A.2: Declarations

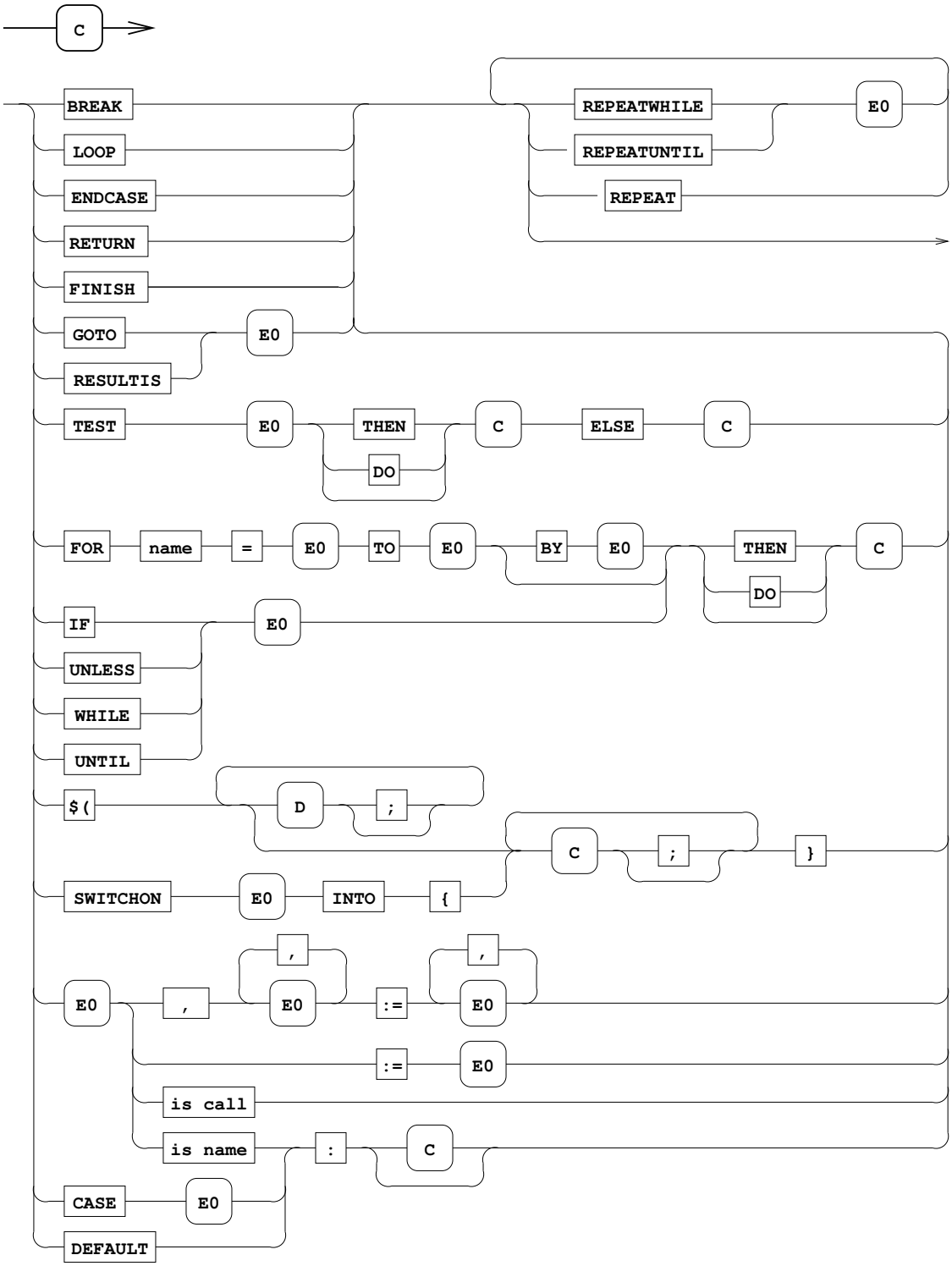


Figure A.3: Commands

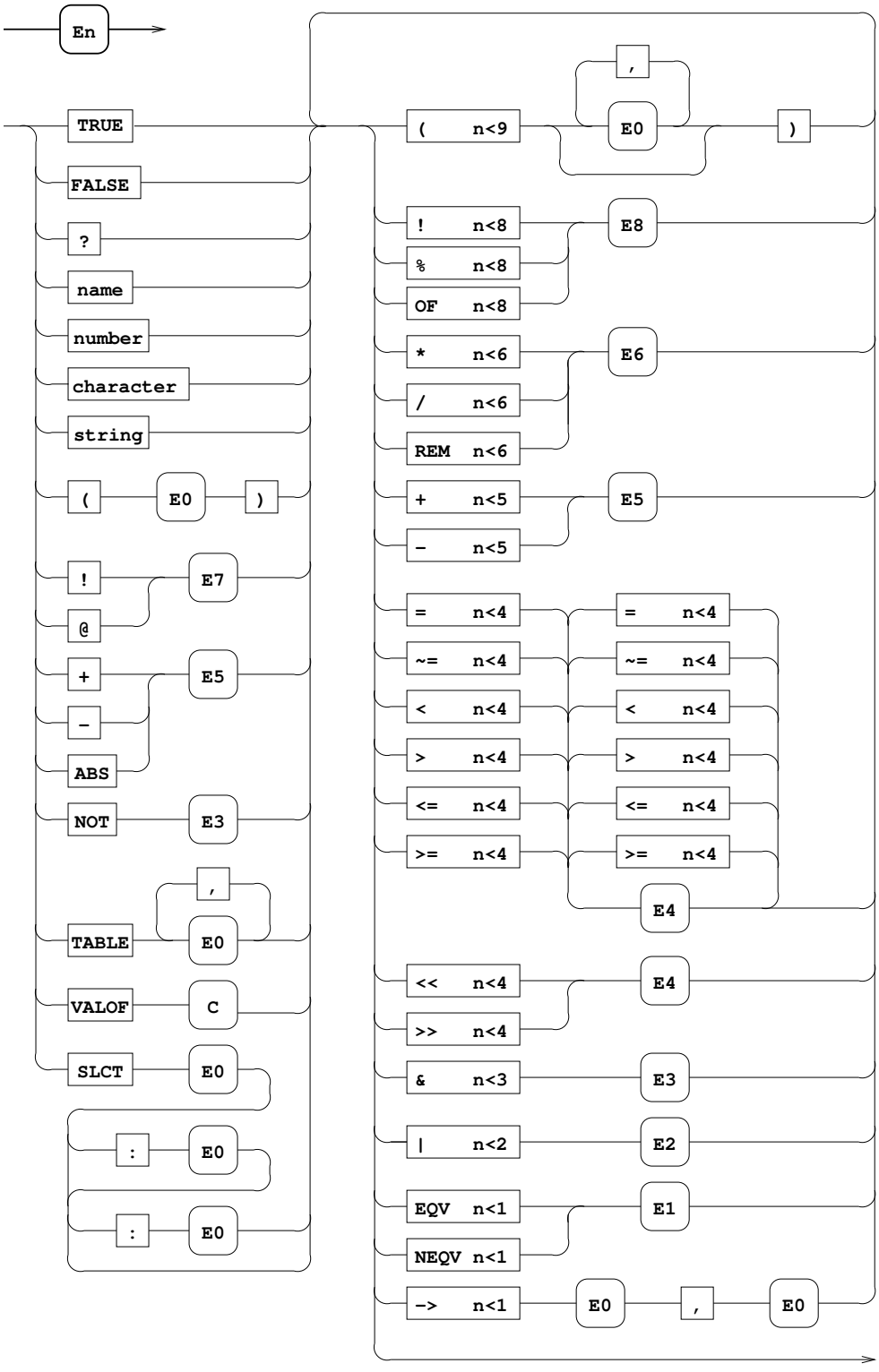


Figure A.4: Expressions