The BCPL Cintcode and Cintpos User Guide

by

Martin Richards

mr@cl.cam.ac.uk

http://www.cl.cam.ac.uk/users/mr10/

Computer Laboratory
University of Cambridge

Revision date: September 10, 2009

Abstract

BCPL is a simple systems programming language with a small fast compiler which is easily ported to new machines. The language was first implemented in 1967 and has been in continuous use since then. It is a typeless and provides machine independent pointer arithmetic allowing a simple way to represent vectors and structures. BCPL functions are recursive and variadic but, like C, do not allow dynamic free variables, and so can be represented by just their entry addresses. There is no built-in garbage collector and all input-output is done using library calls.

This document describes the new revised version of the BCPL Cintcode System giving a definition of the language, its library and running environment. It also describes a native code version of the system and the Cintpos portable operating system. Installation instructions are included.

Keywords

Systems programming language, Typeless language, BCPL, Cintcode, Coroutines, Cintpos.

Contents

Preface

The concept for BCPL originated in 1966 and was first outlined in my PhD thesis [4]. Its was first implemented early in 1967 when I was working at M.I.T. Its heyday was perhaps from the mid 70s to the mid 80s, but even now it is still continues to be used at some universities, in industry and by private individuals. It is a useful language for experimenting with algorithms and for research in optimizing compilers. Cintpos is the multi-tasking version of the system based on the Tripos [5]. It is simple and easy to maintain and can be used for real-time applications such as process control. BCPL was designed many years ago but is still useful in areas where small size, simplicity and portability are important.

This document is intended to provide a record of the main features of the BCPL in sufficient depth to allow a serious reader to obtain a proper understanding of philosophy behind the language. An efficient interpretive implementation is presented, the source of which is freely available via my home page [3]. The implementation is machine independent and should be easy to transfer to almost any architecture both now and in the future.

The main topics covered by this report are:

- A specification of the BCPL language.
- A description of its runtime library and the extensions used in the Cintpos system.
- The design and implementation of command language interpreters for both the single and multi-threaded versions of the system.
- A description of OCODE, the intermediate code used in the compiler, and Cintcode, the compact byte stream target code used by the interpreter.
- A description of the single and multi-threaded interactive debugger and other debugging aids.
- The efficient implementation of the Cintcode interpreter for several processors including both RISC and i386/Pentium based machines.
- The profiling and statistics gathering facilities offered by the system.

- The SIAL intermediate code that allows easy translation of BCPL in native code for most architectures.
- The MC package that allows machine independent dynamic compilation and execution of native machine code.

Chapter 1

The System Overview

This document contains a full description of an interpretive implementation of BCPL that supports a command language and low level interactive debugger. As an introduction, an example console session is presented to exhibit some of the key features of the single threaded version of the system.

1.1 A Console Session

When the system is started (on a machine called meopham) in the directory `bcplprogs/demo`, its opening message is as follows:

```
meopham$ cintsys

BCPL Cintcode System (25 Jan 2007)
0>
```

The characters `0>` are followed by a space character and is the command language prompt string inviting the user to type a command. The integer gives the execution time of the preceeding command. A program to compute factorials can be displayed using the `type` command as follows:

```
> type fact.b
GET "libhdr"

LET start() = VALOF
{ FOR i = 1 TO 5 DO writef("fact(%n) = %i4*n", i, fact(i))
  RESULTIS 0
}

AND fact(n) = n=0 -> 1, n*fact(n-1)
0>
```

The directive `GET "libhdr"` causes the standard library declarations to be inserted at that position. The text:

```
LET start() = VALOF
```

is the heading for the declaration of the function `start` which, by convention, is the first function to be called when a program is run. The empty parentheses `()` indicate that the routine expects no arguments. The text

```
FOR i = 1 TO 5 DO
```

introduces a for-loop whose control variable `i` successively takes the values from `1` to `5`. The body of the for-loop is calls the library routine `writef` whose effect is to output the format string after replacing the substitution items `%n` and `%i4` by appropriately formatted representations of `i` and `fact(i)`. Within the string `*n` represents the newline character. The statement `RESULTIS 0` exits from the `VALOF` construct providing the result of `start` that indicates the program completed successfully. The text:

```
AND fact(n) =
```

introduces the definition of the function `fact` which take one argument (`n`) and yields `n` factorial. The word `AND` causes `fact` to available to the previously defined function. This program can be compiled by using the following command:

```
10> bcpl fact.b to fact

BCPL (10 June 2004)
Code size = 104 bytes
10>
```

This command compiles the source file `fact.b` creating an executable object module in the file called `fact`. The program can then be run by simply typing the name of this file.

```
10> fact
fact(1) =    1
fact(2) =    2
fact(3) =    6
fact(4) =   24
fact(5) =  120
0>
```

When the BCPL compiler is invoked, it can be given additional arguments that control the compiler options. One of these (`d1`) directs the compiler to output the compiled code in a readable form, as follows:

```
10> bcpl fact.b to fact d1

BCPL (10 June 2004)
   0:  DATAW 0x00000000
   4:  DATAW 0x0000DFDF
   8:  DATAW 0x6174730B
  12:  DATAW 0x20207472
  16:  DATAW 0x20202020
// Entry to:   start
  20: L1:
  20:     L1
  21:    SP3
  22: L4:
  22:    LP3
  23:     LF  L2
  25:     K9
  26:    SP9
  27:    LP3
  28:    SP8
  29:    LLL  L9920
  31:    K4G   94
  33:     L1
  34:    AP3
  35:    SP3
  36:     L5
  37:    JLE  L4
  39:     L0
  40:    RTN
  44: L9920:
  44:  DATAW 0x6361660F
  48:  DATAW 0x6E252874
  52:  DATAW 0x203D2029
  56:  DATAW 0x0A346925
  60:  DATAW 0x0000DFDF
  64:  DATAW 0x6361660B
  68:  DATAW 0x20202074
  72:  DATAW 0x20202020
// Entry to:   fact
  76: L2:
  76:   JNE0  L5
  78:     L1
  79:    RTN
  80: L5:
  80:    LM1
  81:    AP3
  82:     LF  L2
  84:     K4
  85:    LP3
  86:    MUL
  87:    RTN
  88: L3:
  88:  DATAW 0x00000000
  92:  DATAW 0x00000001
  96:  DATAW 0x00000014
 100:  DATAW 0x0000005E
Code size = 104 bytes
20>
```

This output shows the sequence of CINTCODE instructions compiled for the two procedures defined in the factorial program. In addition to the instructions, there are some data words holding the string constant, initialisation data and symbolic information for the debugger. The data word at location 4 holds a special bit pattern indicating the presence of a procedure name placed just before the entry point. As can be seen the procedure in this case is `start`. Similar information is packed at location 60 for the function `fact`. Most Cintcode instructions occupy one byte and perform simple operations on the registers and memory of the Cintcode machine. For instance, the first two instructions of `start` (`L1` and `SP3` at locations 20 and 11) load the constant `1` into the Cintcode `A` register and then stores it at word 3 of the current stack frame (pointed to by `P`). This corresponds to the initialisation of the for-loop control variable `i`. The start of the for-loop body has label `L4` corresponding to location 22. The compilation of `fact(i)` is `LP3 LF L2 K9` which loads `i` and the entry address of `fact` and enters the function incrementing `P` by `9` locations). The result of this function is returned in `A` which is stored in the stack using `SP9` in the appropriate position for the third argument of the call of `writef`. The second argument, `i`, is setup using `LP3 SP8`, and the first argument which is the format string is loaded by `LLL L9920`. The next instruction (`K4G 94`) causes the routine `writef`, whose entry point is in global variable 94, to be called incrementing `P` by 4 words as it does so. Thus the compilation of the call `writef("fact(%n) = %i5*n", i, f(i))` occupies just 11 bytes from location 22 to 32, plus the 16 bytes at location 44 where the string is packed. The next three instructions (`L1 AP3 SP3`) increment `i` and `L5 JNE L4` jumps to label L4 if `i` is still less than `5`. If the jump is not taken, control falls through to the instructions `L0 RTN` causing `start` to return with result `0`. Each instruction of this function occupies one byte except for the `LF`, `LLL`, `K4G` and `JNE` instructions which each occupy two. The body of the function `fact` is equally easy to understand. It first tests whether its argument is zero (`JNE0 L5`). If it is, it returns one (`L1 RTN`). Otherwise, it computes `n-1` by loading `-1` and adding `n` (`LM1 AP3`) before calling `fact` (`LF L2 K4`). The result is then multiplied by `n` (`LP3 MUL`) and returning (`RTN`). The space occupied by this code is just 12 bytes.

The debugger can be entered using the abort command.

```
20> abort

!! ABORT 99: User requested
*
```

The asterisk is the prompt inviting the user to enter a debugging command. The debugger provides facilities for inspecting and changing memory as well as setting breakpoints and performing single step execution. As an example, a breakpoint is placed at the first instruction of the routine `clihook` which is used by the command language interpreter (CLI) to transfer control to a command. Consider the following commands:

```
* g4 b1
* b
1:   clihook
*
```

This first loads the entry point of `clihook` (held in global variable 4) and sets (`b1`) a breakpoint numbered 1 at this position. The command `b`, without an argument, lists the current breakpoints confirming that the correct one has been set. Normal execution is continued using the `c` command.

```
* c
20>
```

If we now try to execute the factorial program, we immediately hit the breakpoint.

```
0> fact

!! BPT 1:   clihook
   A=            0 B=           0    15740:    K4G  1
*
```

This indicates that the breakpoint occured when the Cintcode registers `A` and `B` were both zero, and that the program counter is set to 15740 where the next instruction to be obeyed is `K6G 1`. Single step exection can now be performed using the `\` command.

```
* \A=            0 B=           0    42036:     L1
* \A=            1 B=           0    42037:    SP3
* \A=            1 B=           0    42038:    LP3
*
```

After each single step execution a summary of the current state is printed. In the above sequence we see that the execution of the instruction `L1` loading `1` into the `A` register. The execution of `SP3` does not have an immediately observable effect since it updates a local variable held in the current stack frame, but the stack frame can be displayed using the `t` command.

```
* p t4

P    0:       42164        15742  start                      1
*
```

This confirms that location `P3` contains the value `1` corresponding to the initial value of the for-loop control variable `i`. At this stage it is possible to change its value to `3`, say.

```
* 3 sp3
* p t4

P    0:       42164        15742  start                      3
*
```

If single stepping is continued for a while we observe the evaluation of the recursive call `fact(3)`.

```
* \A=          3 B=           1    42039:     LF  42092
* \A= fact       B=           3    42041:     K9
* \A=          3 B=           3    42092:   JNE0  42096
* \A=          3 B=           3    42096:    LM1
* \A=         -1 B=           3    42097:    AP3
* \A=          2 B=           3    42098:     LF  42092
* \A= fact       B=           2    42100:     K4
* \A=          2 B=           2    42092:   JNE0  42096
* \A=          2 B=           2    42096:    LM1
* \A=         -1 B=           2    42097:    AP3
* \A=          1 B=           2    42098:     LF  42092
* \A= fact       B=           1    42100:     K4
* \A=          1 B=           1    42092:   JNE0  42096
* \A=          1 B=           1    42096:    LM1
* \A=         -1 B=           1    42097:    AP3
* \A=          0 B=           1    42098:     LF  42092
* \A= fact       B=           0    42100:     K4
* \A=          0 B=           0    42092:   JNE0  42096
* \A=          0 B=           0    42094:     L1
* \A=          1 B=           0    42095:    RTN
* \A=          1 B=           0    42101:    LP3
* \A=          1 B=           1    42102:    MUL
* \A=          1 B=           1    42103:    RTN
* \A=          1 B=           1    42101:    LP3
* \A=          2 B=           1    42102:    MUL
* \A=          2 B=           1    42103:    RTN
* \A=          2 B=           1    42101:    LP3
* \A=          3 B=           2    42102:    MUL
* \A=          6 B=           2    42103:    RTN
* \A=          6 B=           2    42042:    SP9
* \A=          6 B=           2    42043:    LP3
* \A=          3 B=           6    42044:    SP8
* \A=          3 B=           6    42045:    LLL  42060
* \A=      10515 B=           3    42047:    K4G  94
*
```

At this moment the routine `writef` is just about to be entered to print an message about factorial 3. We can unset breakpoint 1 and continue normal execution by typing `0b1 c`.

```
* 0b1 c
fact(1) =    1
fact(4) =   24
fact(5) =  120
10>
```

Notice that `fact(1)` is the first to be written since it has already been evaluated, but the next time round the `FOR` loop `i` has value `4` since `i` was updated during the debugging session.

As one final example in this session we will re-compile the BCPL compiler.

```
10> bcpl ../../cintcode/com/bcpl.b to junk

BCPL (10 June 2004)
```

```
Code size = 10848 bytes
Code size = 9680 bytes
Code size = 12764 bytes
540>
```

This shows that the total size of the compiler is 33,292 bytes and that it can be compiled (on a 1GHz Pentium machine) in 0.54 seconds. Since this involves executing 27,188,756 Cintcode instructions, the rate is just over 50 million Cintcode instructions per second with the current interpreter.

Chapter 2

The BCPL Language

The design of BCPL owes much to the work done jointly by Cambridge and London Universities on CPL (originally Cambridge Programming Language) which was conceived at Cambridge to be the main language to run on the new and powerful Ferranti Atlas computer to be installed in 1963. At that time there was another Atlas computer in London and it was decided to make the development of CPL a joint project between the two Universities. As a result the name changed to Combined Programming Language. It could reasonably be called Christopher's Programming Language in recognition of Christpher Strachey whose bubbling enthusiasm and talent steered the course of its development.

CPL was an ambitious language in the ALGOL tradition but with many novel and significant extensions intended to make its area of application more general. These included a greater richness in control constructs such as the now well known `IF`, `UNLESS`, `WHILE`, `UNTIL`, `REPEATWHILE`, `SWITCHON` statements. It could handle a wide variety of data types including string and bit patterns and was one of the first strictly typed languages to provided a structure mechanism that permitted convenient handling of lists, trees and directed graphs. Work on CPL ran from about 1961 to 1967, but was hampered by a number of factors that eventually killed it. It was, for instance, too large and complicated for the machines available at the time, and the desire for elegance and mathematical cleanliness outweighed the more pragmatic arguments for efficiency and implementability. Much of the implementation was done by research students who came and left during the lifetime of the project. As soon as they knew enough to be useful they had to transfer their attention to writing their theses. Another problem (that became of particular interest to me) was that the implementation had to move from EDSAC II to the Atlas computer about halfway through the project. The CPL compiler thus needed to be portable. This was achieved by writing it in a simple subset of CPL which was then hand translated into a sequence of low level macro calls that could be expanded into the assembly language of either machine. The macrogenerator used was GPM[6] which was designed by Strachey specifically for this task. It was a delightfully elegant work of art in its own right and is well worth study. A variant of GPM, called BGPM, is included in the standard BCPL distribution.

BCPL was initially similar to the subset of CPL used in the encoding of the CPL compiler. An outline of BCPL's main features first appeared in my PhD thesis [4] in 1966 but it was not fully designed and implemented until early the following year when I was working at Project MAC of the Massachussetts Institute of Technology. Its first implementation was written in Ross's Algol Extended for Design (AED-0)[1] which was the only language then available on CTSS, the time sharing system at Project MAC, other than LISP that allowed recursion.

2.1 Language Overview

A BCPL program is made up of 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 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 or to perform input/output.

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 a comments the lexical analyser is only looking for `/*` and `*/` and so 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 directives of the form `GET "`*filename*`"` is replaced by the contents of the whole named file. Earlier versions of the compiler only inserted the file up to the first occurring dot. By convention, `GET` directives normally appear on separate lines. If the filename does not end in `.h` or `.b` the extension `.h` is added. This is looked up in the current directory and then the directories specified by the environment variable `BCPLHDRS`. If these all fail, `g/` is prepended to the file name which is then looked up in the directory specified by the `BCPLROOT` environment variable. There is a compiler option `hdrs` that allows the user to specify an alternative to `BCPLHDRS`, such as `POSHDRS` for the Cintpos system. The default setting for `hdrs` is held in the rootnode. For `cintsys` it is `BCPLHDRS` and for `Cintpos` it is `POSHDRS`. Both `cintsys` and `cintpos` have options to change these default settings. To check whether these environment variables are set correctly, enter `cintsys` or `cintpos` with the `-f` option.

2.1.3 Conditional Compilation

There is a simple mechanism, whose implementation takes fewer than 20 lines of code in the 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 the word length is 64 bits.

2.2.1 Names

Syntactically a name is of a sequence of letters, digits, dots and underlines starting with a letter that is not one of the reserved words (such as `IF`, `WHILE`, `TABLE`).

A name may be declared to a local variable, a static variable, a global variable, a manifest constant or a function. Since the language is typeless, the value of a name is a bit pattern whose interpretation depends on how it is used.

2.2.2 Constants

Decimal numbers consist of a sequence of digits, while binary, octal or hexadecimal are represented, repectively, by `#b`, `#o` or `#x` followed by digits of the appropriate sort. The case of the letters in hexadecimal numbers is not significant nor is the case of the letters `b`, `o` or `x` after a `#`. The `o` may be omitted in octal numbers. The 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
#X3fff
#x_DEADCODE
```

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 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]).
`*#g`	Set the encoding mode to GB2312 for the rest of this string or character constant. The default encoding is UTF8 unless speified by the GB2312 compiler option, See the specification of the `bcpl` command on page **??**.
`*#u`	Set the encoding mode to UTF8 for the rest of this string or character constant.
`*#`*hhhh*	In UTF8 mode, this specifies a single Unicode character with up to four hexadecimal digits. In string constants, this is converted to a sequence of bytes giving its UTF-8 representation. In character constants, it yields the integer *hhhh*. Thus `'*#C13F'=#xC13F`.
`*##`*h..h*	In UTF8 mode, this specifies a Unicode character with up to eight hexadecimal digits, but is otherwise treated as the `*#hhhh` escape.
`*#`*dddd*	In GB2312 mode, this specifies the GB2312 decimal code (*dddd*) for an extended character. In string constants, this is converted to a sequence of bytes giving its GB2312 representation. In character constants, it yields the integer *dddd*. Thus `'*#g*#4566'=4566`.
`*`*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. The `*#` escapes allow Unicode and GB2312 characters to be handled. For instance, if the following statements output to a suitable UTF8 configured device:

```
writef("*#uUnicode 2200 prints as: '*#2200'*n"}
writef("%# in writef can also be used: '%#'*n", #x2200)
```

the result is as follows

```
Unicode 2200 prints as: '∀'
%# in writef can also be used: '∀'
```

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 are normally called in the context of an expression where a result is required, while routine are called in the context of a command not requiring a result. However both functions and routines can be called in either context. Unwanted results are thrown away, the result of a routine is undefined.

Syntactically, a call is an expression followed by an argument list 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 11).

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 consitent 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 expressions 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 ones of more than the word length return 0.

Expressions of the form: E_1`*`E_2, E_1`/`E_2, E_1 `MOD` E_2, E_1`+`E_2, E_1`-`E_2. E_1 `EQV` E_2 and E_1 `XOR` 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). `REM` and `NEQV` can be used as synonyms of `MOD` and `XOR`, 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 coditional constructs, such as if and while commands, are evaluated in a Boolean. This effects 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 know, 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	Names, Literals, `?`, `TRUE`, `FALSE`, `(E)`,	
9L	Function and method calls	
8L	`!  %  OF  ::`	Dyadic
7	`!  @`	Prefixed
6L	`*  /  MOD REM`	Dyadic operators
5	`+  -  ABS`	
4	`=  ~=  <=  >=  <  >`	Extended Relations
4L	`<<  >>`	
3	`~ NOT`	Bitwise and Boolean operators
3L	`&`	
2L	`|`	
1L	`EQV NEQV XOR`	
1R	`-> ,`	Conditional expression
0	`VALOF TABLE`	Valof and Table expressions
0	`SLCT :`	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`, `MOD`, `SLCT`, `*`, `/`, `+`, `-`, `ABS`, the relational operators, `<<`, `>>`, `NOT`, `~`, `&`, `|`, `EQV`, `NEQV`, `XOR`, 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* updates 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` $\{ C_1 ;\ldots; C_n \}$

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 50.

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 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:

$$\texttt{MANIFEST}\ \{\ N_1\texttt{=}K_1\ \texttt{;}\ldots\texttt{;}\ N_n\texttt{=}K_n\ \}$$

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:

$$\texttt{GLOBAL}\ \{\ N_1\texttt{:}K_1\ \texttt{;}\ldots\texttt{;}\ N_n\texttt{:}K_n\ \}$$

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:

`LET` N_1 `,...,` N_n `=` E_1 `,...,` 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 variables N_1, . . ,N_n are allocated consecutive locations in the stack and so, for instance, the variable N_i may be accessed by the expression $(@N_1)!(\mathtt{i}-\mathtt{1})$. This feature is a recent addition to the language.

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

`LET` N `= 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:

`LET` N `(` N_1 `, . . . ,` N_n `) =` E
`LET` N `(` N_1 `, . . . ,` N_n `) 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 procedures `level` and `longjump`, described on page **??**. 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 `unloadseq`, 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 4. 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 Library

This manual describes three variants of the BCPL system. The simplest is invoked by the shell command `cintsys` and provides a single threaded command language interpreter. The system invoked by `cintpos` provides a multi-threaded system where the individual threads (called tasks) are run in parallel and are pre-emptible. A third version is available for some architectures and provides a single threaded version in which the BCPL source is compiled into native machine code. Although this version is faster, it is more machine dependent, has fewer debugging aids and will only run a single command.

The libraries of these three systems have much in common and so are all described together. The description of all constants, variables and functions have a right justified line such as the following

CIN:y, POS:y, NAT:n

where `CIN:`, `POS:` and `NAT:` denote the single threaded, multi-threaded and native code versions, respectively, and the letters y and n stand for yes and no, showing whether the corresponding constant, variable or function is available on that version of the system.

The resident library functions, variables and manifest constants are declared in the standard library header file `g/libhdr.h`. Most of the functions are defined in BCPL in either `sysb/blib.b` or `sysb/dlib.b`, but three functions (`sys`, `chgco` and `muldiv`) are in the hand written Cintcode file `cin/syscin/syslib`. Most functions relating to the multi-threaded version are defined in `klib.b`.

The following three sections describe the manifest constants, variables and functions (in alphabetical order) provided by the standard library.

3.1 Manifest constants

B2Wsh CIN:y, POS:y, NAT:y

This constant holds the shift required to convert a BCPL pointer into a byte address. Most implementations use pack 4 bytes into 32-bit words requiring `B2Wsh=2`, but on

64-bit implementations, such as native code on the DEC Alpha or the 64-bit Cintcode version of BCPL, its value is 3.

bootregs CIN:n, POS:y, NAT:n

This is the location in Cintcode memory used in Cintpos to hold Cintcode registers during system startup.

bytesperword CIN:y, POS:y, NAT:y

Its value is `1<<B2Wsh` being the number of bytes that can be packed into a BCPL word. On 32-bit implementations it is 4, and on 64-bit versions it is 8.

bitsperbyte CIN:y, POS:y, NAT:y

This specifies the number of bits per byte. On most systems `bitsperbyte` is 8.

bitsperword CIN:y, POS:y, NAT:y

It value is `bitsperbyte*bytesperword` being the number of bits per BCPL word. It is usually 32, but can be 64.

CloseObj CIN:y, POS:y, NAT:y

This identifies the position of the `close` method in objects using BCPL's version of object oriented programming. Typical use is as follows:

```
        CloseObj#(obj)
```

For more details, see `mkobj` described on page 46.

co_c, co_fn, co_list, co_parent, co_pptr, co_size CIN:y, POS:y, NAT:y

These are the system fields as the base of coroutine stacks. If a coroutine is suspended, its `pptr` field holds the stack frame pointer (P) at the time it became suspended. The `parent` field points to the parent coroutine, if it has one, or is `-1` for root coroutines, and is zero otherwise. The `list` field holds the next coroutine in the list of coroutines originating from global `colist`. The `fn` and `size` fields hold the coroutine's main function and stack size, and the `c` field is a system work location. For more information about coroutines, see `createco` described on page 42.

deadcode CIN:y, POS:y, NAT:n

To aid debugging, the entire Cintcode memory is initialised to `deadcode`. Typically `deadcode=#xDEADCODE`.

endstreamch CIN:y, POS:y, NAT:y

This is the value returned by `rdch` when reading from a stream that is exhausted. Its value is normally `-1`.

entryword CIN:y, POS:y, NAT:n

To aid debugging, every functions entry point is marked by `entryword`. This is normally followed by a function name compressed into a string of 11 characters. If the function name is too long its first and last five character are packed into the string separated by a single quote `'`. Typically `entryword=#x0000DFDF`.

globword CIN:y, POS:y, NAT:n

This constant is used to assist the debugging of Cintcode programs. If the i^{th} global variable is not otherwise set, its value is `globword+`i. Typically `globword=#x8F8F0000`.

id_inscb, id_inoutscb, id_outscb CIN:y, POS:y, NAT:n

These constants are mnemonics for the possible values of the `id` field of a stream control block. See `scb_id` below.

InitObj CIN:y, POS:y, NAT:y

This identifies the position of the `init` method in objects using BCPL's version of object oriented programming. Typical use is as follows:

```
        InitObj#(obj, arg1, arg2)
```

For more details, see `mkobj` described on page 46.

isrregs CIN:n, POS:y, NAT:n

Under Cintpos this is the location in Cintcode memory used to hold the Cintcode registers representing the state at the start of the interrupt service routine.

klibregs CIN:n, POS:y, NAT:n

Under Cintpos This is the location in Cintcode memory used to hold Cintcode registers during system startup.

mcaddrinc CIN:y, POS:y, NAT:y

This is the difference between machine addresses of consecutive words in memory and is usually 4 or 8. Very occasionally, BCPL implementions have negatively growing stacks, in which case `mcaddrinc` will be negative.

maxint, minint CIN:y, POS:y, NAT:y

The constant `minint` is `1<<(bitsperword-1)` and `maxint` is `=minint-1`. They hold the most negative and largest positive numbers that can be represented by a BCPL word. On 32-bit implementations, they are normally `#x80000000` and `#x7FFFFFFF`.

pollingch CIN:n, POS:y, NAT:n

This is the value returned by `rdch` if a charcter is not immediately available from the currently selected stream. Its value is normally `-3`. Currently only TCP streams under Cintpos provide the polling mechanism.

rootnodeaddr CIN:y, POS:y, NAT:n

This manifest constant is used in Cintsys and Cintpos to hold the address of the root node. Its value is otherwise zero.

rtn_. . . CIN:y, POS:y, NAT:y

The root node is a vector accessible to all running programs to provide access to all global information. It is available in all versions of BCPL but many of its fields are only used in Cintpos. The global variable `rootnode` holds a pointer to the root node.

On some systems the address of the root node is also held in the manifest constant `rootnodeaddr`. Manifest constants starting with `rtn_` give the positions of the fields within the root node.

rtn_abortcode CIN:y, POS:y, NAT:n

This rootnode field holds the most recent return code from a command language interpreter (CLI). It is used by commands such as `dumpsys` and `dumpdebug` when inspecting Cintcode memory dumps.

rtn_adjclock CIN:y, POS:y, NAT:n

This rootnode field holds a correction in minutes to be added to the time of day supplied by the system. It is normally set to zero.

rtn_blklist CIN:y, POS:y, NAT:y

All blocks of memory whether free or in used are chained together in increasing address order. This rootnode field points to the first in the chain.

rtn_blib CIN:y, POS:y, NAT:n

Under Cintsys and Cintpos this rootnode field holds the appropriate versions of the modules `BLIB`, `SYSLIB` and `DLIB` chained together.

rtn_boot CIN:y, POS:y, NAT:n

Under Cintsys and Cintpos this rootnode field holds the appropriate version of the `BOOT` module.

rtn_boottrace CIN:y, POS:y, NAT:n

Under Cintsys and Cintpos this rootnode field holds 0, 1, 2 or 3 as set by the `-v` and `-V` options to trace the progress of booting the system.

rtn_bptaddr, rtn_bptinstr CIN:y, POS:y, NAT:n

These each hold vectors of 10 elements used by the standalone debugger to hold breakpoint addresses and operation codes overwritten by `BRK` instructions. They are in the rootnode to make them accessible to the debug task in Cintpos and to the `dumpdebug` command.

rtn_clkintson CIN:n, POS:y, NAT:n

Under Cintpos, this boolean field controls whether clock interrupts are enabled. It is provided to make single step execution possible within the interactive debugger without interference from clock interrupts. For more details see the chapter on the debugger starting on page 97.

rtn_clwkq CIN:n, POS:y, NAT:n

Under Cintpos, this field is used to holds the ordered list of packets waiting to be released by the clock device.

rtn_context CIN:y, POS:y, NAT:n

Under certain circumstances the entire Cintcode memory is dumped in a compacted form to the file `DUMP.mem` for later inspection by commands such as `dumpsys` and `dumpdebug`. This field is set at the time a dump file is written to specify why the dump was requested. The possible values are as follows:

1:	dump caused by second SIGINT
2:	dump caused by SIGSEGV
3:	fault in BOOT or standalone debug
4:	dump by user calling `sys(Sys_quit, -2)`
5:	dump caused by non zero user fault code
6:	dump requested from standalone debug

rtn_crntask CIN:y, POS:y, NAT:n

Under Cintpos, this rootnode field point to the TCB of the currently running task, which is the highest priority task that can run.

rtn_datestamp0 rtn_datestamp1 rtn_datestamp2 CIN:y, POS:y, NAT:n

These field hold the date stamp at the time the Cintcode memory was dumped to `DUMP.mem`.

rtn_dbgvars CIN:y, POS:y, NAT:n

This rootnode field holds vectors of 10 elements used by the standalone debugger to hold the debugger variables `V0` to `V9`. It is in the rootnode to make it accesibble to the debug task (in Cintpos) and to the `dumpdebug` command.

rtn_dcount CIN:y, POS:y, NAT:n

This holds a point to the debug count vector.

rtn_devtab CIN:y, POS:y, NAT:n

Under Cintpos, this holds the Cintpos device table. The zeroth entry is the table's upperbound and each other entries is either zero or points to the device control block (DCB) of the corresponding device.

rtn_dumpflag CIN:y, POS:y, NAT:n

If `dumpflag` is `TRUE` when Cintsys or Cintpos exits, the entire Cintcode memory is dumped in a compacted form to the file `DUMP.mem` for later inspection by commands such as `dumpsys` or `dumpdebug`.

rtn_envlist CIN:y, POS:y, NAT:n

This rootnode field holds the list of logical name-value pairs used by the functions `setlogval` and `getlogval`, and the CLI command `setlogval`.

rtn_hdrsvar CIN:y, POS:y, NAT:n

This field holds the name of the environment variable giving the directories holding BCPL headers, typically "BCPLHDRS" or "POSHDRS".

rtn_idletcb CIN:y, POS:y, NAT:n

This rootnode field holds the TCB of the IDLE task for used by the standalone debugger and the commands `dumpsys` and `dumpdebug`.

rtn_info CIN:y, POS:y, NAT:n

This rootnode field holds a vector of information that can be shared between all tasks. It is typically a vector of 50 elements. The use of these elements are system dependent.

rtn_insadebug CIN:n, POS:y, NAT:n

This rootnode field is used by the keyboard input device of Cintpos to tell it whether to place a newly received character in a request packet or just store it in the `lastch` field.

rtn_intflag CIN:y, POS:y, NAT:n

This flag is set to `TRUE` on receiving an interrupt from the user (typically a SIGINT signal generated by ctrl-C) and is reset to `FALSE` whenever the standalone debugger is entered. Cintsys or cintpos exits if a user interrupt is received when `intflag` is `TRUE` or if control is within `BOOT` or `sadebug`.

rtn_keyboard CIN:y, POS:y, NAT:n

This rootnode field holds the stream control block for the standard keyboard device.

rtn_klib CIN:y, POS:y, NAT:n

Under Cintpos this rootnode filed holds the the `KLIB` module. It is otherwise zero.

rtn_lastch CIN:n, POS:y, NAT:n

This rootnode field holds the most recent character received from the keyboard device. The standalone debugger uses it for polling input. On reading this field the standalone debugger resets it to `pollingch=-3`.

rtn_lastg, rtn_lastp, rtn_lastst CIN:y, POS:y, NAT:n

These rootnode fields hold the most recent settings of the Cintcode `P`, `G` and `ST` registers. They are used by the commands `dumpsys` and `dumpdebug` when inspecting a Cintcode memory dump caused by faults such as memory violation (SIGSEGV) when all other Cintcode dumped registers are invalid.

rtn_mc0, rtn_mc1, rtn_mc2, rtn_mc3 CIN:y, POS:y, NAT:n

These hold the machine address of the start of the Cintcode memory and other values used by the MC package.

rtn_membase, rtn_memsize CIN:y, POS:y, NAT:n

These rootnode fields hold, respectively, the start of the memory block chain and the upper bound in words of the Cintcode memory.

rtn_pathvar CIN:y, POS:y, NAT:n

This field holds the name of the environment variable giving the directories searched by loadseg, typically "BCPLPATH" or "POSPATH".

rtn_rootvar CIN:y, POS:y, NAT:n

This field holds the name of the environment variable holding the system root directory, typically "BCPLROOT" or "POSROOT".

rtn_scriptsvar CIN:y, POS:y, NAT:n

This field holds the name of the environment variable giving the directories holding CLI script files, typically "BCPLSCRIPTS" or "POSSCRIPTS".

rtn_screen CIN:y, POS:y, NAT:n

This rootnode field holds the stream control block for the standard screen device.

rtn_sys CIN:y, POS:y, NAT:n

Under Cintsys and Cintpos, this holds the entry point to the `sys` function.

rtn_tallyv CIN:y, POS:y, NAT:n

This rootnode field points to a vector used to hold profile execution counts. When tallying is enabled, the value of `tallyv!i` is the count of how often the Cintcode instruction at location `i` has been executed. The upper bound of `tallyv` is held in `tallyv!0`. For more information about the profile facility see the `stats` command described on page 87.

rtn_tasktab CIN:y, POS:y, NAT:n

Under Cintpos, this rootnode field holds the Cintpos task table. The zeroth entry is the table upperbound and the other entries are either zero or points to the task control block (TCB) of the corresponding task. Note that the IDLE task is not held in this table since it is not a proper task. The IDLE task TCB is held in the rootnode's `idletcb` field.

rtn_tcblist CIN:y, POS:y, NAT:n

Under Cintpos, all TCBs are chained together in decreasing priority order. This rootnode field points to the first TCB in this chain and so refers to the highest priority task. The last TCB on the chain has priority zero and represents the idle task.

rtn_trbuf, rtn_trword CIN:y, POS:y, NAT:y

The rootnode field `trbuf` points to the trace buffer and `trword` has value `#xBFBFBFBF`. It is used as a lightweight debugging aid. See the command `tracebuf` on page 88 for more details.

rtn_upb CIN:y, POS:y, NAT:n

This is the upperbound of the rootnode. It value is typically 50.

rtn_vecstatsv CIN:y, POS:y, NAT:n

This points to a vector holding counts of how many blocks of each requested size have been allocated by `getvec` but not yet returned. It is used by the `vecstats` command.

rtn_vecstatsvupb CIN:y, POS:y, NAT:n

This field hold the upper bound of `vecstatsv`.

saveregs CIN:n, POS:y, NAT:n

This is the location in Cintcode memory used in Cintpos to hold the Cintcode registers at the time of the most recent interrupt.

scb_... CIN:y, POS:y, NAT:n

Each currently open stream has a stream control block (SCB) that holds all that the system needs to know about the stream. Manifest constants beginning `scb_` allow convenient access to the SCB fields. These are described below.

scb_blength CIN:y, POS:y, NAT:n

This SCB field hold the length of the buffer in bytes. It is typically 4096.

scb_block CIN:y, POS:y, NAT:n

This SCB field holds the current block number of a disc file. The first block of a file has number zero.

scb_buf CIN:y, POS:y, NAT:n

This SCB field is either zero or points the the buffer of bytes associated with the stream.

scb_bufend CIN:y, POS:y, NAT:n

This SCB field holds the size of the buffer in bytes.

scb_encoding CIN:y, POS:y, NAT:n

This SCB field controls how `codewrch` treats extended characters written to this stream. If its value is GB2312, the extended character is translated into one or two bytes in GB2312 format, otherwise the translation is to a sequence of bytes in UTF-8 format. This field is normally set using either `codewrch(UTF8)` or `codewrch(GB2312)`.

scb_end CIN:y, POS:y, NAT:n

This SCB field hold the number of valid bytes in the buffer or -1, if the stream is exhausted.

scb_endfn CIN:y, POS:y, NAT:n

This SCB field is either zero or the function to close down the stream. It is given the SCB as its argument and returns `TRUE` if it successfully outputs the contents of the buffer. It otherwise returns `FALSE` with an error code in `result2`.

scb_fd CIN:y, POS:y, NAT:n

This SCB field holds a machine dependent file or mailbox descriptor.

scb_id CIN:y, POS:y, NAT:n

This SCB field holds one of the values `id_inscb`, `id_outscb` or `id_inoutscb`, indicating whether the stream is for input, output or both.

scb_lblock CIN:y, POS:y, NAT:n

This SCB field holds the number of last block. The first block of a stream is numbered zero.

scb_ldata CIN:y, POS:y, NAT:n

This SCB field holds the number of bytes in the last block of a stream.

scb_pos CIN:y, POS:y, NAT:n

This SCB field points to the position within the buffer of the next character to be transferred.

scb_rdfn CIN:y, POS:y, NAT:n

This SCB field is zero if the stream cannot perform input, otherwise it is the function to refill (or replenish) the buffer with more characters. It is given the SCB as its argument and returns `TRUE` if it successfully replenishes the buffer with at least one character. It otherwise returns `FALSE` setting `result2` to `-1` if the end of file has been encountered, `-2` if there was a timeout before any character were read, `-3` no character was available in polling mode. Any other value in `result2` is and error code.

scb_reclen CIN:y, POS:y, NAT:n

A file is normally regarded as a potentially huge sequence of bytes, but can also be treated as a sequence of fixed length records. The `reclen` SCB field hold the length in bytes of such records. The first record of a file has number zero. Unless the length of a file is a multiple of the record length, the length of last record of a file will be shorter than the specified record length.

scb_size CIN:y, POS:y, NAT:n

This constant is equal to the number of words in a stream control block.

scb_timeout CIN:y, POS:y, NAT:n

This SCB field holds the stream timeout value for TCP streams. If it is zero no timeout is applied. If it is negative, data is only tranferred if it is immediately available. If it is strictly positive it represents a timeout value in milli-seconds.

scb_timeoutact CIN:y, POS:y, NAT:n

This SCB field controls the effect of a time out on this stream while reading using `rdch`. A value of 0 causes the time out to be ignored, a value of `-1` caused the `rdch` to return with the value `endstreamch`, and a value of `-2` causes `rdch` to return with the value `timeoutch`.

scb_type CIN:y, POS:y, NAT:n

This SCB field holds the type of the stream which will be one of the following: `scbt_net`, `scbt_file`, `scbt_ram`, `scbt_console` or `scbt_mbx`, `scbt_tcp`. The last three have strictly positive values causing output to be triggered by end-of-line characters, while the first three are negative and only trigger output when the IO buffer is full. TCP streams have type `net` or `tcp`, streams to and from disk file have type `file`, stream to or from a vector in main memory have type `ram`, `mbx` specifies mailbox streams, and `console` indicates that the stream is either to standard output or from standard input which are normally the screen and keyboard, respectively.

scb_task CIN:y, POS:y, NAT:n

Under Cintpos, this SCB field holds either zero or the number of the handler task associated with the stream, if it has one.

scb_upb CIN:y, POS:y, NAT:n

This constant is the upperbound of a stream control block. its value is `scb_size-1`.

scb_wrfn CIN:y, POS:y, NAT:n

This SCB field is zero if the stream cannot perform output, otherwise it is the function to output (or deplete) the buffer. It is given the SCB as its argument and returns `TRUE` if it successfully outputs the contents of the buffer. It otherwise returns `FALSE` with an error code in `result2`.

scb_write CIN:y, POS:y, NAT:n

This SCB field is `TRUE` if the buffer has been updated by functions such as `wrch` since it was last written out (depleted).

scbt_net, scbt_file, scbt_ram, scbt_console, scbt_mbx, scbt_tcp
CIN:y, POS:y, NAT:n

These constants are mnemonics for the possible values of the `type` field of a stream control block. See `scb_type` above.

sectword CIN:y, POS:y, NAT:n

The first word of every loaded section is `sectword`. This are normally followed by a section name. Typically `sectword=#x0000FDDF`.

stackword CIN:y, POS:y, NAT:n

All words in runtime stacks are initialised to `stackword`. Typically `stackword=#xABCD1234`.

Sys_... CIN:y, POS:y, NAT:y

Manifest constants of the form `Sys_...` provide mnemonics for the operations invoked by the `sys` function. The use of these manifest constants is described in pages following Section 3.3 starting on page 51.

t_bhunk, t_bhunk64, t_end, t_end64, t_hunk, t_hunk64, t_reloc, t_reloc64
CIN:y, POS:y, NAT:n

These are constants identifying components of Cintcode object modules. Cintcode modules hold the relocatable byte stream interpretive code used by all BCPL interpretive systems. Constants with names ending with `64` are used in the 64-bit version of Cintcode. For more details, see the description of `loadseg` on page 54.

tickspersecond CIN:n, POS:y, NAT:n

Under Cintpos, the clock device ticks at a rate of **tickspersecond** ticks per second. Real time delays are specified in ticks, so, for example, a delay of 5 seconds can be achieved by the call: `sendpkt(notinuse, -1, 0, 0, 0, 5*tickspersecond)`. The second argument (`-1`) specifies the clock device, and the arg1 field (`5*tickspersecond`) specifies the real time delay in ticks. The value of `tickspersecond` is typically 50.

timeoutch CIN:n, POS:y, NAT:n

This is the value returned by `rdch` when a timeout occurs while trying to read from a stream. Its value is normally `-2`. Currently only TCP streams under Cintpos provide the timeout mechanism.

ug CIN:y, POS:y, NAT:y

This constant specified the first Global variable available to user programs. Currently `ug=200` so globals below this value are reserved for system use and the standard library. Since `ug` may change it would be wise to use it.

3.2 Global variables

This section describes the global variables declared in `libhdr.h`.

clihook CIN:y, POS:y, NAT:y

This function is used by command language interpreters to call the main function (`start`) of a newly loaded program. Its main purpose is to allow the user to set a breakpoint at the start of the next CLI command before `start` has been loaded into memory, see page 41.

cis, cos CIN:y, POS:y, NAT:y

These are, respectively, the currently selected input and output streams. Zero indicates that no stream is selected.

globsize CIN:y, POS:y, NAT:y

This variable is global zero and holds the size of the global vector.

result2 CIN:y, POS:y, NAT:y

This global variable is used by some functions to return a second result.

start CIN:y, POS:y, NAT:y

This is global 1 and is, by convention, the main function of a program. It is the first user function to be called when a program is run by the Command Language Interpreter.

userenv CIN:y, POS:y, NAT:y

This variable is available to the user to hold information that is preserved from one CLI command to the next. The standard command language interpreter resets all global variable from `ug` to the end of the global vector between commands. `userenv` is not in this region of the global vector and so is preserved. Normally `userenv` is either zero or points to a user defined structure holding environmental data.

currco CIN:n, POS:y, NAT:n

This points to the currently executing coroutine.

rootnode CIN:n, POS:y, NAT:n

This points to the rootnode.

colist CIN:n, POS:y, NAT:n

This holds the list of currently existing coroutines.

returncode CIN:n, POS:y, NAT:n

This holds the return code of the command most recently executed by the current command language interpreter.

currentdir CIN:n, POS:y, NAT:n

This holds the name of the current working directory.

randseed CIN:n, POS:y, NAT:n

This is the seed used by the random number generator `randno`.

tcb CIN:n, POS:y, NAT:n

Under Cintpos this is a pointer to the currently executing task.

taskid CIN:n, POS:y, NAT:n

Under Cintpos this is the identifier of the currently executing task.

pktlist CIN:n, POS:y, NAT:n

Under Cintpos when running in multi-event mode, `pktlist` contains mapping information from packet to its corresponding coroutine.

consoletask CIN:n, POS:y, NAT:n

This is a variable used by command language interpreters.

multi_count CIN:n, POS:y, NAT:n

This is a variable used in the implementation of `gomultievent` under Cintpos.

mainco_busy CIN:n, POS:y, NAT:n

This is a variable used in the implementation of `gomultievent` under Cintpos.

3.3 Global functions

One of the main purposes of the global vector is hold entry points of functions defined in one module and used in a different module. This section describes the function defined in the standard resident library. Most of these are defined in BCPL in the files: `sysb/klib.b`, `sysb/blib.b` and `sysb/dlib.b`, one library (`cin/syscin/syslib`) is in hand written Cintcode since it contains instructions that cannot be generated by the BCPL compiler. The functions defined in `syslib` are `sys`, `changeco` and `muldiv`.

The standard library functions are described in alphabetical order.

`abort(`*`code`*`)` CIN:y, POS:y, NAT:n

This procedure causes an exit from the current invocation of the interpreter, returning *code* as the error code. If *code* is zero execution exits from the Cintcode system. If *code* is `-1` execution resumes using the faster version of the interpreter (`fasterp`). If *code* is `-2` the entire Cintcode memory is written to file `DUMP.mem` is a compacted form for processing by CLI commands such as `dumpsys` or `dumpdebug`. If *code* is positive, under normal conditions, the interactive debugger is entered.

res `:= callco(`*cptr*`, `*arg*`)` CIN:y, POS:y, NAT:y

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 `:= callseg(`*name*`, `*a1*`, `*a2*`, `*a3*`, `*a4*`)` CIN:y, POS:y, NAT:y

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.

ch `:= capitalch(`*ch*`)` CIN:y, POS:y, NAT:y

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

res `:= clihook()` CIN:y, POS:y, NAT:y

This function simply calls `start` and returns its result. 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. Occassionally, a user may find it useful to override the standard definition of clihook with a private version.

`codewrch(`*code*`)` CIN:y, POS:y, NAT:y

This routine uses `wrch` to write the Unicode character *code* as a sequence of bytes in either UTF8 or GB2312 format. If the `encoding` field of the current output stream is `UTF8`, the output is in UTF8 format as described in the following table.

Code range	Binay value	UTF8 bytes
`0-7F`	`zzzzzzz`	`0zzzzzzz`
`80-7FF`	`yyyyyzzzzzz`	`110yyyyy 10zzzzzz`
`800-FFFF`	`xxxxyyyyyyzzzzzz`	`1110xxxx 10yyyyyy 10zzzzzz`
`1000-1FFFFF`	`wwwxxxxxxyyyyyyzzzzzz`	`11110www 10xxxxxx 10yyyyyy 10zzzzzz`
etc	etc	etc

If the `encoding` field of the current output stream is `GB2312`, the output is in GB2312 format as described in the following table.

Decimal range	GB2312 bytes
`0 < dd < 127`	`<dd>`
`128 < xxyy < 9494`	`<xx+160> <yy+160>`

res `:= compch(`*ch1*`, `*ch2*`)` CIN:y, POS:y, NAT:y

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*`)` CIN:y, POS:y, NAT:y

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 the strings are equal.

res `:= cowait(`*arg*`)` CIN:y, POS:y, NAT:y

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.

cptr `:= createco(`*fn*`, `*size*`)` CIN:y, POS:y, NAT:y

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 quanitities. 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.

Coroutines have distinct stacks but share the same global vector, and it is natural to represent them by pointers to their stacks. At the base of each stack there are six words of system information as shown in figure 3.1.

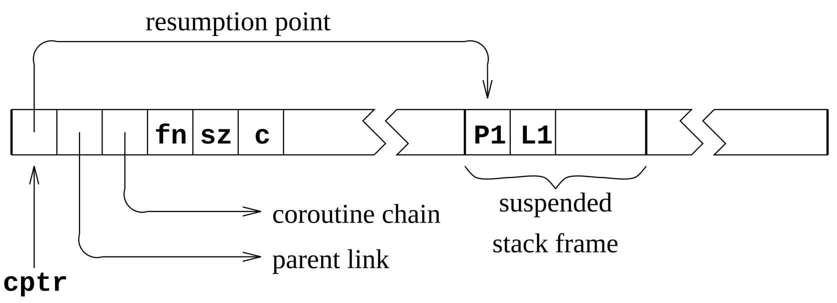

Figure 3.1: A coroutine stack

The resumption point is 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 `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 Cintcode P register points to a stack frame within its stack. Passing control from one coroutine to another involves saving the resumption point in the

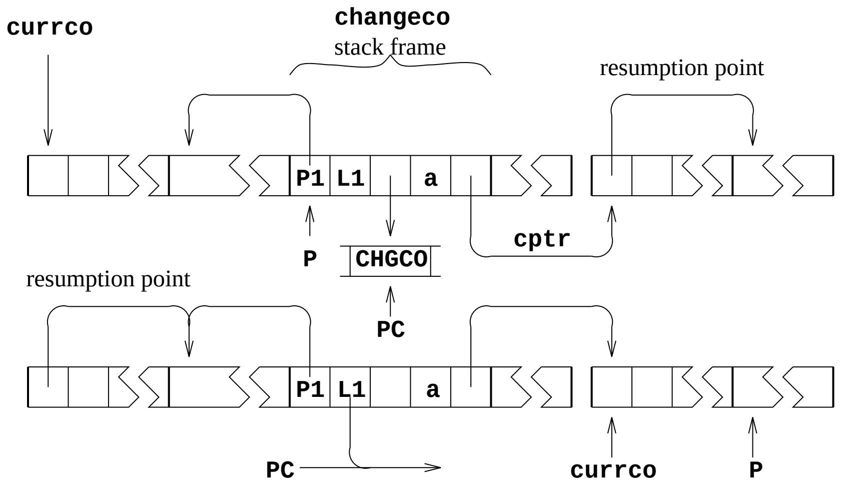

Figure 3.2: The effect of `changeco(a, cptr)`

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 3.2. The function `changeco` is defined by hand in `syslib` and its body consists of the single Cintcode 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 these are the only functions that cause coroutine suspension.

The function `createco` creates a new coroutine with a given main function *fn* and stack size *size* leaving it suspended in the call of `cowait` in the following loop.

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

When control is next transfered to this 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 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 six locations ready for the call of `changeco(0,c)` that follows. The state just after this call is shown in figure 3.3. 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*`)` CIN:y, POS:y, NAT:y

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.

flag `:= deletefile(`*name*`)` CIN:y, POS:y, NAT:y

This call deletes the named file, returning if successful, and `FALSE` otherwise.

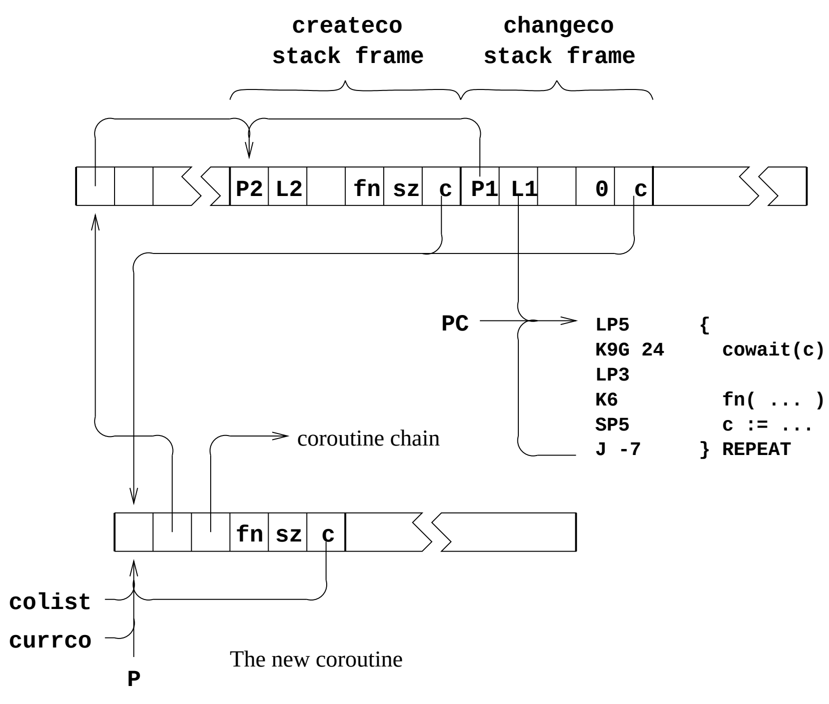

Figure 3.3: The state just after `changeco(0,c)` in `createco`

`endread()` CIN:y, POS:y, NAT:y

This routine closes the currently selected input stream by calling `endstream(cis)`.

`endstream(`*`scb`*`)` CIN:y, POS:y, NAT:y

This routine closes the stream whose control block is *scb*.

`endwrite()` CIN:y, POS:y, NAT:y

This routine closes the currently selected output stream by calling `endstream(cos)`.

n `:= findarg(`*`keys`*`, `*`item`*`)` CIN:y, POS:y, NAT:y

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`.

scb `:= findinput(`*`name`*`)` CIN:y, POS:y, NAT:y

This function 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 3.3.2 for information about the treatment of filenames.

scb `:= findoutput(`*`name`*`)` CIN:y, POS:y, NAT:y

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 3.3.2 for information about the treatment of filenames.

v := `getvec(`*upb*`)` CIN:y, POS:y, NAT:y

This function allocates of space using 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. 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.

`getvec` 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. A vector previously allocated by `getvec` can be freed by the above call of `freevec`. Coalescing of adjacent free blocks is performed by `getvec`.

An extra word is allocated just before the start of each block to hold its size, and four or five words are added to the end of each block and filled with special data that is checked when the block is returned to free store. This catches many common space allocation errors.

res := `globin(`*segl*`)` CIN:y, POS:y, NAT:y

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*.

cptr := `initco(`*fn*`, `*size*`, `*a*`,...)` CIN:y, POS:y, NAT:y

This function provides a convenient method of creating 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)
  TEST cptr
  THEN result2 := callco(cptr, @a)
  ELSE result2 := 0
  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 newly created coroutine has initialised itself, it returns control to `initco` by means a call of `cowait`. The result of `initco` is the newly created coroutine pointer, or zero on failure. The second result (in `result2`) is the value returned by the first call of `cowait` in the newly created coroutine.

scb := `input()` CIN:y, POS:y, NAT:y

This function returns `cis`, the SCB of the currently selected input stream.

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

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`. The value returned by *fn*(a,b,c,d,e,f,g,h,i,j,k) is saved by `instrcount` in the global variable `result2`.

flag `:= intflag()` CIN:y, POS:y, NAT:n

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.

P `:= level()` CIN:y, POS:y, NAT:y

This call returns the current stack frame pointer(P) for use in a later call of `longjump`.

segl `:= loadseg(`*name*`)` CIN:y, POS:y, NAT:n

This function calls `sys(Sys_loadseg,` *name*`)` to loads the specified compiled program into memory. See `Sys_loadseg` on page 54 for details.

The remaining 4 words contain global initialisation data indicating that global 1(`00000001`) is to be set to the entry point at position 36 (`00000024`) relative to the start of the hunk, and that the highest referenced global number is 28 (`0000001C`).

`longjump(`P`,` L`)` CIN:y, POS:y, NAT:y

This call causes execution to resume at label L in the body of a procedure that owns the stack frame given by P that must have been obtained by a previous call of `level`. Jumps may only be used to points within the current coroutine. 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.

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

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

```
LET mkobj(upb, fns, a, b, c, d, e, f, g, h, i, j, k) = VALOF
{ LET obj = getvec(upb)
  IF obj DO
  { !obj := fns
    InitObj#(obj, @a) // Send the init message to the object
  }
  RESULTIS 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 of up to 11 initialisation arguments.

res `:= muldiv(`*a*`, `*b*`, `*c*`)` CIN:y, POS:y, NAT:y

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.

Remember that scaled fixed point values can be output conveniently using `writef` as in:

```
writef("%10.6d*n", 123_456789)
```

which will output the following:

```
123.456789
```

`newline()` CIN:y, POS:y, NAT:y

This simply outputs the newline character (`'*n'`) to the currently selected output stream.

`newpage()` CIN:y, POS:y, NAT:y

This simply outputs the newline character (`'*p'`) to the currently selected output stream.

scb `:= output()` CIN:y, POS:y, NAT:y

This function returns `cos`, the SCB of the currently selected output stream.

scb `:= pathfindinput(`*name*`, `*pathname*`)` CIN:y, POS:y, NAT:y

This function opens an input stream. If *name* is the string `"*"` then input comes from standard input which is normally the keyboard, otherwise *name* is taken to be a filename. If `name` is a relative file name and `pathname` is non zero, the directories specified by the shell variable *pathname* are searched. If `pathname` is zero the filename is looked up in the current directory. If the file cannot be opened `pathfindinput` returns zero.

n `:= randno(`*upb*`)` CIN:y, POS:y, NAT:y

This function returns a random integer in the range `1` to `upb`. It uses a seed held in global variable `randseed` which can be set using `setseed` described below. Its implementation is as follows:

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

res := `rdargs(`*`keys, argv, upb`*`)` CIN:y, POS:y, NAT:y

This implementation of BCPL incorporates a command language interpreter which is described in Chapter 4. Most commands require argument and these are easily read using this function.

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*) into 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, but 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 3.4 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.

ch := `rdch()` CIN:y, POS:y, NAT:y

This call reads the next character from the currently selected input stream. If the stream is exhausted, it returns the special value `endstreamch`. Input from the keyboard is buffered until the ENTER (or RETURN) key is pressed to allow simple line editing

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

in which the backspace key may be used to delete the most recent character typed. See Section 3.3.1 for more detailed information.

kind := `rditem(`*v*`,` *upb*`)` CIN:y, POS:y, NAT:y

This function is usually called from `rdargs` to read an item from the currently selected input stream. After ignoring leading spaces and tabs, it packs the item into the vector *v* whose upper bound is *upb* and returns an integer describing the kind of item read. Table 3.5 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 3.5: `rditem` results

Within quoted strings `*n` represents the newline character, `*s` represents a space, `**` represents an asterisk and `*"` represents a a double quote character.

flag := `renamefile(`*oldname*`,` *newname*`)` CIN:y, POS:y, NAT:y

The call renames the file *oldname* as file *newname*, deleting *newname* if necessary, returning `TRUE` if the renaming was successful, and `FALSE` otherwise. Both *oldname* and *newname* are strings.

n := `readn()` CIN:y, POS:y, NAT:y

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, it 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.

res `:= resumeco(`*cptr*`,` *arg*`)` CIN:y, POS:y, NAT:y

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.

ch `:= sardch()` CIN:y, POS:y, NAT:y

This function calls `sys(Sys_sardch)` to read the next character from the keyboard as soon as it is available, echoing the character to the screen.

`sawrch(`*ch*`)` CIN:y, POS:y, NAT:y

This function calls `sys(Sys_sawrch(`*ch*`)` to write the specified character to the screen.

`sawritef(`*format*`,` *a*`,` *b*`, ...)` CIN:y, POS:y, NAT:y

This function is similar to `writef` but performs its output using `sawrch`.

`selectinput(`*scb*`)` CIN:y, POS:y, NAT:y

This call executes `cis :=` *scb* to select *scb* as the current input stream. It aborts (with code 186) if *scb* is not an input stream.

`selectoutput(`*scb*`)` CIN:y, POS:y, NAT:y

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

oldseed `:= setseed(`*newseed*`)` CIN:y, POS:y, NAT:y

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

code `:= start(`*a1*`,` *a2*`,` *a3*`,` *a4*`)` CIN:y, POS:y, NAT:y

This function is, by convention, the main procedure of a program. If it is called from the command language interpreter (see section 4), 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

`stop(`*code*`)` CIN:y, POS:y, NAT:y

This function is provided to stop the execution of the current command running under control of the CLI. Its argument, *code*, is the command's completion code and is assigned to `returncode`.

n `:= str2numb(`*str*`)` CIN:y, POS:y, NAT:y

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

res `:= sys(`*op*`,...)` CIN:y, POS:y, NAT:y

The file `sysc/cintsys.c` contains the main program of the Cintsys system. It also includes the definition of an important function `dosys` which provide access to I/O operations and many other operating system primitives. The file `sysc/cinterp.c` contains a C implementation of the Cintcode interpreter. With different compile time settings this file can generate a faster version by reducing the number of debugging aids present. Sometimes there is an even faster version of the interpreter implemented in assembly language, see, for instance, `sysasm/linux/cintasm.s`. The BCPL function `sys` provides an interface between BCPL and `dosys`.

The file `sysc/cintpos.c` contains the main program of the Cintpos system. It has much is common with `sysc/cintsys.c` including the function `dosys`.

The `sys` function is defined by hand in `cin/syscin/syslib` and just invokes the `SYS` Cintcode instruction. When `SYS` is encountered by the interpreter, it normally just calls `dosys` passing the BCPL P and G pointers as arguments. But certain `sys` operations such as `sys(Sys_quit,code)` are processed directly by the interpreter.

As might be expected there are many `sys` operations concerned with interrupts that are only available under Cintpos.

res `:= sys(Sys_buttons)` CIN:y, POS:y, NAT:y

res `:= sys(Sys_callc,` *fno*`,` *a1*`,` *a2* `...)` CIN:y, POS:y, NAT:y

This calls `cfuncs(args, g)` where `cfuncs` is a C function defined in `sysc/cfuncs.c`. The argument `args` points to memory locations holding *fno*, *a1*, *a2*, etc., and `g` points to the base of the global vector.

The following table summarises the callc operations currently available (when running under Linux).

Function number	Purpose
c_name2ipaddr	*a1* = name or dotted decimals of a host
	`res` is the IP address of the host
	or -1 if error.
c_name2port	*a1* = name or decimals of a port
	`res` is the port number
	or -1 if error.
c_newsocket	`res` is the port number
	or -1 if error.
c_reuseaddr	*a1* = the file descriptor of a socket
	a2 = 1 means that the socket can be reused
	`res` = -1 if error
c_setsndbufsz	*a1* = the file descriptor of a socket
	a2 = size to set the send buffer for this socket
	`res` = -1 if error
c_setrcvbufsz	*a1* = the file descriptor of a socket
	a2 = size to set the recv buffer for this socket
	`res` = -1 if error
c_bind	*a1* = the file descriptor of a socket
	a2 = the remote IP address for this socket
	a3 = the remote port number for this socket
	`res` = -1 if error
c_tcpconnect	*a1* = the file descriptor of a socket
	a2 = the remote IP address for this socket
	a3 = the remote port number for this socket
	`res` = -1 if error
c_tcplisten	*a1* = the file descriptor of a socket
	a2 = the maximum number of connections waiting to be accepted
	`res` = -1 if error
c_tcpaccept	*a1* = the file descriptor of a socket
	`res` = the file descriptor of the socket to use for the accepted connection
	or -1 if error
c_tcpclose	*a1* = the file descriptor of the socket to be closed
	`res` = -1 if error
c_fd_zero	*a1* = pointer to a vector bits
	`res` = -1 if error
c_fd_set	*a1* = the number of the bit to set
	a2 = pointer to a vector bits
	`res` = -1 if error
c_fd_isset	*a1* = the number of the bit to test
	a2 = pointer to a vector bits
	`res` = 1 if the bit was set, 0 otherwise
c_fd_select	*a1* = the number of the bit to test
	a2 = pointer to a vector bits identifying read sockets of interest
	a3 = pointer to a vector bits identifying write sockets of interest
	a4 = pointer to a vector bits identifying sockets of interest
	a5 = pointer to two words holding the timeout in seconds and microseconds
	`res` = the number of sockets that can now be read or written to, or 0 if the timeout pe

res := `sys(Sys_callnative,` *f*, *a1*, *a2*, *a3*`)` CIN:y, POS:y, NAT:y

This function is used to enter a subroutine in native machine code.

res := `sys(Sys_close,` *fp*`)` CIN:y, POS:y, NAT:y

This closes the file whose file pointer is *fp*. It return 0 if successful.

res := `sys(Sys_cputime)` CIN:y, POS:y, NAT:y

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

res := `sys(Sys_delay,` *ticks*`)` CIN:y, POS:y, NAT:y

Under Cintpos, this call returns after the spcified delay given by *ticks*. The manifest constant `tickspersecond` specifies how many ticks there are in one second.

res := `sys(Sys_deletefile,` *name*`)` CIN:y, POS:y, NAT:y

This deletes the file whose name is given by `name`. See page 65 for information about the treatment of file names.

res := `sys(Sys_devcom,` *com*, *arg*`)` CIN:y, POS:y, NAT:y

res := `sys(Sys_dumpmem,` *context*`)` CIN:y, POS:y, NAT:y

`sys(Sys_freevec,` *ptr*`)` CIN:y, POS:y, NAT:y

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,...)`. It checks that the block is not already free and that it has not been corrupted.

res := `sys(Sys_filemodtime,` *name*`)` CIN:y, POS:y, NAT:y

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

res := `sys(Sys_filesize,` *fd*`)` CIN:y, POS:y, NAT:y

This call return the size in bytes of the currently opened disk file whose file descriptor is *fd*. The file descriptor is typically obtained by the expression `scb!scb_fd`.

res := `sys(Sys_ftime)` CIN:y, POS:y, NAT:y

res := `sys(Sys_getpid)` CIN:y, POS:y, NAT:y

This function returns the process id of the currently executing process.

str := `sys(Sys_getprefix)` CIN:y, POS:y, NAT:y

This returns the current prefix string. See `sys(Sys_setprefix,...)` on page 58.

res := `sys(Sys_getsysval,` *addr*`)` CIN:y, POS:y, NAT:y

This function return the contents of the machine memory location whose address is *addr*.

res `:= sys(Sys_getvec,` *upb*`)` CIN:y, POS:y, NAT:y

This allocates a vector whose lower bound is 0 and whose upper bound is *upb*. It return zero if the request cannot be satisfied. A word is allocated just before the start of the vector to hold its size, and several (4 or 5) words are allocated just past the end of the vector and filled with redundant data that is checked when the space is returned to free store.

res `:= sys(Sys_globin,` *seg*`)` CIN:y, POS:y, NAT:n

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

res `:= sys(Sys_graphics,...)` CIN:y, POS:y, NAT:y

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_inc,` *addr*`,` *amount*`)` CIN:y, POS:y, NAT:y

This function adds *amount* atomically to the specified memory location and returns it new value.

res `:= sys(Sys_interpret,` *regs*`)` CIN:y, POS:y, NAT:n

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:

res `:= sys(Sys_intflag)` CIN:y, POS:y, NAT:y

This returns `TRUE` if the user has pressed a particular combination of keys to interrupt the program that is currently running. On many systems this mechanism not implemented and so just returns `FALSE`.

res `:= sys(Sys_loadseg,` *name*`)` CIN:y, POS:y, NAT:n

This loads a Cintcode module from file *name*. Under `cintsys`, if *name* is a relative file name it searches the current directory followed by the directories specified by the environment variable `BCPLPATH`. If these all fail, `cin/` is prepended to the file name which is then looked up in the directory specified by the `BCPLROOT` environment variable. Under `cintpos`, the environment variables `POSPATH` and `POSROOT` are used. The `-cin` option can be used with either `cintsys` or `cintpos` to override these defaults. To check that these variables are set correctly enter `cintsys` or `cintpos` with the `-f` option.

If loading is successful, `loadseg` returns the list of loaded program sections, otherwise it returns zero. Before the loaded code can be used, its globals must be initialised using `globin`.

Cintcode modules generated by the BCPL compiler are typically text files containing the compiled code encoded in hexadecimal (although a purely binary representation is available). The compiled form of the `logout` command:

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

is

```
000003E8 0000000E
0000000E 0000FDDF 474F4C0B 2054554F 20202020
0000DFDF 6174730B 20207472 20202020 7B1C2310
00000000 00000001 00000024 0000001C
```

The first two words (`000003E8 0000000E`) indicate the presence of a "hunk" of code of size 14(`000000E`) words which then follow. The first word of the hunk (`000000E`) is again its length. The next four words (`0000FDDF 474F4C0B 2054554F 20202020`) contain the `SECTION` name `"logout"`. These are followed by the four words `0000DFDF 6174730B 20207472 20202020` which hold the name of the procedure `"start"`. The body of `start` is compiled into one word (`7BF1C2310`) which correspond to the Cintcode instructions:

`L0`	Load A with 0
`K3G 28`	Call the function in global 28, incrementing the stack by 3
`RTN`	Return from `start` – never reached

`sys(Sys_lockirq)` CIN:y, POS:y, NAT:y

Under cintpos, this call disables interrupts.

res `:= sys(Sys_muldiv,` *a*`,` *b*`,` *c*`)` CIN:y, POS:y, NAT:y

This invoke 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. This function is little used since the `muldiv` function is now defined in `syslib` invoking the Cintcode instruction `MDIV`.

fp `:= sys(Sys_openread,` *name*`,` *envname*`)` CIN:y, POS:y, NAT:y

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 65 for information about the treatment of file names. If *name* is a relative filename, the file is first searched for in the current directory, otherwise, if *envname* is non null, the directories specified by the environment variable *envname* are searched.

res `:= sys(Sys_openreadwrite,` *name*`)` CIN:y, POS:y, NAT:y

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 65 for information about the treatment of file names. An opened file can be thought of as a potentially huge vector of bytes with a pointer to the position of the next byte to transfer. Reading or writing bytes from or to the file cause this pointer to be advanced by the appropriate amount. This pointer can be explictly set using `sys(Sys_seek,...)` described below and its current value obtained by `sys(Sys_tell,...)` also described below.

fp `:= sys(Sys_openwrite,` *`name`*`)` CIN:y, POS:y, NAT:y

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 65 for information about the treatment of file names.

res `:= sys(Sys_platform)` CIN:y, POS:y, NAT:y

res `:= sys(Sys_putsysval,` *`addr`*`,` *`val`*`)` CIN:y, POS:y, NAT:n

`sys(Sys_quit,` *`code`*`)` CIN:y, POS:y, NAT:n

This saves the Cintcode registers in the vector of registers given to the interpreter when it was invoked 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)` above). A *code* of zero denotes successful completion and, if invoked at the outermost level, causes the BCPL Cintcode System to terminate.

n `:= sys(Sys_read,` *`fp`*`,` *`buf`*`,` *`len`*`)` CIN:y, POS:y, NAT:y

This reads 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,...)`. The number of bytes actually read is returned as the result.

res `:= sys(Sys_renamefile,` *`old`*`,` *`new`*`)` CIN:y, POS:y, NAT:y

This renames file *old* to *new*. It return 0 if successful.

`sys(Sys_rti,` *`regs`*`)` CIN:n, POS:y, NAT:n

Under Cintpos, this returns from an interrupt by setting the Cintcode registers to the values specified by *regs*.

ch `:= sys(Sys_sardch)` CIN:y, POS:y, NAT:y

This returns the next character from standard input (normally the keyboard). The character is echoed to standard output (normally the screen). If the `-c` or `--` command options are given when `cintsys` or `cintpos` is invoked, standard input is prefixed with text from the command line. For details, see Section 11.2 on page 156.

`sys(Sys_saveregs,` *`regs`*`)` CIN:n, POS:y, NAT:n

Under Cintpos, this saves the current Cintcode registers in *regs*.

`sys(Sys_sawrch,` *`ch`*`)` CIN:y, POS:y, NAT:y

This sends character repesented by the least significant 8 bit of *ch* to the standard output (normally the screen). If *ch=10*, the characters carriage return followed by linefeed are transmitted.

res `:= sys(Sys_seek,` *fd*, *pos*`)` CIN:y, POS:y, NAT:y

oldcount `:= sys(Sys_setcount,` *newcount*`)` CIN:y, POS:y, NAT:n

One of the Cintcode registers is called `count` which is inspected just before the interpreter processes the next instruction. If `count>0` it is decremented and the instruction processed. If `count=0` the interpreter returns to the calling (C) program with error code `3`.

The Cintcode System normally has two resident interpreters. One is called `cinterp` implemented in C and the other is called `fasterp` which is sometimes implemented in assembly language. `fasterp` is faster than `cinterp` since it provides fewer debugging aids, does not count instruction executions and does not implement the profiling feature. Setting `count` to a negative value causes this faster interpreter to be invoked and setting `count` to a positive value causes the slower interpreter to be used. Normally the CLI command `interpreter` is used to make this switch, see Section 4.3.

With some debugging versions of `fasterp`, setting `count` to `-2` causes it to execute just one instruction before returning with error code `10`. This feature assists the debugging of a new versions of `fasterp` and is particularly useful when `fasterp` is implemented in assembly language.

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 (unused)
regs`!6`	PC register	– the program counter
regs`!7`	Count register	– see below
regs`!8`	MW register	– Used only on 64-bit systems, see below

The count register is normally decremented every time a Cintcode instruction is interpreted. When the count reaches zero the interpreter saves the registers and returns with a result (=3) to indicate that this 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 `fasterp` 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. The returned result is either the second argument of `sys(Sys_quit,...)` or one of the builtin 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 existant 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 value of the watched location in the Cincode memory has changed in the course of executing the previous instruction

res `:= sys(Sys_setprefix,` *`prefix`*`)` CIN:y, POS:y, NAT:y

This is primarily a function for the Windows CE version of the BCPL Cintcode System for which there is no current working directory mechanism. It sets the prefix that is prepended to all future relative file names. See Section 3.3.2 and the CLI `prefix` command described on page 82.

res `:= sys(Sys_setraster,` *n*`,` *`arg`*`)` CIN:y, POS:y, NAT:n

There is a variant of `cintsys` called `rastsys` that provides a means of generating data for time-memory images, and `cintpos` has a similar variant called `rastpos`. The `setraster` operation controls the rastering feature as follows. 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 being 12. If *n*`=1`, the number of Cintcode instructions executed per raster line is set to *arg*, the default being 1000. If *n* is zero and *arg* is non-zero then rastering is activated sending its output to the file with name *arg* (the rastering data file). Raster information is normally 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 text file that encodes the raster lines using run length encoding. Typical output is as follows:

`K1000 S12`	1000 instruction 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 84 for more information on how to use the rastering facility.

res `:= sys(Sys_sound,` *`fno`*`,` *`a1`*`,` *`a2`* `...)` CIN:y, POS:y, NAT:y

This calls `sound(args, g)` where `sound` is a C function defined in `sysc/sound.c`. The argument `args` points to memory locations holding *fno*, *a1*, *a2*, etc., and `g` points to the base of the global vector.

The following table summarises the sound features available (when running under Linux).

Function number	Purpose
fno=0	Test for sound *res* is TRUE if the sound feature is implemented.
fno=1	Open sound device for input *a1* = typically "/dev/dsp" or "/dev/dsp1" *a2* = sample format, eg 16 for `S16_LE`, 8 for `U8` *a3* = channels, typically 1 for mono or 2 for stereo *a4* = rate ie samples per second, typically 44100 `res` is the file descriptor (an integer) of the opened device or -1 if error.
fno=2	Read samples *a1* = the file descriptor *a2* = the buffer *a3* = the number of bytes to read *res* = the number of bytes transferred into the buffer
fno=3	Close a sound device *a1* = the file descriptor
fno=4	Open a sound device for output *a1* = typically "/dev/dsp" or "/dev/dsp1" *a2* = sample format, eg 16 for `S16_LE`, 8 for `U8` *a3* = channels, typically 1 for mono or 2 for stereo *a4* = rate ie samples per second, typically 44100 *res* is the file descriptor (an integer) of the opened device or -1 if error.
fno=5	Write samples *a1* = the file descriptor *a2* = the buffer *a3* = the number of bytes to write *res* = the number of bytes actually transferred
fno=6	Open a sound device for output *a1* = typically "/dev/midi" or "/dev/dmmidi1" *res* is the file descriptor (an integer) of the opened device or -1 if error.
fno=7	Write MIDI bytes *a1* = the file descriptor *a2* = the buffer *a3* = the number of MIDI bytes to write *res* = the number of bytes actuallty transferred

Note that it may be necessary to run alsamixer to enable the sound device and adjust its volume setting.

`sys(Sys_setst, `*`val`*`)` CIN:n, POS:y, NAT:n

Under Cintpos, this sets the Cintcode ST register to *val*. Interrupts are enabled only when ST is zero. By convention, ST=1 why execution within `klib`, ST=2 when executing within the interrupt routine, and ST=3 during the initial bootstrapping process.

res `:= sys(Sys_shellcom, `*`comstr`*`)` CIN:y, POS:y, NAT:y

`sys(Sys_tally, `*`val`*`)` CIN:y, POS:y, NAT:n

This call provides a profiling facility that uses a globally accessible tally vector to hold frequency counts of Cintcode instructions executed. When *val* is `TRUE` the tally vector is cleared and tallying is enabled. When *val* is `FALSE` tallying is disabled. When tallying is active, 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 by the `-t` command line argument (see Section 11.2) when the interpreter is entered. The default size being typically 80000 words. The tally vector is held in `rootnode!rtn_tallyv` with the upper bound stored in its zeroth element. It can thus be inspected by any program.

Statistics of program execution is normally gathered and analysed using the CLI command `stats` (see Section 4.3).

pos `:= sys(Sys_tell, `*`fd`*`)` CIN:y, POS:y, NAT:y

`sys(Sys_tracing, `*`val`*`)` CIN:y, POS:y, NAT:n

This sets the Cintcode tracing mode to *val*. When the tracing mode is `TRUE`, the Cintcode interpreter outputs a one line trace of every Cintcode instruction executed.

res `:= sys(Sys_unloadseg, `*`seg`*`)` CIN:y, POS:y, NAT:y

This unloads the the loaded module given by *seg*. If *seg* is zero it does nothing. Unloading a module just returns the space it occupied to freestore.

`sys(Sys_unlockirq)` CIN:n, POS:y, NAT:n

Under cintpos, this call enables interrupts.

res `:= sys(Sys_usleep, `*`usecs`*`)` CIN:y, POS:y, NAT:y

Under cintsys, this call causes the system to sleep for `usecs` micro-seconds. Under cintpos, it causes the current task to sleep for `usecs` micro-seconds.

`sys(Sys_waitirq)` CIN:y, POS:y, NAT:y

`sys(Sys_watch, `*`addr`*`)` CIN:y, POS:y, NAT:n

This sets the address of a location of Cintcode memory to be inspected every time the interpreter executes and instruction. When the watched value changes it returns with result 12. The watch feature is disabled if *addr* is zero or if `fasterp` is being used.

n `:= sys(Sys_write,` *fp*`,` *buf*`,` *len*`)` CIN:y, POS:y, NAT:y

This writes *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,...)`. The result is the number of bytes transferred, or zero if there was an error.

`unloadseg(`*segl*`)` CIN:y, POS:y, NAT:y

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

res `:= unrdch()` CIN:y, POS:y, NAT:y

This 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` where single character lookahead is necessary. See Section 3.3.1 for more detailed information.

`wrch(`*ch*`)` CIN:y, POS:y, NAT:y

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.

`writed(`*n*`,` *d*`)` CIN:y, POS:y, NAT:y
`writeu(`*n*`,` *d*`)` CIN:y, POS:y, NAT:y
`writen(`*n*`)` CIN:y, POS:y, NAT:y

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*`)` CIN:y, POS:y, NAT:y
`writeoct(`*n*`,` *d*`)` CIN:y, POS:y, NAT:y
`writebin(`*n*`,` *d*`)` CIN:y, POS:y, NAT:y

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*`)` CIN:y, POS:y, NAT:y
`writet(`*str*`,` *d*`)` CIN:y, POS:y, NAT:y

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.

`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*`)`
CIN:y, POS:y, NAT:y

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 3.6.

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`. Format characters are case insensitive but field

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`.
`%#`	Write the next argument as an extended in UTF-8 or GB2312 format using `codewrch`.
`%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`.
`%`*n*`.`*m*`d`	Write the next argument as a scaled decimal number in a field with of *n* with *m* digits after the decimal point.
`%+`	Skip over the next argument.
`%-`	Step back to the previous argument.
`%%`	Write the character `%`.
`%p`*c*	Plural formation. Write character *c* if the next argument is not 1.
`%p\`*a*`\`*b*`\`	Plural formation. Write text *a* if the next argument is 1, otherwise write text `b`.
`%f`	Take the next argument as a `writef` format string and call `writef` recursively to process it passing it the remaining arguments. The argument pointer is advanced by the appropriate amount
`%m`	The next arument is taken as a message number and processes as for `%f` above using the message format string obtained by the call `get_text(messno, str, upb)` where `str` is a vector local to `writef` to hold the message string. This provides an easy way to generate messages in different languages. `get_text` is a global function typically defined by the user. The default version always yields the message string `"<mess:%-%n>"`

Figure 3.6: `writef` substitution items

width characters are not. A recent entension allows the field width to be specified as a decimal integer immediately following the percent, as in `%12i` meaning `%iB`.

Some examples of the `%`n`.`m`d` substitution item are given below.

```
writef("%9.2d", 1234567)    writes   12345.67
writef("%9.2d", -1234567)   writes  -12345.67
writef("%9.0d", 1234567)    writes    1234567
writef("%9d", 1234567)      writes    1234567
```

As an example of how the `%p` substitution item can be used, the following code:

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

outputs:

```
There are 0 things.
There is  1 thing.
There are 2 things.
```

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

3.3.1 Streams

BCPL uses streams as a convenient method of obtaining device independent input and output. All the information needed to process a stream is held in a vector called a stream control block (SCB) whose fields have already been summarized in Section 3.1.

The elements `pos` and `end` hold positions within the byte buffer, `file` holds a file pointer for file streams or `-1` for streams connected to the console. The element `id` indicates whether the stream is for input or output and work is private work space for the action procedures `rdfn`, `wrfn` which are called, repectively, when the byte buffer becomes empty on reading or full on output. The procedure `endfn` is called to close the stream.

Input is read from the currently selected input stream whose SCB is held in the global variable `cis`. For an input stream, `pos` holds the position of the next character to be read, and `end` points to just past the last available character in the buffer. Characters are read using `rdch` whose definition is given in figure 3.7. If a character is available in the buffer it is returned after incrementing `pos`. Exceptionally, the character carriage return (CR) is ignored since on some systems, such as Windows, lines are terminated with carriage return and linefeed while on others, such as Linux, only linefeed is used. If the buffer is exhausted, `replenish` is called to refill it, returning `TRUE` if one or more character are transferred. If replenish fails it returns `FALSE` with the reason why in `result2`. Possible reasons are: -1 indicating end of file, -2 indicating a timeout has occurred and -3 meaning input is in polling mode and no character is currently available. By setting the `timeoutact` field of the SCB to -1, a timeout is treated as end of file.

```
AND rdch() = VALOF
{ LET pos = cis!scb_pos // Position of next byte, if any
  UNLESS cis DO abort(186)
  IF pos<cis!scb_end DO { LET ch = cis!scb_buf%pos
                          cis!scb_pos := pos+1
                          IF ch='*c' LOOP // Ignore CR
                          RESULTIS ch
                        }

  // If replenish returns FALSE, it failed to read any characters
  // and the reason why is placed in result2 as follows
  //    result2 = -1    end of file
  //    result2 = -2    timeout
  //    result2 = -3    polling mode with no characters available.
  //    result2 = code  error code
  UNTIL replenish(cis) DO
  { IF result2=-2 DO
    { LET act = cis!scb_timeoutact    // Look at the timeout action
      IF act=-2 RESULTIS timeoutch    // Timed out
      IF act=-1 RESULTIS endstreamch  // End of file reached
      LOOP  // Try replenishing again
    }
    RESULTIS result2<0 -> result2, endstreamch
  }
} REPEAT
```

Figure 3.7: The definition of `rdch`

```
LET unrdch() = VALOF
{ LET pos = cis!scb_pos
  IF pos<=scb_bufstart RESULTIS FALSE // Cannot UNRDCH past origin.
  cis!scb_pos := pos-1
  RESULTIS TRUE
}
```

Figure 3.8: The definition of `unrdch`

Whenever possible, the buffer contains the previously read character. This is to allow for a clean and simple implementation of `unrdch` whose purpose is to step input back by one character position. Its definition if given in figure 3.8.

Output is sent to the currently selected output stream whose SCB is held in the global variable `cos`. The SCB field `pos` of an output stream holds the position in the buffer of the next character to be written, and `end` holds the position just past the end of the buffer. Characters are written using the function `wrch` whose definition is given in figure 3.9. The character `ch` is copied into the byte buffer and `pos` incremented. If the buffer is full, it is emptied by calling the element `wrfn`. If writing fails it return `FALSE`, causing `wrch` to abort.

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

  IF pos >= cos!scb_bufend DO
  { // The buffer is full
    UNLESS deplete(cos) RESULTIS FALSE
    UNLESS cos!scb_buf  RESULTIS TRUE // Must be writing to NIL:
    pos := cos!scb_pos
  }

  // Pack the character and advance pos.
  cos!scb_buf%pos := ch
  pos := pos+1
  cos!scb_pos := pos
  // Advance end of valid data pointer, if necessary
  IF cos!scb_end < pos DO cos!scb_end := pos
  cos!scb_write := TRUE // Set flag to indicate the buffer has changed.

  UNLESS cos!scb_type<0 & ch<'*s' RESULTIS TRUE // Normal return

  // The stream is interactive and ch is a control character.

  IF ch='*n' DO  wrch('*c')  // Fiddle for Cygwin

  // Call deplete at the end of each interactive line.
  IF ch='*n' | ch='*p' RESULTIS deplete(cos)
  RESULTIS TRUE
}
```

Figure 3.9: The definition of `wrch`

3.3.2 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 dependents. 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 82.

3.4 Coroutine examples

This section contains example code that uses the coroutine mechanism.

3.4.1 A square wave generator

The following function is the main function of a coroutine that generates square wave samples.

```
LET squarefn(args) = VALOF
{ LET freq, amplitude, rate = args!0, args!1, args!2
  LET x = 0
  cowait(@freq) // Return a pointer -> [freq, amplitude, rate]

  { // freq is a scaled fixed point value with
    // three digits after the decimal point.
    LET q4 = rate*1000
    LET q2 = q4/2
    UNTIL x > q2 DO { cowait(+amplitude) // First half cycle
                      x := x + freq
                    }
    UNTIL x > q4 DO { cowait(-amplitude) // Second half cycle
                      x := x + freq
                    }
    x := x - q4
  } REPEAT
}
```

The following call creates a coroutine that initially generates a square wave with frequency 440Hz and amplitude 5000 at a rate of 44100 samples per second.

```
sqco    := initco(squarefn, 300, 440_000, 5_000, 44_100)
sqparmv := result2    // sqparmv -> [freq, amplitude, rate]
```

One second's worth of samples can now be obtained by:

```
FOR i = 1 TO 44100 DO
{ LET sample = callco(sqco)
  ...
}
```

At any moment, the frequency and amplitude can be changed by assignments such as:

```
sqparmv!0 := newfrequency
sqparmv!1 := newamplitude
```

Other examples of the use of `initco` can be found below.

3.4.2 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 3.10 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.

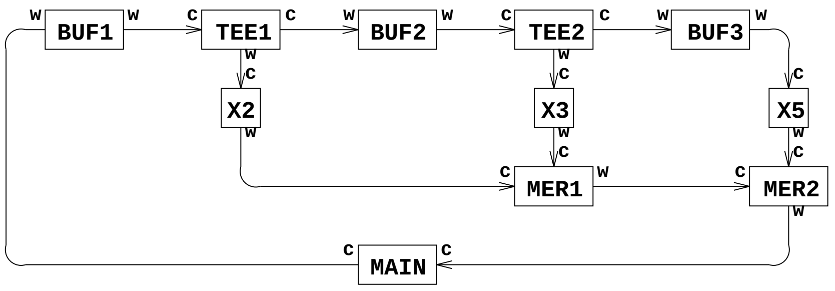

Figure 3.10: Coroutine data flow

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
}
```

Chapter 4

The Command Language

The Command Language Interpreter (CLI) is a simple interactive interface between the user and the system. It loads and executes previously compiled programs that are held either in the current directory or a directory specified by the environment variable `BCPLPATH`. The source of the standard commands can be found in the `com` directory. These commands are described below in Section 4.3. The command language is a combination of the features provided by the CLI and the collection of commands that can be invoked. Under Cintpos, exactly the same CLI code provides command language interpreters in several contexts such as those created by the commands: `run`, `newcli`, `tcpcli` and `mbxcli`. Details of the implementation of the CLI are given at the end of this chapter from page 89.

Commands can set a return code in the global `returncode` with zero meaning successful termination and other values indicating the severity of the fault. Commands that set a non zero return code are expected to leave a reason code in `result2`. The CLI copies the return code and reason code of the previous command into the CLI variables `cli_returncode` and `cli_result2`, respectively. These can be inspected by commands such as `if` and `why` and also used by the CLI to terminate a command-command if the failure was severe enough. For details, see the command `failat` on page 80 below.

4.1 Bootstrapping single threaded BCPL

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 4.1 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(0,0)`.

The global vector and stack shown in figure 4.1 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.

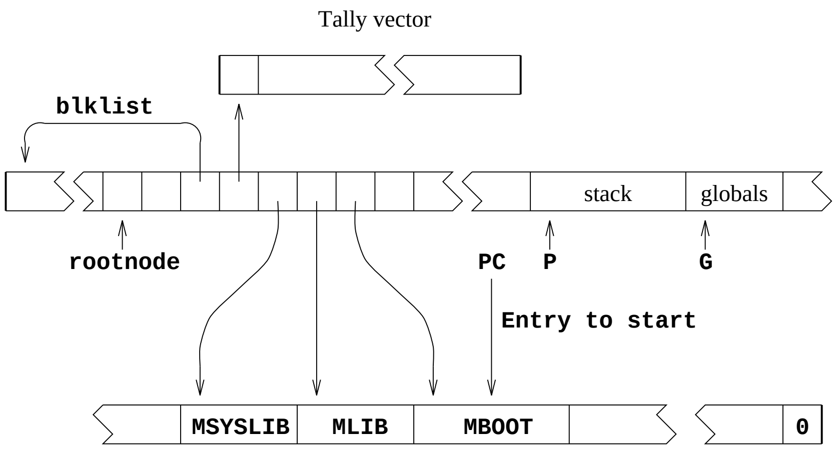

Figure 4.1: The initial state

Control is passed to the `CLI` by means of the call `sys(1,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(1,regs)` is embedded in the loop shown below that occurs at the end of the body of `start`.

```
{ LET res = sys(1, regs)  // Call the interpreter
  IF res=0 DO sys(0, 0)
  debug res               // Enter the debugger
} REPEAT
```

At the moment `sys(1,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 be 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(24, rootnode!rtn_syslib)
sys(24, 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 in 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 the section 6.

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 the 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()`.

4.2 Bootstrapping Cintpos

Bootstrapping Cintpos is somewhat more complicated that bootstrapping the single threaded version of BCPL since there are more resident modules of code, and the Cintpos system structures and resident tasks must be set up. Bootstrapping starts when the `cintpos` program is entered. It first decodes the command arguments, possibly changing the Cintcode memory or tally vector sizes. It then allocates these vectors, initialising every word of the Cintcode memory with the value `#xDEADCODE`. It also allocates a vector to hold counts of how many blocks of each requested size have been allocated `getvec` but not yet freed. It then allocates and initialises the stack and global vector to be used by `BOOT`. The rootnode is then initialised, including the setting of the fields: `rtn_boot` (holding the module `BOOT`), `rtn_klib` (holding the module `KLIB`), `rtn_blib` (holding the modules `BLIB, SYSLIB and DLIB`) and `rtn_sys` (holding the entry point to the function `sys`).

The initial values of the Cintcode registers are now placed in the register set `bootregs` and then Cintcode interpreter is entered to start execution from this initial state. If the interpreter returns a non zero result, a message containing this value is written to the standard output stream, and, if the `rtn_dumpflag` field of the root node is `TRUE`, the entire Cintcode memory is dumped to the file `DUMP.mem` in compacted form suitable for inspection by commands such as `dumpsys` or `dumpdebug`.

4.2.1 The Cintpos BOOT module

The function `start` in `BOOT` is the very first BCPL compiled code to be entered when Cintpos starts. On entry, the Cintcode registers `A`, `B` and `C` are zero, `P` and `G` point to BOOT's stack and global vector, and `ST` is set to 2, indicating that we are in `BOOT` and that interrupts are disabled. The global vector has already been initialised to hold all the entry points in `BOOT`, `KLIB`, `BLIB`, `SYSLIB` and `DLIB`, but the stack currently is filled entirely with the value `stackword=#xABCD1234` except for its zeroth word which was set by `cintpos` to hold the stacksize. To improve the behaviour of the standalone debugger, this stack is turned into a root coroutine stack of the specified size, initialising the globals `currco` and `colist` appropriately.

All input and output within BOOT and the standalone debugger is done using the standalone version of `rdch` and `wrch`, so these globals are updated appropriately. BOOT next intialises the variables used by the standalone debugger. These include the vectors `bpt_addr`, `bpt_instr` and `bpt_dbgvars` which respectively hold break point address, breakpoint instructions that have been overwritten by the `BRK` instruction, and the vector of the 10 standalone debugger variables `V0` to `V9`. These three vectors are placed in the rootnode to make them accessible both to the DEBUG task and to `dumpdebug` when it is inspecting a system dump.

BOOT now creates and initialises a global vector and a stack to be used during the further initialisation of the Cintpos system. The all elements of the global vector are given values of the form `globword`(=`#x8F8F0000`)+n, except for the globals `globsize` , `sys`, `rootnode`, `currco` and `colist`, the last two being set to zero. Every element of the stack is set to `stackword` (=`#xABCD1234`). The register set `klibregs` is initialised, giving zero to `A`, `B` and `C`, the stack and global vector pointers to `P` and `G`, the value one to `ST` to indicate execution is in KLIB and interrupts are disabled, and the entry point `startklib` in `PC`. This register set is then handed to a recursive call of the interpreter. This inner call is the one than performs the rest of the initialisation and enters the normal execution of the system. In due course the interpreter will return with a completion code which controls what BOOT should do next.

A completion code of zero signifies successfully completion and BOOT causes the termination of `cintpos`. A return code of -1 is special, causing BOOT to re-enter the interpreter immediately. Its purpose is to allow a running program to change which interpreter is used. There are typically two interpreters: a slow one in which all debugging aids are turned on, and a fast one in which most aids are turned off. The call `sys(Sys_interpret, regs)` selects the fast interpreter if the `count` register in `regs` is -1, otherwise it selects the slow interpreter. The return code -2 allows a running program to invoke the `dummpmem` mechanism to write the file DUMP.mem representing the current state of the Cintcode memory. Any other return code caused BOOT to invoke the standalone debugger, which many in due course return allowing the interpreter to be re-entered.

BOOT cunningly places a private version of the `sys` function in its global vector so that, even if a breakpoint is set in the public version of `sys`, BOOT and in particular the standalone debugger can continue to work as normal. When BOOT invokes the interpreter for the first time execution begins at the start of `klibstart` which is described in the next section.

4.2.2 `klibstart`

Needs to be written.

4.3 Commands

This section describes the Command Language Interpreter commands whose source code can be found in either `cintcode/com` or `cintpos/com`. The `rdargs` argument format string for each command is given.

abort `NUMBER` CIN:y, POS:y, NAT:y

The command: `abort` *n* calls the `BLIB` function `abort` with argument *n*. If *n* is zero, this causes a successful return from the BCPL system. If *n* is non zero, the interactive debugger is entered with fault code *n*. The default value for *n* is `99`. The interactive debugger is described in section 6.

adjclock *offset* CIN:y, POS:y, NAT:y

The syntax of *offset* is [-][*h*][:m], that is: an optional minus sign, followed by an optional number of hours, followed optionally by :m to specify a number of minutes. The offset is converted into a signed integer representing the number of minutes to be added to the time of day as supplied by the system. If `adjclock` is given no argument, it just outputs the current offset.

alarm `AT/A,MESSAGE` CIN:n, POS:y, NAT:n

This command is only available under Cintpos. Its first parameter has the format: `[+][[hours:]minutes:]seconds`. If `+` is present the time is relative to now. The command suspends itself until the specified time, then outputs the time followed by the message. Typical usage is as follows:

```
run alarm +3:30 "You time is up!"
```

After three and a half minute a message such as the following will appear.

```
*** Alarm: time is 15:13:14 - You time is up!
```

bcpl `FROM/A,TO/K,VER/K,SIZE/K/N,TREE/S,NONAMES/S,`
`D1/S,D2/S,OENDER/S,EQCASES/S,BIN/S,XREF/S,GDEFS/S,HDRS/K,`
`GB2312/S,UTF8/S,SAVESIZE/K/N` CIN:y, POS:y, NAT:y

This invokes the BCPL compiler. The `FROM` argument specified the name of the file to be compiled. If the `TO` argument is given, the compiler generates code to the specified file. Without the `TO` argument the compiler will output the OCODE intermediate form to the file `ocode` as a compiler debugging aid. This file can be converted to a more readable form usinf the `procode` command, described below. 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 100,000 words. The `NONAMES` switch causes the compiler not include section and function names in the compiled code. The switches `D1` and `D2` control compiler debugging output. `D1` causes a readable form of the compiled Cintcode to be output. `D2` causes a detailed 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. `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 the ASCII encoded hexadecimal normally used. The XREF option causes a line to be output by the compiler for each non local identifier occurring in the program. A typical such line is as follows:

```
        all G:201 LG queens.b[9] all&~(ld|row|rd)
```

It shows that the variable all was declared as global variable 201 and its was loaded in the compilation of statements on line 9 of the program queens.b and the context of its use was: all&~(ld|row|rd). These lines can be filtered and sorted to form a cross reference listing of a program. See, for instance, the file BCPL/cintcode/xrefdata or Cintpos/cintpos/xrefdata.

The GDEFS switch is a debugging aid to output the global numbers of any global function defined in the program. For example:

```
        bcpl gdefs com/bench100.b to junk
```

generates the following output:

```
BCPL (3 July 2007)
G  1 = start
G259 = trace
G260 = schedule
G261 = qpkt
G262 = wait
G263 = holdself
G264 = release
G270 = idlefn
G271 = workfn
G272 = handlerfn
G273 = devfn
Code size = 1436 bytes
```

The UTF8 and GB2312 options specify the default encoding for extended characters in string and character constants. This default can be overridden in individual constants using the *#u and *#g escape sequences, as described on page 13.

The SAVESIZE option allows the user to specify the number of words in the argument stack used to hold function return information. The default value is three making room for the old P pointer, the return address and the entry point of the current function. When compiling into native code using the Sial mechanism, the save space size may be different, since, for instance, some or all of this information may be stored in the hardware (SP) stack.

bcpl2sial FROM/A,TO/K,VER/K,SIZE/K/N,TREE/S,NONAMES/S,
D1/S,D2/S,OENDER/S,EQCASES/S,BIN/S,XREF/S,GDEFS/S,HDRS/K,
GB2312/S,UTF8/S,SAVESIZE/K/N   CIN:y, POS:y, NAT:y

This command compiles a BCPL program into the internal assembly language Sial which is designed as a low level intermediate target code for BCPL and is described in

Section 9.1. The command `sial-sasm`, described below, can be used to convert Sial into a human readable form and various commands, such as `sial-386` and `sial-alpha` will convert Sial to assembly language for corresponding architectures. The `bcpl2sial` command takes the same arguments as the `BCPL` command.

bcplxref `FROM/A,TO/K,PAT/K` CIN:y, POS:y, NAT:y

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.

bench100 CIN:y, POS:y, NAT:y

bin-hex CIN:y, POS:y, NAT:y

bin-x8 CIN:y, POS:y, NAT:y

`bounce` CIN:n, POS:y, NAT:n

This command is part of the bounce demonstration that is only available under Cintpos. It is normally invoked by the command: `run bounce` which creates a new CLI task and then enters the `bounce` program whose main loop is:

```
        qpkt(taskwait()) REPEAT
```

which repeatedly suspends the task until a packet is received then immediately returns it to the sender. Packets are normally sent to the bounce task using the `send` command, described below.

break CIN:y, POS:y, NAT:y

c ***command-file arguments*** CIN:y, POS:y, NAT:y

The `c` command allows a file oc 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 49) 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` CIN:y, POS:y, NAT:y

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

changepri CIN:y, POS:y, NAT:y

checksum FROM/A,TO/K CIN:y, POS:y, NAT:y

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.

cmpltest CIN:y, POS:y, NAT:y

cobench CIN:y, POS:y, NAT:y

cobounce CIN:y, POS:y, NAT:y

compare CIN:y, POS:y, NAT:y

cosim CIN:y, POS:y, NAT:y

dat CIN:y, POS:y, NAT:y

date CIN:y, POS:y, NAT:y

delete ,,,,,,,,, CIN:y, POS:y, NAT:y

This command will delete up to ten given files.

detab `FROM/A,TO/K,SEP/K` CIN:y, POS:y, NAT:y

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.

dumpdebug CIN:y, POS:y, NAT:y

dumpmem CIN:y, POS:y, NAT:y

dumpsys CIN:y, POS:y, NAT:y

echo `TEXT,N/S` CIN:y, POS:y, NAT:y

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` CIN:y, POS:y, NAT:y

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.

endcli CIN:y, POS:y, NAT:y

enlarge CIN:y, POS:y, NAT:y

fail `CODE` CIN:y, POS:y, NAT:y

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

failat CIN:y, POS:y, NAT:y

getlogname CIN:y, POS:y, NAT:y

harness CIN:y, POS:y, NAT:y

help CIN:y, POS:y, NAT:y

hex-bin CIN:y, POS:y, NAT:y

hexdump CIN:y, POS:y, NAT:y

hold CIN:y, POS:y, NAT:y

idvec CIN:y, POS:y, NAT:y

if CIN:y, POS:y, NAT:y

input `TO/A,TERM/K` CIN:y, POS:y, NAT:y

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|` CIN:y, POS:y, NAT:y

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 56.

join `,,,,,,,,,,,,,,,AS/A/K,CHARS/S` CIN:y, POS:y, NAT:y

This command will concatenat 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.

lab ***label*** CIN:y, POS:y, NAT:y

This command read the rest of the command line but otherwise has no effect. It is used as the destination of `skip` commands.

library CIN:y, POS:y, NAT:y

logout CIN:y, POS:y, NAT:y

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

makeinit CIN:y, POS:y, NAT:y

map `BLOCKS/S,NAMES/S,CODE/S,MAPSTORE/S,TO/K,PIC/S` CIN:y, POS:y, NAT:y

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.

mbxcli `MBXNAME` CIN:n, POS:y, NAT:n

n This command creates a new CLI task taking input from the specified mailbox, typically `MBX:`*name*.If no argument is specified the default mailbox `MBX:commands` is used. Any task can write command lines to a mailbox in a first come first served manner and any CLI created by mbxcli can read and perform them, similarly in a first come first served manner. If a mailbox CLI performs the `endcli` command it commits suicide.

mbxrx `-n,-d,-b/K` CIN:n, POS:y, NAT:n

n This command is designed to test the mailbox system under Cintpos. It will read a number of mailbox lines specified by the `-n` argument. Each line read is written to the standard output stream. It then delays for a number of ticks specified by the `-d` argument before reading the next mailbox line. The mailbox is specified by the `-b` argument with the default being `MBX:junk`.

mbxtx `-n,-d,-b/K` CIN:n, POS:y, NAT:n

n This command is designed to test the mailbox system under Cintpos. It will write a number of lines specified by the `-n` argument to a mailbox. Each line sent is written to the standard output stream. It then delays for a number of ticks specified by the `-d` argument before sending the next mailbox line. The mailbox is specified by the `-b` argument with the default being `MBX:junk`.

mcpl CIN:y, POS:y, NAT:y

mcpl2mial CIN:y, POS:y, NAT:y

mial-386.b CIN:y, POS:y, NAT:y

mial-masm CIN:y, POS:y, NAT:y

mkdata CIN:y, POS:y, NAT:y

mkjunk CIN:y, POS:y, NAT:y

newcli CIN:y, POS:y, NAT:y

nlconv `FILE,TOUNIX/S,TODOS/S,Q/S` CIN:y, POS:y, NAT:y

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

origbcpl CIN:y, POS:y, NAT:y

This is an old version of the BCPL compiler dated 13 August 2001.

playback CIN:y, POS:y, NAT:y

playfast CIN:y, POS:y, NAT:y

playtime CIN:y, POS:y, NAT:y

prefix `PREFIX,UNSET/S` CIN:y, POS:y, NAT:y

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(Sys_setprefix,`*prefix*`)` and `sys(33)` described on page 58. See also Section 3.3.2.

preload `,,,,,,,,,` CIN:y, POS:y, NAT:y

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.

prmcode CIN:y, POS:y, NAT:y

This command converts an MCODE (intermediate code for MCPL) file specified by `FROM` to a more readable form. If `FROM` is missing it reads from the file `MCODE`. If the `TO` argument is missing it send the result to the screen. The file MCODE is a byproduct of the `mcpl` command, see `mcpl` above.

procode `FROM,TO/K` CIN:y, POS:y, NAT:y

This command converts an OCODE (intermediate code for BCPL) 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,NO/S` CIN:y, POS:y, NAT:y

If the `NO` switch is given prompts are disabled, otherwise they will be enabled. Under Cintpos, disabling prompts is useful, for instance, if a CLI task is taking input from a tcp/ip connection where the source of the commands is another program. The `PROMPT` argument is optional, but if present will be the new prompt format string. Prompts are generated by the CLI using a call of the following form.

`writef(`*prompt*`,` *cpumsecs*`,` *taskno*`,` *hours*`,` *mins*`,` *secs*`)`

where *prompt* is the prompt format string, *cpumsecs* is the time in milliseconds used by the previous command, *taskno* is the current task number under Cintpos and zero otherwise. The arguments *hours*, *mins*, and *secs* represent the current time of day. The default prompt format under Cintpos is: `"%+%n> "` and under the other systems is: `"%+%n> "`. An example of how it might be used is as follows.

```
0>
0> prompt "%+%+%z2:%z2:%z2 %-%-%-%-%-%n> "
15:11:52 0>
15:11:55 0> bench100

bench mark starting, Count=1000000

starting

finished
qpkt count = 2326410  holdcount = 930563
these results are correct
end of run
15:12:14 10690>
```

This shows that `bench100` finished execution 14 seconds after 3:12 pm after running for 10.690 seconds.

quit `RC/N` CIN:y, POS:y, NAT:y

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` CIN:y, POS:y, NAT:y

This commands 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 posible to include anotations 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` specified the approximate number of dots per inch used by the output device. The default is 300. `A`*n* specified 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 4.2. 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 editied by hand.

raster `COUNT,SCALE,TO/K,HELP/S` CIN:y, POS:y, NAT:y

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(Sys_setraster,...)` calls that are described on page 58. 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 58.

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 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 4.2.

record `TO,OFF/S` CIN:n, POS:y, NAT:n

This command allows Cintpos console sessions to be recorded.

rename `FROM/A,TO=AS/A/K` CIN:y, POS:y, NAT:y

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

repeat CIN:y, POS:y, NAT:y

run CIN:y, POS:y, NAT:y

`send TASK,COUNT` CIN:n, POS:y, NAT:n

This is part of the Cintpos bounce demonstration. It repeated sends a packet to the specified task the specified number of times. The default task number is 7 and the default count is 1000000. It can be used to measure the efficiency of inter-task communication.

setflags CIN:y, POS:y, NAT:y

setlogname CIN:y, POS:y, NAT:y

shellcom CIN:y, POS:y, NAT:y

sial-386 CIN:y, POS:y, NAT:y

sial-alpha CIN:y, POS:y, NAT:y

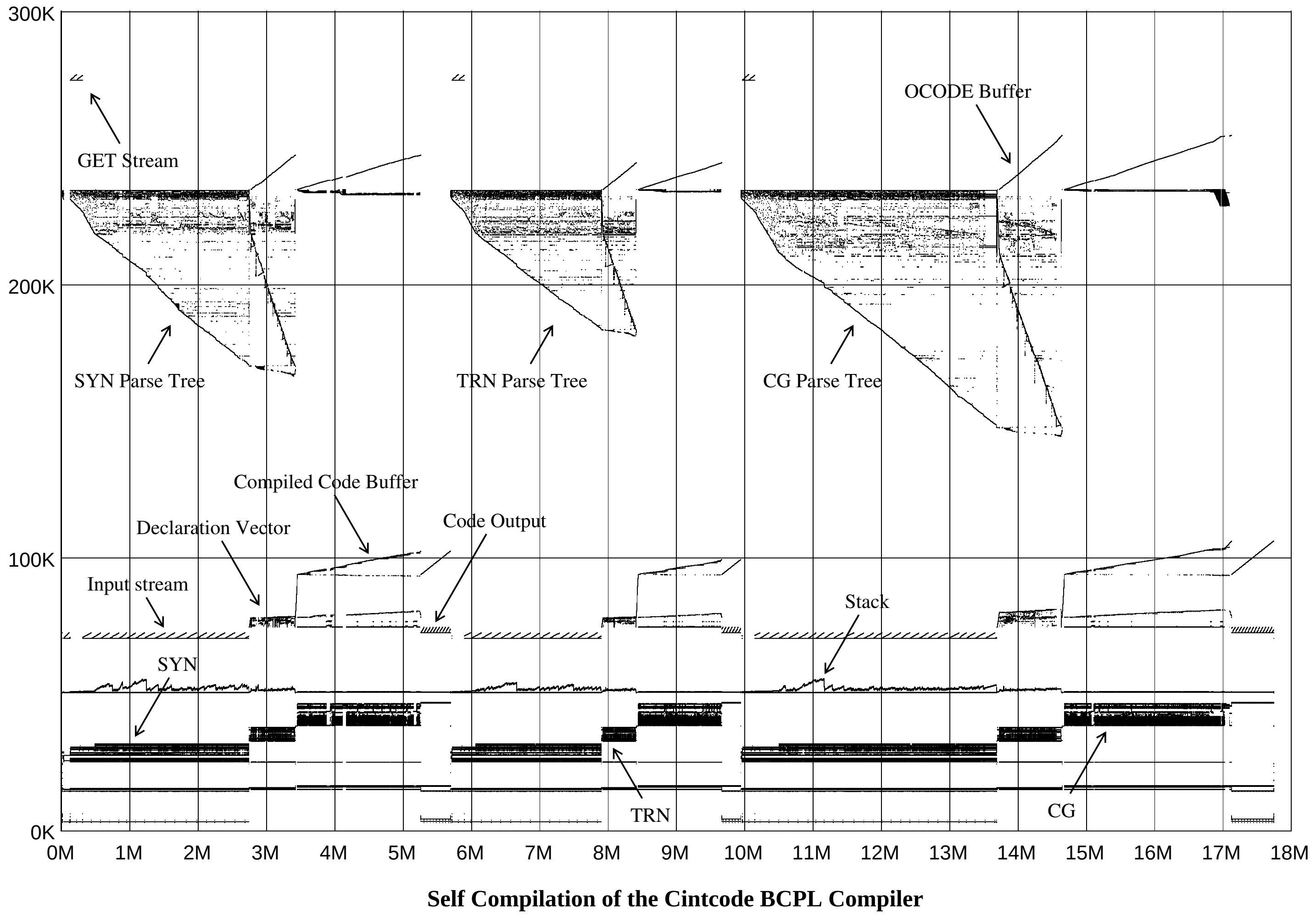

Figure 4.2: Self compilation memory-time graph

sial-sasm CIN:y, POS:y, NAT:y

sial-vax CIN:y, POS:y, NAT:y

skip `DESTINATION-LABEL` CIN:y, POS:y, NAT:y

The command `skip` *`label`* skips through the command stream until a line starting with `lab` *`label`* is encountered. It then skips until the end of that line before resuming normal command execution from there. The `skip` command is only allowed within command-commands.

stack `SIZE` CIN:y, POS:y, NAT:y

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` CIN:y, POS:y, NAT:y

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

status CIN:y, POS:y, NAT:y

strtodat CIN:y, POS:y, NAT:y

syncdemo CIN:y, POS:y, NAT:y

taskid CIN:y, POS:y, NAT:y

tbcpl CIN:y, POS:y, NAT:y

tcpaddr CIN:y, POS:y, NAT:y

tcpbench CIN:y, POS:y, NAT:y

tcpcli CIN:y, POS:y, NAT:y

tcpdump CIN:y, POS:y, NAT:y

tcprx CIN:y, POS:y, NAT:y

tcptest CIN:y, POS:y, NAT:y

tcptx CIN:y, POS:y, NAT:y

testtime CIN:y, POS:y, NAT:y

time CIN:y, POS:y, NAT:y

timeouts CIN:y, POS:y, NAT:y

tracebuf ***none*** CIN:y, POS:y, NAT:y

This command outputs the values pushed into the trace buffer by the `trpush` function. These are output in hex, eight values per line with the most recently pushed value first. While `tracebuf` is running insertion of new values into the buffer is disables. Trace buffer values often represent events and can assist the debugging of subtle (often real time) programming errors.

The rootnode has a field `trbuf` that points to the trace buffer provided the rootnode field `trword` is set to `#xBFBFBFBF`. Settinf `trword` to any other value disable the `trpush` function. If `buf` is the trace buffer, `buf!0` is its upper bound (typically 1001) and `buf!1` is the position, between 2 and `buf!0`, of where the next value will be pushed. As a safety check this location will hold the value `#xBFBFBFBF`. Values can be pushed into the trace buffer from anywhere in the system, and in Cintpos this can be within the kernel and even within the interrupt service routine or device drivers.

type FROM/A,TO,N/S CIN:y, POS:y, NAT:y

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

typeflush CIN:y, POS:y, NAT:y

typehex FROM/A,TO/K CIN:y, POS:y, NAT:y

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

unhold CIN:y, POS:y, NAT:y

unpreload CIN:y, POS:y, NAT:y

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

vecstats CIN:y, POS:y, NAT:y

wait CIN:y, POS:y, NAT:y

why CIN:y, POS:y, NAT:y

x8-bin CIN:y, POS:y, NAT:y

4.4 cli.b and cli_init.b

The Command Language Interpreter is a simple program implemented in BCPL whose source code can be found in the files **sysb/cli.b** and **sysb/cli_init.b**. This section mainly describes the Cintpos version. The CLI is the first program the interacts with after starting the system. Under Cintpos it runs as task one (named Root_Cli). It uses variables in the global vector to hold its state during command execution. These variables have reserved global numbers typically in the range 133 to 149. They are declared in **g/clihdr.b**. Since running commands use the same global vector they can access (and even modify) these variables – a feature that is both dangerous and useful. Commands such as **run** and **c** rely on this feature. The CLI global variables are as follows.

cli_init CIN:y, POS:y, NAT:y

This holds the function used to initialise the CLI, and depends on which context the CLI is to run in. It is called when the CLI is first entered using the following code.

```
{ LET f =   cli_init(parm.pkt)
  IF f DO f(result2) // Must get result2 after calling cli_init
}
```

As can be seen `cli_init` must either return zero or a function that can be applied to `result2`. The function is typically `deletetask` or `unloadseg` with `result2` being suitably set.

cli_returncode, cli_result2 CIN:y, POS:y, NAT:y

These hold the return code and the value of `result2` of the most recently executed command.

cli_faillevel CIN:y, POS:y, NAT:y

cli_data CIN:y, POS:y, NAT:y

This holds CLI data dependant on the context in which the CLI is running.

cli_commanddir CIN:y, POS:y, NAT:y

cli_prompt CIN:y, POS:y, NAT:y

This variable holds the current prompt string. It should be a `writef` format string since it used in the Cintpos CLI as follows:

```
writef(cli_prompt, taskid, mins/60, mins REM 60, secs)
```

where `hours`, `mins` and `secs` correspond to the current time of day. On single threaded BCPL systems the corresponding call is:

```
{ writef(cli_prompt, msecs)
```

where `msecs` is the real time of execution of the latest command.

cli_currentinput, cli_currentoutput, cli_standardinput, cli_standardoutput CIN:y, POS:y, NAT:y

The standard input and output streams are those that were setup when the CLI was started. Sometimes a CLI will change its currently selected streams. For instance, while executing a command-command the currently selected input will be from a tempory file of commands. On reaching the end of file input will revert to the standard input.

cli_commandfile CIN:y, POS:y, NAT:y

This is either zero or holds the name of temporary command file used in command-commands.

cli_status CIN:y, POS:y, NAT:y

This holds a collection of bits specifying the context in which the CLI is running. The mnemonics for these bits and their meanings are as follows.

clibit_noprompt	Do not output prompts even when not in a command-command.
clibit_eofdel	Delete this task when EOF is received under Cintpos.
clibit_comcom	This CLI is currently in a command-command executing commands from a temporary file.
clibit_maincli	This CLI is the task 1 CLI under Cintpos or the main CLI under other systems.
clibit_newcli	This CLI was created by the `newcli` command under Cintpos.
clibit_runcli	This CLI was created by the `run` command under Cintpos.
clibit_mbxcli	This CLI was created by the `mbxcli` command under Cintpos.
clibit_tcpcli	This CLI was created by the `tcpcli` command under Cintpos.
clibit_endcli	The `endcli` command has been executed on this CLI under Cintpos.

cli_background CIN:y, POS:y, NAT:y

This is an obsolete variable that mainly controlled the generation of prompts. It is to be superceded by the `noprompt` bit in `cli_status`.

cli_defaultstack CIN:y, POS:y, NAT:y

This holds the size of the coroutine stack that the CLI creates every time it runs a command. Its value can be changed by the `stack` command.

cli_commandname CIN:y, POS:y, NAT:y

This holds the name of the current command

cli_module CIN:y, POS:y, NAT:y

This is either zero or the module of loaded code corresponding to the currently execution command. It is used by the CLI to unload the command when it has finished execution.

Chapter 5

Console input and output

When `cintsys` or `cintpos` is started a stream is opened to recieve input from standard input which is normally the keyboard and a second stream is opened to allow output to standard output which is normally the screen. This combination of keyboard and screen is called the console. The treatment of console streams depends on whether `cintsys` or `cintpos` is being used.

5.1 Cintsys console streams

The stream control block for the keyboard stream is obtained by calling `findinput("**")`. The stream is created the first time it is called. Subsequent calls yield exactly the same stream control block. This stream has a buffer large enough to hold 4096 characters. Characters are read from the keyboard using `sardch` which reads and echoes each character to the screen. Exceptionally, ctrl-c (code 3) causes a SIGINT interrupt, RUBOUT (code 127) is translated to backspace (code 8), ctrl-j, ctrl-m and the ENTER (or RETURN) key all yield code 10 (the BCPL newline character) but they all echo carriage return and linefeed to the screen.

Simple line editing of keyboard input is performed as follows. As characters are typed they are normally transferred into the buffer, but if a backspace is received, the latest character, is any, in the buffer is removed and its echoed symbol removed from the screen. The contents of the buffer is not made available to the user until either a newline character is received or the buffer becomes full.

A user can receive keyboard characters as soon as they are typed using calls of `sardch`.

The stream control block for the screen stream is obtained by calling `findoutput("**")`. The stream is created the first time it is called. Subsequent calls yield exactly the same stream control block. This stream has a buffer large enough to hold 4096 characters. Call of `wrch` places characters in this buffer, and when a newline or newpage character is written, or the buffer becomes full, or a call of `deplete` is made, the contents of the buffer is transmitted to the screen by calls of `sawrch`.

The program `BCPL/bcplprogs/test/inputtst.b` can be used to demonstate some of the features of console input.

5.2 Cintpos console streams

Under Cintpos interaction with the console is somewhat more complicated since Cintpos can have several tasks all wishing to communicate with the keyboard and screen. This interaction is controlled by a task called the Console Handler (typically task 3). Tasks wishing to read from the keyboard or write to the screen must send request packets to this task where they will be properly scheduled.

The call `findinput("**")` yields a new stream control block connected to the keyboard. Initially it has no buffer. When the client task tries to read from this stream, a read request packet is sent to the console handler which will in due course return with a buffer of one or more characters or an indication that the keyboard stream is exhausted. Keyboard read requests can be sent simultaneously from several tasks and, indeed, a single task can send multiple requests. These are queued in the console handler and processed on a first come first served basis.

The console handler obtains characters from the keyboard by sending ttyin request packets to the keyboard device (typically device -2). This device returns keyboard characters to the console handler as they are typed without echoing them to the screen. It does no translation except that the characters ctrl-j, ctrl-m and the ENTER key all yield code 10 (the BCPL newline character). Keyboard characters received by the console handler are normally packed into an input buffer to form input lines. Simple line editing is performed using the backspace key (code 8 or 127) which causes the most recent character in the line buffer to be removed. When a newline is received or the buffer is full or the escape sequence `@e` is typed, the line buffer is ready to send to the currently selected task. Initially this is task 1 (the main CLI task) but can be changed by the user using the escape mechanism described below. While a user is typing an input line, it will appear on the screen and other screen output requests will be held until the input line is complete. At any time if there is a completed input line for a task that has sent a read request packet, it will be returned to the client with the line buffer and number of characters in its two result fields. Lines that have not yet been requested are queued as are read requests that are not yet satisfied. Note that a simple way to temporally stop output to the screen is to type a character such as SPACE, and then delete it later using backspace.

Cintpos console input has the following escape mechanism. All escape sequence start with an at sign (`@`) and their effects are shown in the following table.

Sequence	Purpose
`@A`	Set flag 1 in the currently selected task
`@B`	Set flag 2 in the currently selected task
`@C`	Set flag 3 in the currently selected task
`@D`	Set flag 4 in the currently selected task
`@E`	Send the current incomplete line to the currently selected task
`@F`	Throw away the current incomplete line and all outstanding completed lines
`@H`	Hold the currently selected task
`@L`	Throw away the current incomplete line
`@S`*dd*	Set the currently selected task to task *dd* and allow output from any task
`@T`*dd*	Set the currently selected task to task *dd* and only allow output from task *dd*
`@U`	Unhold the currently selected task
`@X`*hh*	Input the character with hex code *hh*
`@Y`	Toggle message tagging. When tagging is enabled every line of output identifies the originating task
`@Z`	Toggle echo mode. When echoing is off subsequent characters are not echoed to the screen. This is useful for typing passwords.
`@`*ddd*	Input the character with octal code *ddd*
`@@`	Input `@`

5.2.1 Devices

The input and output device intentifiers may be inspected and changed by the following call:

```
old_in_devid := sendpkt(notinuse, console_task, Action_devices,
                        ?, ?,
                        new_in_devid,
                        new_out_devid)
old_out_devid := result2
```

The device identifiers are only changed if the new identifiers are non zero. This call is used, for instance, by the `record` command to change replace the screen output device with a task that forwards each character to the screen while recording timing information. For details, see the programs `com/record.b` and `com/recordtask.b`

5.2.2 Exclusive input

The console handler can be set to exclusive input mode by the call:

```
sendpkt(notinuse, console_task, Action_exclusiveinput,
        ?, ?,
        TRUE)
```

While in `exclusiveinput` mode normal input line editing by the console handler is suspended and client tasks have direct access to the keyboard input device on a first come first served basis by the call:

```
ch := sendpkt(notinuse, console_task, Action_exclusiverdch,
              ?, ?)
```

Sending an `exclusiveinput` request with argument `FALSE` returns the console handler to its normal line editing mode and causes all outstanding `exclusiverdch` requests to return end-of-file characters (-1) to their client tasks.

5.2.3 Direct access to the screen and keyboard

Although it is not recommended, client task can send read (`Action_ttyin`) and write (`Action_ttyout`) requests to keyboard and screen devices. These will be serviced in a first come first served basis and since the console handler is making such requests you can expect strange results.

Finally the functions `sardch` and `sawrch` provide direct access to the keyboard and screen but are mainly only used for system debugging particularly when the console handler is not running. Note that `sawrch` is the character output function used by `sawritef` whose output may be merged with output from the console handler.

The following test programs can be used to demonstate some of the console handlers features.

```
Cintcode/posprogs/test/inputtst.b
Cintcode/posprogs/test/sardchtst.b
Cintcode/posprogs/test/devrdchtst.b
Cintcode/posprogs/test/xintst.b
```

Chapter 6

The Debugger

When the Cintcode system starts up, control first passes to `BOOT` which initialises the system and creates a running environment for the command language interpreter (`CLI`). This is run by a recursive invocation of the interpreter and so when faults occur control returns to `BOOT` which then enters an interactive debugger. This allows the user to inspect the state of the registers and memory, and perform other debugging operations on the faulted program. The debugger can also be entered using the `abort` command, as follows:

```
560> abort

!! ABORT 99: User requested
*
```

The asterisk (`*`) is the debugger's prompt character. A brief description of the available debug commands can be display using the query (`?`) command.

```
* ?
?     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       Store in variable
=          Print current value
Tn         Print n consecutive locations
$c         Set print style C, D, F, B, O, S, U or X
LL LH      Set Low and High store limits
I          Print current instruction
N          Print next instruction
Q          Quit
B 0Bn eBn  List, Unset or Set breakpoints
C          Continue execution
X          Equivalent to G4B9C
Z          Equivalent to P1B9C
\          Execute one instruction
,          Move down one stack frame
.          Move to current coroutine
;          Move to parent coroutine
[          Move to first coroutine
]          Move to next coroutine
*
```

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

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

Four areas of memory, namely: the global vector, the current stack frame, the Cintcode register, and 10 scratch variables are easily accessed using the letters `G`, `P`, `R`, `V`, respectively.

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

V  0:          0          10          11           0           0
*
* v1*50+v2=          511            A calculation using variables
* g0=       1000                    Display global zero (globsize)
* g=       3615                     Display the address of global zero
* ! =       1000                    Indirect and display
* gt10                              Display the first 10 globals

G  0:       1000     start       stop        sys         clihook
G  5:    GLOB  5     changec       6081        6081          52
*
```

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 respectively display values as 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:

```
* g4=    clihook                       Get the entry to clihook
* n    3340:    K4G  1                 Call global 1, incremeting P by 4
* n    3342:    RTN                    Return from the function
*
```

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

```
* g4=    clihook                       Get the entry to clihook
* b9                                   Set break point 9
* c                                    Resume execution
20>
```

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

```
10> echo ABC                           Invoke the echo command

!! BPT 9:      clihook                 Break point hit
   A=          0 B=         0     3340:    K4G  1
*
```

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:

```
* \A=          0 B=         0    24228:    LLP  4
* \A=       6097 B=         0    24230:    SP3
* \A=       6097 B=         0    24231:     SP  89
* \A=       6097 B=         0    24233:      L  80
* \A=         80 B=      6097    24235:     SP  90
* \A=         80 B=      6097    24237:    LLL  24272
* \A=       6068 B=        80    24239:     LG  78
* \A=    rdargs  B=      6068    24241:      K  85
* \A=       6068 B=      6068     5480:    LP4
*
```

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:

```
* p1b8
* c

!! BPT 8:        24243
   A=    createc B=         1    24243:   JNE0  24254
*
```

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

```
* g3b1             Set breakpoint 1
* b                Display the currently set of breakpoints
1:      sys
8:        24243
9:      clihook
* 0b8 0b9          Unset breakpoints 8 and 9
* b                Display the remaining breakpoint
1:      sys
*
```

The next three calls of `sys` will be to write the characters `ABC`. The following example steps through these and displays the state of the runtime stack just before the third call, before leaving the debugger.

```
* c

!! BPT 1:      sys
   A=         11 B=         65    21188:    SYS
* c
A
!! BPT 1:      sys
   A=         11 B=         66    21188:    SYS
* c
B
!! BPT 1:      sys
   A=         11 B=         67    21188:    SYS
* .   42844:  Active coroutine     clihook   Size 20000  Hwm   127
      43284:    sys               11          67         312       43228
* ,   43268:    cnslwrf        37772
* ,   43248:    wrch              67          32
* ,   43228:    writes         42915          67
* ,   42888:    start          42904       42912           0     4407873
* ,   42872:    clihook            0
* , Base of stack
* 0b1c                    Clear breakpoint 1 and resume
C
210>
```

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

Command	Effect
.	Select current coroutine
,	Display next stack frame
;	Select parent coroutine
[	Select first coroutine
]	Select next coroutine

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

Chapter 7

The design of OCODE

BCPL was designed to be a portable language with a compiler that is easily transferred from machine to machine. To help to achieve this, the compiler is structured as shown in figure 7.1 so that the codegenerator (CG), which is inherently machine dependent, is separated from the rest of the compiler. The front end of the compiler performs syntax analysis producing a parse tree (Tree) which is then translated by the translation phase (TRN) to produce an intermediate form (OCODE) suitable for code generation.

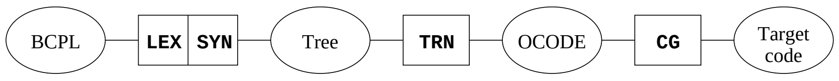

Figure 7.1: The structure of the compiler

7.1 Representation of OCODE

Since OCODE is output by TRN to be read in by CG, there is little need for it to be readable by humans and so is encoded as a sequence of integers which, in the current Cintcode implementation the OCODE is buffered in memory, however the compiler can be made to output a text version to file.

The numerical representation of OCODE can be transformed to the more readable mnemonic form using the `procode` commands, described on page 83. As an example, if the file `test.b` is the following:

```
GET "libhdr"

LET start() BE { LET a, b, c = 1, 0, -1
                 writef("Answer is %n*n", a+b+c)
               }
```

then the command: `bcpl test.b ocode test.ocd` would write the following file:

```
85 2 94 1 5 115 116 97 114 116 95 3 42 1 42 0 42 -1 92 91 9 43
13 65 110 115 119 101 114 32 105 115 32 37 110 10 40 4 40 3 14
40 5 14 41 74 51 6 97 91 3 103 91 3 90 2 92 76 1 1 1
```

These numbers encode the OCODE statements in a natural way as can be verified by comparing them with the following more readable form of the same statements, generated by the command: `procode test.b`.

```
JUMP L2
ENTRY L1 5  's' 't' 'a' 'r' 't'
SAVE 3 LN 1 LN 0 LN -1 STORE STACK 9
LSTR 13  'A' 'n' 's' 'w' 'e' 'r' ' ' 'i' 's' ' ' '%' 'n' 10
LP 4 LP 3 PLUS LP 5 PLUS LG 74 RTAP 6 RTRN STACK 3
ENDPROC STACK 3 LAB L2 STORE GLOBAL 1 1 L1
```

7.2 The OCODE Abstract Machine

OCODE was specifically designed for BCPL and is a compromise between the desire for simplicity and the conflicting demands of efficiency and machine independence. OCODE is an assembly language for an abstract stack based machine that has a global vector and an area of memory for program and static data as shown in figure 7.2.

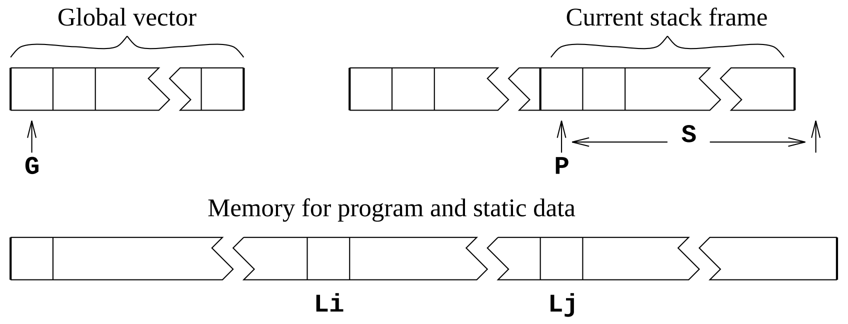

Figure 7.2: The BCPL abstract machine

The global vector is pointed to by the `G` pointer and the current stack frame is pointed to by the `P` pointer. `S` is the size of the current stack frame, and so `P!S` is the first free element of the stack. The value of `S` is always known during compilation and so is not held in a register of the OCODE abstract machine machine. Any assignments to `S` in the description of OCODE statements should be regarded as a specification of `S` for the subsequent statement.

Static variables, tables and string constants are allocated space in the program area and are referenced using labels such as L36 and L92. All global, local and static variables are of the same size and, on most modern implementations, they hold 32 bit values.

OCODE is normally encoded as a sequence of integers, but for human consumption a more readable form is available. The command `procode` translates the numeric OCODE into this mnemonic form. An OCODE statement consists of a function or directive code possibly followed by operands that are either optionally signed integers, quoted characters or labels of the form Ln where n is a label number. The following are examples of mnemonic OCODE statements:

```
LSTR 5 'H' 'e' 'l' 'l' 'o'
LP 3
GETBYTE
SL L36
```

There are OCODE statements for loading and storing values, for applying expression operators, for procedure handling and flow of control. There are also directives for the allocation of storage and to allow information to be passed to the codegenerator.

7.3 Loading and Storing values

A variables may be local, global or static, and may be accessed in three ways depending on its context, and so there are 9 statements for accessing variables as shown in the following table.

Statement	Meaning
LP n	P!S := P!n; S := S+1
LG n	P!S := G!n; S := S+1
LL Ln	P!S := Ln; S := S+1
LLP n	P!S := @P!n; S := S+1
LLG n	P!S := @G!n; S := S+1
LLL Ln	P!S := @Ln; S := S+1
SP n	P!n := P!S; S := S-1
SG n	G!n := P!S; S := S-1
SL Ln	Ln := P!S; S := S-1

The following tables shows the six statements for loading constants.

Statement	Meaning
LF Ln	P!S := entry point Ln; S := S+1
LN n	P!S := n; S := S+1
TRUE	P!S := TRUE; S := S+1
FALSE	P!S := FALSE; S := S+1
QUERY	P!S := ?; S := S+1
LSTR $nC_1 \ldots C_n$	P!S := "$C_1 \ldots C_n$"; S := S+1

The statements TRUE and FALSE are present to improve portability between machines that use different representations for the integers. For instance, on machines using ones complement or sign and modulus arithmetic, TRUE is not equivalent to LN -1.

Indirect assignments and assignments to elements of word and byte arrays use the statements STIND and PUTBYTE whose meanings are given in table 5.3.

Statement	Meaning
`STIND`	`!(P!(S-1)) := P!(S-2); S := S-2`
`PUTBYTE`	`(P!(S-2))%(P!(S-1)) := P!(S-3); S := S-3`

Assuming `ptr` is in global 200, the following assignments:

```
!ptr := 12;  ptr!3 := 99; ptr%3 := 65
```

translate into the following OCODE:

```
LN 12  LG 200  STIND
LN 99  LG 200  LN 3  PLUS    STIND
LN 65  LG 200  LN 3  PUTBYTE
```

7.4 Expression operators

The monadic expression operators only affect the topmost item of the stack and do not change the value of `S`. They are shown in the next table.

Statement	Meaning
`RV`	`P!(S-1) := !  P!(S-1)`
`ABS`	`P!(S-1) := ABS P!(S-1)`
`NEG`	`P!(S-1) := - P!(S-1)`
`NOT`	`P!(S-1) := ~ P!(S-1)`

All dyadic expression operators take two operands from stack replacing them the result and decrementing `S` by 1. These operators are shown in the following table.

Statement	Meaning
`GETBYTE`	`P!(S-2) := P!(S-2) % P!(S-1)`
`MULT`	`P!(S-2) := P!(S-2) * P!(S-1)`
`DIV`	`P!(S-2) := P!(S-2) / P!(S-1)`
`REM`	`P!(S-2) := P!(S-2) REM P!(S-1)`
`PLUS`	`P!(S-2) := P!(S-2) + P!(S-1)`
`MINUS`	`P!(S-2) := P!(S-2) - P!(S-1)`
`EQ`	`P!(S-2) := P!(S-2) = P!(S-1)`
`NE`	`P!(S-2) := P!(S-2) ~= P!(S-1)`
`LS`	`P!(S-2) := P!(S-2) < P!(S-1)`
`GR`	`P!(S-2) := P!(S-2) > P!(S-1)`
`LE`	`P!(S-2) := P!(S-2) <= P!(S-1)`
`GE`	`P!(S-2) := P!(S-2) >= P!(S-1)`
`LSHIFT`	`P!(S-2) := P!(S-2) << P!(S-1)`
`RSHIFT`	`P!(S-2) := P!(S-2) >> P!(S-1)`
`LOGAND`	`P!(S-2) := P!(S-2) & P!(S-1)`
`LOGOR`	`P!(S-2) := P!(S-2) | P!(S-1)`
`EQV`	`P!(S-2) := P!(S-2) EQV P!(S-1)`
`NEQV`	`P!(S-2) := P!(S-2) NEQV P!(S-1)`

Vector subscription (E_1`!`E_2 is implemented using `plus` and `RV`.

7.5 Procedures

The design of the OCODE statements for procedure call, save and return have been designed with care to allow code generators as much freedom as possible. The mechanism allows some arguments to be passed in registers if this is required, and the distribution of work between the code for a call and the code at the entry point of a procedure is up to the implementer. In a typical program there are about five calls for each procedure and so there is some incentive to keep the size of the call small by transferring some of the work to the save sequence.

The compilation of a procedure definition generates an OCODE sequence of the following form:

ENTRY Li n C_1 ... C_n
SAVE s
body of procedure
ENDPROC

Li is the label allocated for the procedure entry point. As a debugging aid, the length of the procedure name is given by n and its characters by the $C_1 \ldots C_n$. The SAVE statement specifies the initial setting of S, which is just the save space size (=3) plus the number of formal parameters. The state of the stack just after procedure entry is shown in figure 7.3.

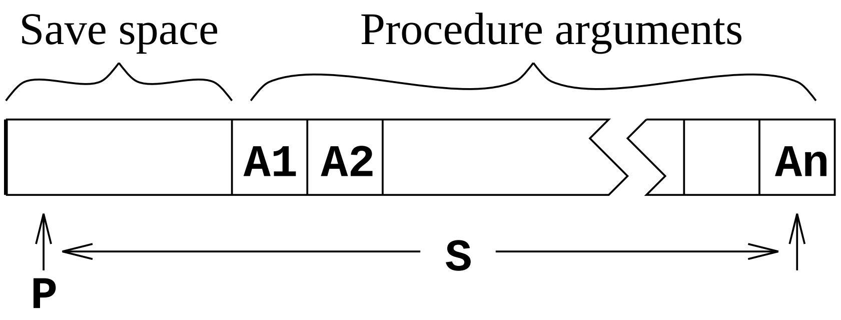

Figure 7.3: The stack frame on procedure entry

The save space is used to hold the previous value of P, the return address and the function entry address. Thus, the first argument of a function is always at position 3 relative to the P pointer. On some older versions of BCPL the size of the save space was different.

The end of the procedure is marked by the ENDPROC statement which is non executable but allows the code generator to keep track of nested procedure definitions. In early versions of OCODE, the first two arguments of ENTRY were interchanged and ENDPROC was given a numerical argument.

The language insists that arguments are laid out in consecutive locations on the stack and that there is no limit to their number. This suggests that a good strategy is to place the values of procedure arguments in the locations they must occupy when the procedure is entered. Thus, a typical call $E(E_1, \ldots, E_n)$ is compiled by first incrementing `S` to leave room for the save space in the new stack frame, then generate code to evaluate the arguments $E_1, \ldots, E_n$ before generating code for E. The state is then as shown in figure 7.4. Finally, either `FNAP` k or `RTAP` k is generated, depending on whether a function or routine call is being compiled. Notice that k is the distance between the old and new stack frames.

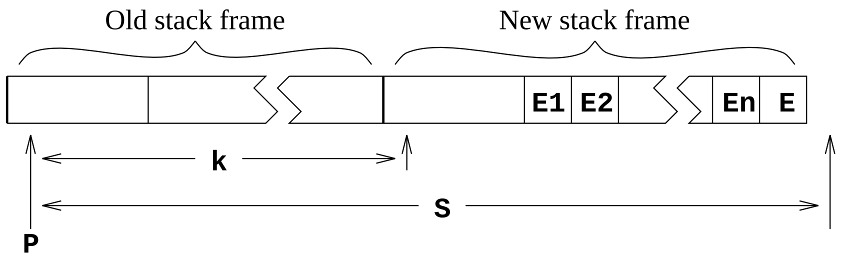

Figure 7.4: The moment of calling `E(E1,E2,...En)`

The return from a routine is performed by `RTRN` which restores the previous value of `P` and resumes execution from the return address. The return from a function is performed by `FNRN` just after the function result has been evaluated on the top of the stack. `FNRN` performs the same action as `RTRN`, after placing the function result in a special register (`A`) ready for `FNAP` to store it in the required location in the previous stack frame.

7.6 Control

The statement `LAB L`n set the value of label `L`n to the current position in the OCODE program. An unconditional transfer to this label can be performed by the satement `JUMP L`n. Conditional jumps inspect the value on the top of the stack `P!(S-1)`. `JT L`n will make the jump if it is `TRUE`, and `JF L`n will jump if `FALSE`. The translation of the command `GOTO` E is the translation of E followed by the OCODE statement `GOTO`. It thus takes the destination address from the top of the stack.

If the command `RESULTIS` E occurs in a context where the value of E is immediately returned as the result of a function, it uses `FNRN`; but in other contexts, its translation is code to evaluate E followed by a statement of the form `RES L`n. This will place the result in the special register (`A`) and jump to the label `L`n, where a statement of the form `RSTACK` k will be present to accept the value and place it in `P!`k while setting `S` to $k + 1$.

The OCODE statement:

$$\texttt{SWITCHON}\ n\ \texttt{L}dK_1L_1 \ldots K_nL_n$$

is used in the compilations of switches. It makes a jump determined by the value on the top of the stack. Its first argument (n) is the number of cases in the switch and the second argument (`L`d) is the the default label. K_1 to K_n are the case constants and L_1 to L_n are the corresponding labels.

The `FINISH` statement is the compilation of the BCPL `FINISH` command. It is converted into code equivalent to `stop(0)` by the code generator.

7.7 Directives

Sometimes the size of the stack frame changes other than in the course of expression evaluation. This happens, for instance, when control leaves a block in which local variables were declared. The statement `STACK` s informs the code generator that the size of the current stack frame is now s.

The `STORE` statement is used to inform the code generator that the point separating the declarations and body of a block has been reached and that any anonymous results on the stack are actually initialised local variables and so should be stored in their true stack locations.

Static variables and tables are allocated space in the program area using statements of the form `ITEMN` n, where n is the initial value of the static cell. The elements of table are placed in consecutive locations by consective `ITEMN` statements. A label may be set to the address of a static cell by preceding the `ITEMN` statement by a statement of the form `DATALAB L`n. In earlier versions of OCODE, there was an `ITEML` statement used in the compilation of non global procedures and labels.

The `SECTION` and `NEEDS` directives in a BCPL program translate into `SECTION` and `NEEDS` statements of the form:

$$\texttt{SECTION}\ nC_1 \ldots C_n\ \texttt{NEEDS}\ nC_1 \ldots C_n$$

where C_1 to C_n are the characters of the `SECTION` or `NEEDS` name and n is the length.

The end of an OCODE module is marked by the `GLOBAL` statement which contains information about global procedures and labels. The form of the `GLOBAL` statement is as follows:

$$\texttt{GLOBAL}\ nK_1L_1 \ldots K_nL_n$$

where n is the number of items in the global initialisation list. K_i is the global number and L_i is its label. When a module is loaded its global entry points must be initialised.

7.8 Discussion

A very early version of OCODE used a three address code in which the operands were allowed to be the sum of up to three simple values with a possible indirection. The intention was that reasonable code should be obtainable even when codegenerating one statement at a time. It was soon found more convenient to use an intermediate code that separates the accessing of values from the application of operators. This improved portability by making it possible to implement very simple non optimising codegenerators. Optimising codegenerators could absorb several OCODE statements before emitting compiled code.

The `TRUE` and `FALSE` statements were added in 1968 to improve portability to machines using sign and modulus or one's complement arithmetic. Luckily two's complement arithmetic has now become the norm. Other extension to OCODE, notably the `ABS`, `QUERY`, `GETBYTE` and `PUTBYTE` statements were added as the corresponding constructs appeared in the language.

In 1980, the BCPL changed slightly to permit position independent code to be compiled. This change specified that non global labels and procedures were no longer variables, and the current version of OCODE reflects this change by the introduction of the `LF` statement and the removal of the old `ITEML` statement that used to allocate static cells for such entry points.

Another minor change in this version of OCODE is the elimination of the `ENDFOR` statement that was provided to fix a problem on 16-bit word addressed machines with more than 64 Kbytes of memory.

Chapter 8

The Design of Cintcode

The original version of Cintcode was a byte stream interpretive code designed to be both compact and capable of efficient interpretation on small 16 bit machines machines based on 8 bit micro processors such as the Z80 and 6502. Versions that ran on the BBC Microcomputer and under CP/M were marketed by RCP Ltd [2]. The current version of Cintcode was extended for 32 bit implementations of BCPL and mainly differs from the original by the provision of 32 bit operands and the removal of a size restriction of the global vector.

There is now also a version of Cintcode for 64-bit implementations of BCPL. This is almost identical to the 32-bit version. A nineth Cintcode register (`MW`) has been added. This is normally zero but can be set by a new Cintcode instruction (`MW`), see below. On 64-bit implementations, the instructions that take four byte immediate operands, namely `KW`, `LLPW`, `LW`, `LPW`, `SPW`, `APW`, `AW` and `MW`, sign extend the four byte immediate operand before adding the `MW` register into the senior half of the 64-bit result before reseting the `MW` to zero. In this version static variable a allocated 64-bit 8 byte aligned locations.

The Cintcode machine has nine registers as shown in figure 8.1.

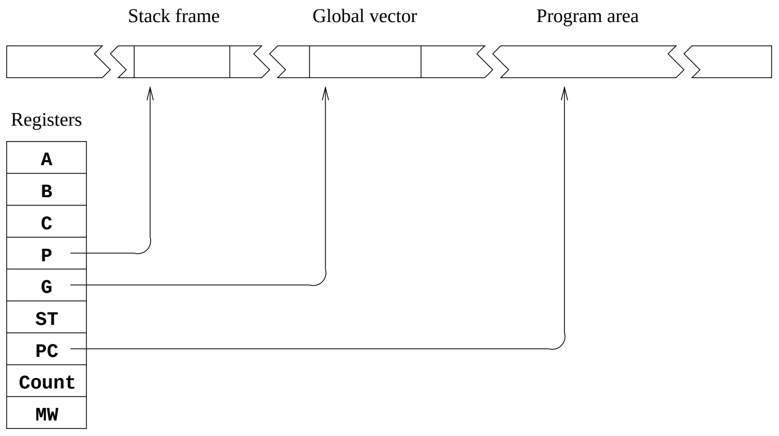

Figure 8.1: The Cintcode machine

The registers `A` and `B` are used for expression evaluation, and `C` is used in in byte subscription. `P` and `G` are pointers to the current stack frame and the global vector, respectively. `ST` was intended as a status register but is currently not used, and `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 causes a second (faster) interpreter to be used.

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 imediately 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 stings, integers, tables and global initialisation data.

8.1 Designing for Compactness

To obtain a compact encoding, information theory suggests that each function code should occur with approximately equal frequency. The self compilation of the BCPL compiler, as shown in figure 4.2, was the main benchmark test used to generate frequency information and a summary of how often various operations are used during this test is given in table 8.1. This data was produced using the tallying feature controlled by the `stats` command, described on page 87.

The statistics from different programs vary greatly, so while encoding the common operations really compactly, there is graceful degradation for the rarer cases ensuring that even unusual programs are handled reasonably well. There are, for instance, several one byte instructions for loading small integers, while larger integers are handled using 2, 3 and 5 byte instructions. The intention is that small changes in a source program should cause small small changes in the size of the corresponding compiled code.

Having several variant instructions for the same basic operation does not greatly complicate the compiler. For example the four variants of the `AP` instruction that adds a local variable into register `A` is dealt with by the following code fragment taken from the codegenerator.

```
TEST 3<=n<=12 THEN gen(f_ap0 + n)
              ELSE TEST 0<=n<=255
                   THEN genb(f_ap, n)
                   ELSE TEST 0<=n<=#xFFFF
                        THEN genh(f_aph, n)
                        ELSE genw(f_apw, n)
```

It is clear from table 8.1 that accessing variables and constants requires special care, and that conditional jumps, addition, procedure calls and indirection are also important. Since access to local variables accounts for about a quarter of the operations performed, about this proportion of codes were allocated to instructions concerned with local variables. Local variables are allocated words in the stack starting at position 3

Operation	**Executions**	**Static count**
Loading a local variable	3777408	1479
Updating a local variable	1965885	1098
Loading a global variable	5041968	1759
Updating a global variable	796761	363
Using a positive constant	4083433	1603
Using a negative constant	160224	93
Conditional jumps (all)	2013013	488
Conditional jumps on zero	494282	267
Unconditional direct jump	254448	140
Unconditional indirect jumps	152646	93
Procedure calls	1324206	1065
Procedure returns	1324204	381
Binary chop switches	43748	12
Label vector switches	96461	17
Addition	2135696	574
Subtraction	254935	111
Other expression operations	596882	74
Loading a vector element	1356315	429
Updating a vector element	591268	137
Loading a byte vector element	476688	53
Updating a byte vector element	405808	29

Table 8.1: Counts from the BCPL self compilation test

relative to the `P` pointer and, as one would expect, small numbered locals are used far more frequently than the others, so operations on low numbered locals often have single byte codes.

Although not shown here, other statistics, such as the distribution of relative addressing offsets and operand values, influenced the design of Cintcode.

8.1.1 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:

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

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

8.1.2 Composite Instructions

Compactness can be improved by combining commonly occurring pairs (and triples) of operations into a single instructions. Many such composite instructions occur in Cintcode; for instance, `AP3` adds local `3` to the `A` register, and `L1P6` will load `v!1` into register `A`, assuming `v` is held in local `6`.

8.1.3 Relative Addressing

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

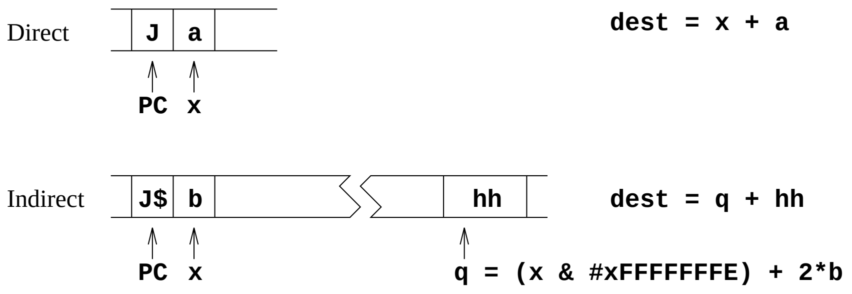

Figure 8.2: The relative addressing mechanism

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 8.2, using the instructions `J` and `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 desination, 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.

8.2 The Cintcode Instruction Set

The resulting selection of function codes is shown in Table 8.2 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.

8.2.1 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 7.3 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 statics 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 75.

	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	CHGCO
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$	MW	-

Table 8.2: The Cintcode function codes

8.2.2 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 succesively 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 variables in the range 0 to 255, `LG1` load 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 ponters to local, global and static variables.

8.2.3 Indirect Load

`GBYT`		`A := B%A`
`RV`		`A := A!0`
`RV`n	$1 \le n \le 6$	`A := A!`n
`RVP`n	$3 \le n \le 7$	`A := P!`n`!A`
`L0P`n	$3 \le n \le 12$	`B := A; A := P!`n`!0`
`L1P`n	$3 \le n \le 6$	`B := A; A := P!`n`!1`
`L2P`n	$3 \le n \le 5$	`B := A; A := P!`n`!2`
`L3P`n	$3 \le n \le 4$	`B := A; A := P!`n`!3`
`L4P`n	$3 \le n \le 4$	`B := A; A := P!`n`!4`
`L`n`G` b	$0 \le n \le 2$	`B := A; A := G!`b`!`n
`L`n`G1` b	$0 \le n \le 2$	`B := A; A := G!(`b`+256)!`n
`L`n`GH` 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.

8.2.4 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 120. Addition is the most frequently used arithmetic operation and so there are various special instructions improve its efficiency.

`A`*n*	$1 \le n \le 5$	`A := A +` *n*
`S`*n*	$1 \le n \le 4$	`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 a 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.

`AP`*n*	$3 \le n \le 12$	`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.

8.2.5 Simple Assignment

`SP`*n*	$3 \le n \le 16$	`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.

8.2.6 Indirect Assignment

PBYT		B%A := C
XPBYT		A%B := C
ST		A!0 := B
STn	$1 \leq n \leq 3$	A!n := B
ST0Pn	$3 \leq n \leq 4$	P!n!0 := A
ST1Pn	$3 \leq n \leq 4$	P!n!1 := A
STPn	$3 \leq n \leq 5$	P!n!A := B
S0G b		G!b!0 := A
S0G1 b		G!(b+256)!0 := A
S0GH h		G!h!0 := A

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

8.2.7 Procedure calls

At the moment a function or routine is called the state of the stack is as shown in figure 8.3. At the entry point of a function or routine the first argument, if any, will be in register `A` and in memory `P!3`.

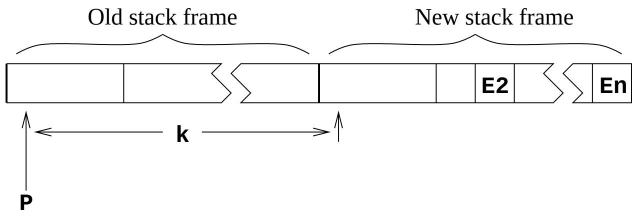

Figure 8.3: The moment of calling `E(E1,E2,...En)`

K*n*	$3 \le n \le 11$
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.

K*n*G *b*	$3 \le n \le 11$
K*n*G1 *b*	$3 \le n \le 11$
K*n*GH *h*	$3 \le n \le 11$

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`.

8.2.8 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 8.1.3. 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`

8.2.9 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 8.1.3 for details on how resolving half words are interpreted.

`SWB` *filler* n *dlab* K_1 L_1 $\ldots K_n$ 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 8.1.3 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 can be further improved by preceeding 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.

8.2.10 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. If A is zero then the interpreter returns with exception code P!4.

If A is -1 it set register count to P!4, setting A to the previous value of count. Changing the value of count may change which of the two interpreters is used. For more details see Section ??.

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 3.3. 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.

8.2.11 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.

8.2.12 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 Kn KnG KnG1 KnGH
SWL SWB MDIV CHGCO
```

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

8.2.13 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 54.

Chapter 9

The Design of Sial

Sial is an internal intermediate assembly language designed for BCPL. The first version was called Cial (Compact Internal Assembly Language) was pronounced "Seal". It was essentially an assembly language for Cintcode with the same function code mnemonics and the same abstract machine registers. It was soon found that rather than having a variety of codes to load an integer constant (such as `L0`, `L1`, `L2`, `LM1`, `LW`, `LH` or `L`), it was better to have one function code to load positive integers and another for negative ones with the values specified by operands. This form is more convenient for translation and easier to compress. The new language is called Sial (also pronouced "seal") with the S standing for smaller. Sial therefore has fewer function codes than Cintcode and most of them take operands but still uses the same abstract machine registers. Although Cintcode load instructions save the value of the `A` register in `B` before setting `A`, Sial loads typically do not.

Sial was designed as an experiment in the compact representation of algorithms that can be easily just-in-time compiled into code for any target machine. Its secondary purpose was to allow an easy way to generate native code translations of BCPL programs giving typically a five to ten fold speedup over the Cintcode interpretive version. An experienced programmer can normally modify an existing naive Sial translator to generate reasonable code for a new target in one or two days.

The following sections give a specification of Sial, outline the implementation of a particular translator (`sial-386`) and finally outline how Sial can be compacted.

9.1 The Sial Specification

Sial consists of a stream of directives and instructions each starting with an opcode followed by operands. Both opcodes and operands and encoded using integers each prefixed by a letter specifying what kind of value it represents. The prefixes are as follows:

`F`	An opcode or directive
`P`	A stack offset, 0 to `#xFFFFFF`
`G`	A global variable number, 0 to 65535
`K`	A 24-bit unsigned constant, often small in value
`W`	A 32-bit signed integer, used for static data and large constants
`C`	A byte in range 0 to 255
`L`	A label generated by translation phase
`M`	A label generated by the Sial codegenerator

The instructions are for an abstract machine with internal registers

`a`	The main accumulator, function first arg and result register
`b`	The second accumulator used in dyadic operations
`c`	Register used by `pbyt` and `xpbyt`, and possibly currupted by some other instructions, such as `mul`, `div`, `rem`, `xdiv` and `xrem`
`P`	Pointer to the base of the current stack frame
`G`	Pointer to the base of the Global Vector
`PC`	Set by jump and call insntrunctions

The opcodes and directives are as follows:

Mnemonic	Operand(s)	Meaning
`lp`	`Pn`	`a := P!n`
`lg`	`Gn`	`a := G!n`
`ll`	`Ln`	`a := !Ln`
`llp`	`Pn`	`a := @ P!n`
`llg`	`Gn`	`a := @ G!n`
`lll`	`Ln`	`a := @ !Ln`
`lf`	`Ln`	`a := byte address of entry point Ln`
`l`	`Kn`	`a := n`
`lm`	`Kn`	`a := - n`
`sp`	`Pn`	`P!n := a`
`sg`	`Gn`	`G!n := a`
`sl`	`Ln`	`!Ln := a`
`ap`	`Pn`	`a := a + P!n`
`ag`	`Gn`	`a := a + G!n`
`a`	`Kn`	`a := a + n`
`s`	`Kn`	`a := a - n`

lkp	Kk Pn	a := P!n!k
lkg	Kk Gn	a := G!n!k
rv		a := ! a
rvp	Pn	a := P!n!a
rvk	Kn	a := a!k
st		!a := b
stp	Pn	P!n!a := b
stk	Kn	a!n := b
stkp	Kk Pn	P!n!k := a
skg	Kk Gn	G!n!k := a
xst		!b := a
k	Pn	Call a(b,...) incrementing P by n leaving b in a
kpg	Pn Gg	Call Gg(a,...) incrementing P by n
neg		a := - a
not		a := ~ a
abs		a := ABS a
xdiv		a := a / b; c := ?
xrem		a := a REM b; c := ?
xsub		a := a - b; c := ?
mul		a := b * a; c := ?
div		a := b / a; c := ?
rem		a := b REM a; c := ?
add		a := b + a
sub		a := b - a
eq		a := b = a
ne		a := b ~= a
ls		a := b < a
gr		a := b > a
le		a := b <= a
ge		a := b >= a
eq0		a := a = 0
ne0		a := a ~= 0
ls0		a := a < 0
gr0		a := a > 0
le0		a := a <= 0
ge0		a := a >= 0

lsh		a := b << a
rsh		a := b >> a
and		a := b & a
or		a := b a—
xor		a := b XOR a
eqv		a := b EQV a
gbyt		a := b % a
xgbyt		a := a % b
pbyt		b % a := c
xpbyt		a % b := c
swb	Kn Ld K1 L1 ... Kn Ln	Binary chop switch, Ld default
swl	Kn Ld L1 ... Ln	Label vector switch, Ld default
xch		Swap a and b
atb		b := a
atc		c := a
bta		a := b
btc		c := b
atblp	Pn	b := a; a := P!n
atblg	Gn	b := a; a := G!n
atbl	Kk	b := a; a := k
j	Ln	Jump to Ln
rtn		Procedure return
goto		
ikp	Kk Pn	a := P!n + k; P!n := a
ikg	Kk Gn	a := G!n + k; G!n := a
ikl	Kk Ln	a := !Ln + k; !Ln := a
ip	Pn	a := P!n + a; P!n := a
ig	Gn	a := G!n + a; G!n := a
il	Ln	a := !Ln + a; !Ln := a
jeq	Ln	Jump to Ln if b = a
jne	Ln	Jump to Ln if b ~= a
jls	Ln	Jump to Ln if b < a
jgr	Ln	Jump to Ln if b > a
jle	Ln	Jump to Ln if b <= a
jge	Ln	Jump to Ln if b >= a
jeq0	Ln	Jump to Ln if a = 0
jne0	Ln	Jump to Ln if a ~= 0
jls0	Ln	Jump to Ln if a < 0
jgr0	Ln	Jump to Ln if a > 0
jle0	Ln	Jump to Ln if a <= 0
jge0	Ln	Jump to Ln if a >= 0
jge0m	Mn	Jump to Mn if a >= 0

`brk`		Breakpoint instruction
`nop`		No operation
`chgco`		Change coroutine
`mdiv`		`a := muldiv(P!3, P!4, P!5)`
`sys`		System function
`section`	`Kn C1 ... Cn`	Name of section
`modstart`		Start of module
`modend`		End of module
`global`	`Kn G1 L1 ... Gn Ln`	Global initialisation data
`string`	`Ml Kn C1 ... Cn`	String constant
`const`	`Mn Ww`	Large integer constant
`static`	`Ln Kk W1 ... Wk`	Static variable or table
`mlab`	`Mn`	Destination of `jge0m`
`lab`	`Lm`	Program label
`lstr`	`Mn`	`a := Mn` (pointer to string)
`entry`	`Kn C1 ... Cn`	Start of a function

The command `bcpl2sial` can be used to translate a BCPL program into Sial. For example, it will compile the following program:

```
SECTION "fact"

GET "libhdr"

LET start() = VALOF
{ FOR i = 1 TO 5 DO writef("fact(%n) = %i4*n", i, fact(i))
  RESULTIS 0
}

AND fact(n) = n=0 -> 1, n*fact(n-1)
```

into

```
F104
F103 K4 C102 C97 C99 C116
F113 K5 C115 C116 C97 C114 C116
F111 L1
F11 K1
F13 P3
F111 L4
F3 P3
F69
F9 L2
F31 P9
F13 P9
F3 P3
F13 P8
F112 M1
F32 P4 G94
F79 K1 P3
F75 K5
F89 L4
```

```
F11 K0
F77
F107 M1 K15 C102 C97 C99 C116 C40 C37 C110
 C41 C32 C61 C32 C37 C105 C52 C10
F113 K4 C102 C97 C99 C116
F111 L2
F92 L5
F11 K1
F77
F111 L5
F12 K1
F16 P3
F69
F9 L2
F31 P4
F73 P3
F39
F77
F106 K1 G1 L1 G94
F105
```

This can be converted in the following more readable for using the `sial-sasm` command:

```
MODSTART
SECTION K4 C102 C97 C99 C116

//Entry to: start
ENTRY   K5 C115 C116 C97 C114 C116
LAB     L1
L       K1
SP      P3
LAB     L4
LP      P3
ATB
LF      L2
K       P9
SP      P9
LP      P3
SP      P8
LSTR    M1
KPG     P4 G94
IKP     K1 P3
ATBL    K5
JLE     L4
L       K0
RTN
STRING  M1 K15 C102 C97 C99 C116 C40 C37 C110 C41 C32 C61 C32 C37 C105 C52 C10

//Entry to: fact
ENTRY   K4 C102 C97 C99 C116
LAB     L2
JNE0    L5
L       K1
RTN
LAB     L5
LM      K1
```

```
AP      P3
ATB
LF      L2
K       P4
ATBLP   P3
MUL
RTN
GLOBAL K1
G1 L1
G94
MODEND
```

9.2 The sial-386 Translator

The source of an Sial translator that generates Intel 386 assembly language is `com/sial-386.b`. It is a simple program about 750 lines in length based on the `sial-sasm` program. It causes the readable version of the Sial source to appear as comments interspersed with the corresponding Intel 386 translations. For the example, program given above it outputs the following assembly language.

```
# Code generated by sial-386

.text
.align 16
# MODSTART
# SECTION K4 C102 C97 C99 C116

# Entry to: start
# ENTRY   K5 C115 C116 C97 C114 C116
# LAB     L1

LA1:
 movl %ebp,0(%edx)
 movl %edx,%ebp
 popl %edx
 movl %edx,4(%ebp)
 movl %eax,8(%ebp)
 movl %ebx,12(%ebp)
# L       K1
 movl $1,%ebx
# SP      P3
 movl %ebx,12(%ebp)
# LAB     L4
LA4:
# LP      P3
 movl 12(%ebp),%ebx
# ATB
 movl %ebx,%ecx
# LF      L2
 leal LA2,%ebx
# K       P9
 movl %ebx,%eax
 movl %ecx,%ebx
 leal 36(%ebp),%edx
 call *%eax
# SP      P9
 movl %ebx,36(%ebp)
# LP      P3
 movl 12(%ebp),%ebx
# SP      P8
 movl %ebx,32(%ebp)
# LSTR    M1
 leal MA1,%ebx
 shrl $2,%ebx
# KPG     P4 G94
 movl 376(%esi),%eax
 leal 16(%ebp),%edx
 call *%eax
# IKP     K1 P3
 movl 12(%ebp),%ebx
 incl %ebx
 movl %ebx,12(%ebp)
# ATBL    K5
 movl %ebx,%ecx
 movl $5,%ebx
# JLE     L4
 cmpl %ebx,%ecx
 jle LA4
# L       K0
 xorl %ebx,%ebx
# RTN
 movl 4(%ebp),%eax
 movl 0(%ebp),%ebp
 jmp *%eax
# STRING  M1 K15 C102 C97 C99 C116 C40 C37 C110 C41 C32 C61 C32 C37 C105 C52 C10
.data
 .align 4
MA1:
 .byte 15
 .byte 102
 .byte 97
 .byte 99
 .byte 116
 .byte 40
 .byte 37
 .byte 110
 .byte 41
 .byte 32
 .byte 61
 .byte 32
 .byte 37
 .byte 105
 .byte 52
 .byte 10
 .text

# Entry to: fact
# ENTRY   K4 C102 C97 C99 C116
# LAB     L2
```

```
LA2:
 movl %ebp,0(%edx)
 movl %edx,%ebp
 popl %edx
 movl %edx,4(%ebp)
 movl %eax,8(%ebp)
 movl %ebx,12(%ebp)
# JNE0    L5
 orl %ebx,%ebx
 jne LA5
# L       K1
 movl $1,%ebx
# RTN
 movl 4(%ebp),%eax
 movl 0(%ebp),%ebp
 jmp *%eax
# LAB     L5
LA5:
# LM      K1
 movl $-1,%ebx
# AP      P3
 addl 12(%ebp),%ebx
# ATB
 movl %ebx,%ecx
# LF      L2
 leal LA2,%ebx
# K       P4
 movl %ebx,%eax
 movl %ecx,%ebx
 leal 16(%ebp),%edx
 call *%eax
# ATBLP   P3
 movl %ebx,%ecx
 movl 12(%ebp),%ebx
# MUL
 movl %ecx,%eax
 imul %ebx
 movl %eax,%ebx
# RTN
 movl 4(%ebp),%eax
 movl 0(%ebp),%ebp
 jmp *%eax
# GLOBAL K1

.globl fact

.globl _fact
fact:
_fact:
 movl 4(%esp),%eax
# G1 L1
 movl $LA1,4(%eax)
# G94
 ret

# MODEND
```

When implementing `sial-386` it was necessary to decide how the Intel registers were to be used and what the BCPL calling sequence should be. The chosen register allocation was as follows:

Intel register	Use
`%eax`	A work register
`%ebx`	The `A` register
`%ecx`	The `B` register
`%edx`	The `C` register
`%esi`	The `G` pointer
`%edi`	A work register
`%ebp`	The `P` pointer

The chosen BCPL calling sequence is as follows:

```
                                   # Entry address must be in %eax
                                   # The first argument must be in A(%ebx)
leal <stack increment>(%ebp),%edx # Set C(%edx) to the new P pointer
call *%eax                         # Subroutine jump to the entry point
```

The entry sequence is as follows:

```
                    # The first argument is in A(%ebx)
                    # The new P pointer is in C(%edx)
movl %ebp,0(%edx)   # C!0  := P
movl %edx,%ebp      # P    := C
popl %edx           # Get the return address
movl %edx,4(%ebp)   # P!1  := return address
movl %eax,8(%ebp)   # P!2  := entry address
movl %ebx,12(%ebp)  # P!3  := the first argument
```

The return sequence is as follows:

```
                    # The result is in A(%ebx)
movl 4(%ebp),%eax   # Get the return address
movl 0(%ebp),%ebp   # P := the saved P pointer
jmp *%eax           # Jump to the return address
```

The structure of `sial-386` is simple. It mainly consists of a large switch within the function `scan` that has a case for each function code and directive. For example, the case for the function code `kpg` is as follows:

```
      CASE f_kpg:    cvfpg("KPG") // Call Gg(a,...) incrementing P by n
                     writef("*n movl %n(%%esi),%%eax", 4*gval)
                     writef("*n leal %n(%%ebp),%%edx", 4*pval)
                     writef("*n call **%%eax")
                     ENDCASE
```

The call `cvfpg(‘‘KPG’’)` reads the Sial statement knowing it is of the form: `KPG` `P`*k* `G`*n*). This outputs the statement as an assembly language comment after placing *k* and *n* in `pval` and `gval`, respectively. The three `writef` calls then output the three assembly language instructions for the `KPG` operation, and `ENDCASE` transfers control to where the next Sial statement is processed. All the other cases are equally simple.

The section name of the program, which must be present, compiles into a C callable function that initialises the BCPL global vector with the entry points defined in this module. To complete the 386 implementation, there is a short handwritten assembly language library `natbcpl/sysasm/mlib.s` that defines the BCPL callable functions `sys`, `changeco` and `muldiv`. The program must be linked the compiled versions of the BCPL library modules `BLIB` and `DLIB`, and also `clib` whose source is in `natbcpl/sysc/clib.c` and a program typically called `initprob.c` that defines the function `inisections` to invoke all the global initialisation functions. The file `initprog.c` is normally created by a call such as:

```
makeinit prog.b to initprog.c
```

The resulting `initprog.c` is typically:
// Initialisation file written by MakeInit version 1.8

```
#include "bcpl.h"

WORD stackupb=10000;
WORD gvecupb=1000;

// BCPL sections
extern BLIB(WORD *);    // file (run-time library)
extern DLIB(WORD *);    // file (system dependent library)
extern prog(WORD *);    // file prog.b

void initsections(WORD *g) {
    BLIB(g);    // file (run-time library)
    DLIB(g);    // file (system dependent library)
    prog(g);    // file prog.b

    return;
}
```

9.3 Compaction of Sial

To be written.

Chapter 10

The MC Package

This chapter describes the MC package which provides a machine independent way to generate and executing native machine code at runtime. The work on this package started in January 2008 and is still under development, however, it currently works well enough to run the n-queens problem on i386 machines about 24 times faster than the normal Cintcode interpretive version. MC package development is performed in the directory `BCPL/bcplprogs/mc/` and fairly stable versions are copied to `BCPL/cintcode/g/mc.h`, `BCPL/cintcode/com/mci386.b` and `BCPL/cintcode/cin/mci386` which can be used from any working directory.

The package is based on a simple machine independent abstract machine code called MC which is easily translated into machine instructions for most architectures. Although native code is generated by MC calls such as `mcRDX(mc_add, mc_b, 20, mc_d)`, MC has a corresponding assembly language to assist debugging. The assembly form of the instruction generated by the previous call is `ADD B,20(D)` meaning set register B to the sum of B and the contents of the memory location whose address is 20 plus the value of register D. MC instructions are fairly low level and typically translate into single native code instructions for most architectures. This example translates into the i386 GNU statement: `addl 20(%edx), %ebx`.

The first operand is the destination for any instruction that updates a register or memory location. Thus assignments are always from right to left as in most programming languages but unlike many assembly codes where, for instance, `movl 20(%edx), %ebx` updates the second operand.

The MC machine has six registers A, B, C, D, E and F that are directly available to the programmer, and also a program counter, stack pointer, stack frame pointer and a condition code register, although these cannot be accessed explicitly.

10.1 MC Example

The following program is a simple demonstration of the i386 version of the MC package.

```
GET "libhdr"
GET "mc.h"
```

```
MANIFEST {
  A=mc_a; B=mc_b; C=mc_c; D=mc_d; E=mc_e; F=mc_f
  a1=1; a2; a3
}

LET start() = VALOF
{ // Load the dynamic code generation package for i386 machines.
  LET mcseg, mcb, n = globin(loadseg("mci386")), 0, 0
  UNLESS mcseg DO
  { writef("Trouble with MC package: mci386*n")
    GOTO fin
  }
  // Create an MC instance for 10 functions with a data space
  // of 100 words and code space of 4000 words.
  mcb := mcInit(10, 100, 4000)
  UNLESS mcb DO
  { writef("Unable to create an mci386 instance*n")
    GOTO fin
  }
  mc := 0                  // Currently no selected MC instance.
  mcSelect(mcb)            // Select the new MC instance.

  mcK(mc_debug, #b0011)    // Trace comments and MC instructions.

  mcKKK(mc_entry, 1, 3, 5) // Entry point for function 1
                           // having 3 arguments and 5 local variables

  mcK(mc_debug, #b1111)    // Trace comments, MC instructions, target
                           // instructions and the compiled binary code.

  mcRA(mc_mv,  A, a1)      // A := <arg 1>
  mcRA(mc_add, A, a2)      // A := A + <arg 2>

  n := mcNextlab()
  mcL(mc_lab, n)           // Ln:
  mcRA(mc_add, A, a3)      // A := A + <arg 3>
  mcR(mc_dec, A)           // A := A - 1
  mcRK(mc_cmp, A, 100)
  mcJS(mc_jlt, n)          // IF A<100 JMP Ln

  mcK(mc_debug, #b0011)    // Trace only comments and MC instructions.
  mcF(mc_rtn)              // Return from function 1 with result in A.
  mcF(mc_endfn)            // End of function 1 code.
  mcF(mc_end)              // End of dynamic code generation.

  writef("*nF1(10, 20, 30) => %n*n", mcCall(1, 10, 20, 30))
fin:
  IF mcseg DO unloadseg(mcseg)
  RESULTIS 0
}
```

When this program runs it outputs the following.

```
//    ENTRY 1 3 5
//    DEBUG 15
//    MV A,A1
```

```
        movl 20(%ebp), %eax
  573:  8B 45 14
//    ADD A,A2
        addl 24(%ebp), %eax
  576:  03 45 18

//    LAB L1
        lab L1
  579: L1:
//    ADD A,A3
        addl 28(%ebp), %eax
  579:  03 45 1C
//    DEC A
        decl  %eax
  582:  48
//    CMP A,$100
        cmpl $100, %eax
  583:  83 F8 64
//    JLT L1
        jl L1
  586:  7C F7
//    DEBUG 3
//    RTN
//    ENDFN
//    END

F1(10, 20, 30) => 117
```

The result of 117 (= 10+20+(30-1)*3) shows that the body of the loop was correctly executed three times.

The header file (`mc.h`) defines manifests (such as `mc_mv` and `mc_add`) and globals (such as `mcK` and `mcRA`) provided by the package. The package itself must be dynamically loaded (by `globin(loadseg("mci386"))`) and then selected (by `mcSelect(mcb)`). MC instructions are compiled by calls such as `mcRA(op,...` or `mcRK(op,...` where `op` specifies the instruction or directive and the letters following `mc` (eg `RA` or `RK`) specify the sort of operands supplied.

A register operand is denoted by `R` and an integer operand by `K`. There are 9 possible kinds of memory operands denoted by `A`, `V`, `G`, `M`, `L`, `D`, `DX`, `DXs` and `DXsB`. `A` denotes an specified argument of the current function, `V` denotes a specified local variable of the current function, `G` denotes a specified BCPL global variable, `M` denotes a location in Cintcode memory specified by a BCPL pointer, `L` denotes the position within the data or code areas of the compiled code corresponding to a given label, `D` denotes a specified absolute machine address, `DX` denotes a location whose machine address is the sum of a given byte offset and register, `DXs` is similar to `DX` only the index register is scaled by a given factor of 1, 2, 4 or 8 and finally `DXsB` is like `DXs` but has a second specified register added into the effective address.

The following table summarises the MC code generation functions. The first argument is always specifies the directive or instruction and the remaining arguments specify the operands. The destination of any instruction that updates a register or memory location is always the first operand.

Function	Operands
`mcF`	No operand
`mcK`	One integer operand
`mcR`	One MC register operand
`mcA`	One operand specifying an argument number
`mcV`	One operand specifying an local variable number
`mcG`	One operand specifying a global variable number
`mcM`	One operand giving the word address of a location in Cintcode memory
`mcL`	One numeric label operand, defaulting to 32-bit relative
`mcD`	One operand giving an absolute machine address
`mcDX`	One memory operand specified by an offset added to an index register
`mcDXs`	One memory operand specified by an offset added to an index register scaled by `s` which must be 1, 2, 4 or 8
`mcDXsB`	One memory operand specified by an offset added to a base register and an index register scaled by `s` which must be 1, 2, 4 or 8
`mcJS`	Jump instructions with near relative destinations
`mcJL`	Jump instructions with possibly distant relative destinations
`mcJR`	Jump instructions with destination given by resister
`mcRA`	Two operands, `R` and `A`
`mcRV`	Two operands, `R` and `V`
`mcRG`	Two operands, `R` and `G`
`mcRM`	Two operands, `R` and `M`
`mcRL`	Two operands, `R` and `L`
`mcRD`	Two operands, `R` and `D`
`mcRDX`	Two operands, `R` and `DX`
`mcRDXs`	Two operands, `R` and `DXs`
`mcRDXsB`	Two operands, `R` and `DXsB`
`mcRR`	Two operands, `R` and `R`
`mcAR`	Two operands, `A` and `R`
`mcVR`	Two operands, `V` and `R`
`mcGR`	Two operands, `G` and `R`
`mcMR`	Two operands, `M` and `R`
`mcLR`	Two operands, `L` and `R`
`mcDR`	Two operands, `D` and `R`
`mcDXR`	Two operands, `DX` and `R`
`mcDXsR`	Two operands, `DXs` and `R`
`mcDXsBR`	Two operands, `DXsB` and `R`
`mcRK`	Two operands, `R` and `K`
`mcAK`	Two operands, `A` and `K`
`mcVK`	Two operands, `V` and `K`
`mcGK`	Two operands, `G` and `K`
`mcMK`	Two operands, `M` and `K`
`mcLK`	Two operands, `L` and `K`
`mcDK`	Two operands, `D` and `K`
`mcDXK`	Two operands, `DX` and `K`
`mcDXsK`	Two operands, `DXs` and `K`
`mcDXsBK`	Two operands, `DXsB` and `K`
`mcKK`	Two integer operands
`mcKKK`	Three integer operands
`mcPRF`	One `printf` format string and one register

10.2 MC Library Functions

`mcb := mcInit(maxfno, dsize, csize)`

Create an instance of the MC package, allocating space for *maxfno* functions, *dsize* words of data space and *csize* words of code space. The MC control block is assigned to *mcb*.

`mcSelect(mcb)`

Select an instance of the MC package by assigning *mcb* to the global variable `mc`. For efficiency reasons, `mcSelect` copies various field in the control block to global variables. If `mc` was non zero, the previous setting of the globals are saved in the previously selected MC instance. It is thus important to set `mc` to zero before the first call od `mcSelect`.

`res := mcCall(fno, a1, a2, a3)`

Call the function with number *fno* giving it the three arguments *a1*, *a2*, *a3*. The result is assigned to *res*. Function *fno* must have been defined to expect three arguments.

`mcClose()`

Close the currently selected MC instance deleting all its workspace and compiled code. It also sets `mc` to zero.

`mcPRF(mess, reg)`

This function is an invaluable debugging aid which compiles code to call the C function `printf` with the given format string (packed in the data area) and the value of the specified register. All registers, including the condition code, are preserved. The register argument may be omitted if the format string requires no register argument. Typical use of `mcPRF` is as follows:

```
        mcRK(mc_mv, D, #x01234567)
        mcRK(mc_mv, A, #x89ABCDEF)
        mcRK(mc_mv, A, #x10000000)
        mcPRF("With  D=%8x  ",  D)
        mcPRF("A=%8x  ", A)
        mcPRF("B=%8x*n", B)
        mcR(mc_div, B)
        mcPRF("the instruction:  DIV B*n")
        mcPRF("gives D=%8x  ", D)
        mcPRF("A=%8x  ", A)
        mcPRF("B=%8x*n", B)
```

This causes the following output:

```
With  D= 1234567  A=89abcdef  B=10000000
the instruction:  DIV B
gives D= 9abcdef  A=12345678  B=10000000
```

n `:= mcNextlab()`

Allocate the next available label assigning its number to *n*. Labels are use by instructions that refer to static data and in jump instructions. There is essentially no limit to the number of labels that may be allocated.

`mcComment(`*format, a, b,..., k*`)`

This is a debugging aid to make the compiled code more readable using `writef` to write a message to the listing output during code generation if the least significant bit of `mcDebug` is a one. The variable `mcDebug` is set by the DEBUG directive described below.

res `:= mcDatap()`
res `:= mcCodep()`

These calls return the current positions in the data and code area respectively.

All the other functions compile MC directives and instructions and are described below.

10.3 The MC Language

The MC abstract machine language is fairly low level and is somewhat influenced by the i386 architecture. Particularly the rather small number of MC registers allowed, the rich variety of memory addressing modes and the specification of the instructions for multiplication, division and shifts. However, it is machine independent and reasonably easy to compile into native machine code for most machines. Before describing the MC instructions, a few key features will be introduced. As mentioned earlier the MC machine has six registers named `A` to `F` which are typically mapped directly onto machine registers of the target architecture. These can be used for any purpose except for a few instructions such as `MUL`, `DIV` and the shifts which may implicitly use some of them implicitly.

When an MC function is declared it has a specified number of arguments and local variables (see the ENTRY statement below). When a function is called by the CALL instruction, the required number of arguments must have already been pushed onto the stack. On return these arguments will have been automatically popped from the stack. If the wrong number of arguments are given, the effect is undefined. By convention, the result of a function is returned in register `A`.

Numeric labels are used to refer to static data and positions in the code. They are allocated by calls of `mcNextlab`, described above. Many architectures allow both conditional and unconditional jumps to use short offsets (typically single bytes) to specify the relative address of the destination. Jump instructions automatically use short relative addresses for backward jumps if possible, but, for forward jumps, the programmer is required to give a hint. Jump instructions compiled by `mcJS` expect forward jumps to use short relative addresses while `mcJL` specifies that larger relative addresses are to be used. If a short relative address proves insuffiscent and error message is generated telling the programmer that `mcJL` should have been used. The function `mcJR` is used when the destination address of a jump instruction is in a register.

Conditional jump instructions inspect the condition code to determine whether or not to jump. The condition code is set by the `CMP`, `ADD`, `ADDC`, `SUB` and `SUBC` instructions and preserved by jump instructions (`JMP` and `Jcc`). All other instructions (including `INC` and `DEC` leave the condition code undefined.

All MC directives and instructions are described below in alphabetical order. The name of the operation is given in bold caplital letters together with the list of possible operand types. The BCPL manifest for the operation consists of the name in lower case letters preceeded by `mc_`. For example, `mc_add` is the manifest constant for the ADD operation, and since `RDXs` appears in its list of operand types, it can be compiled by, for instance, `mcRDXs(mc_add, mc_a, 20, mc_d, 4)`.

ADD `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Add the second operand into the first and set the condition code appropriately. For example, `mcRG(mc_add, mc_d, 150)` will compile code to add global 150 in register `D`.

ADDC `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Add the condition code carry bit and the second operand into the first and set the condition code appropriately. Adding 1 into the 64-bit value held in `B:A` can be done by the code generated by:

```
mcRK( mc_add,  mc_a, 1)  // Don't use INC here!
mcRK( mc_addc, mc_b, 0)
```

ALIGNC `K`

Align the next instruction to an address which is a multiple of `k` which must be 2, 4 or 8.

ALIGND `K`

Align the next item of data to an address which is a multiple of `k` which must be 2, 4 or 8.

AND `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Perform the bit wise AND of the second operand into the first.

CALL `KK`

Call the function who number is the first argument with `n` arguments that have already been pushed onto the stack when `n` is the second operand. On return these arguments will have been popped and, by convention, the result will be in register `A`.

CDQ `F`

Sign extend register `A` into `D`. That is, if `A` is positive set `D` to zero, otherwise it is to `#xFFFFFFFF`. This is normally used in conjuction with DIV.

CMP `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Set the condition code to difference between the first operand and the second. The condition code is used by conditional jumps and conditional setting instructions. For example,

```
mcRK(mc_cmp, mc_b, 100)
mcJL(mc_jle, 25)
```

will compile code to jump the label `L25` is `B<=100`, using signed arithmetic.

DATAB `K`
Assemble one byte of data with the specified value.

DATAK `K`
Assemble one aligned word of data with the specified value.

DATAL `L`
Assemble one aligned word of data initialised with the absolute address of code or data specified by the given label.

DEBUG `K`
Set the debug tracing level (`mcDebug`) to the specified value. The least significant four bits of `mcDebug` control the level of tracing as follows.

`#b0001`	Output any `mcComment` comments.
`#b0010`	Output the MC instructions.
`#b0100`	Output the target machine instructions.
`#b1000`	Output the compiled binary code.

DEC `R A V G M L D DX DXs DXsB`
Decrement the specified register or memory word by 1, leaving the condition code undefined.

DIV `K R A V G M L D DX DXs DXsB`
Divide the double length value in `D:A` by the specified operand. The result is left in `A` and the remainder in `D`. The DIV instruction performs signed arithmetic.

DLAB `L`
Set the specified label to the absolute address of the next available byte in the data area.

ENDFN `F`
This marks the end of the body of the current function.

END `F`
This directive specifies that no more code generation will be done. The system

will free all temporary work space only preseving the MC control block, the function dispatch table, and the data and code areas.

ENTRY `KKK`

This specifies the entry point of the function whose number is given by the first operand. The second operand specifies how many arguments the function takes and the third specified how many local variables the function may use. Calls to this function must have the required number of arguments pushed onto the stack, and on return this number of values will be automatically popped from the stack. Functions called directly from BCPL using `mcCall` always take three arguments, but functions called using the CALL instruction can take any number of arguments.

INC `R A V G M L D DX DXs DXsB`

Increment the specified register or word of memory by one, leaving the condition code undefined.

JEQ `JS JL JR`

Jump to the specified location if the first operand of a previous `CMP` instruction was equal to its second operand.

JGE `JS JL JR`

Jump to the specified location if the first operand of a previous `CMP` instruction was greater than or equal to its second operand using signed arithmetic.

JGT `JS JL JR`

Jump to the specified location if the first operand of a previous `CMP` instruction was greater than its second operand using signed arithemetic.

JLE `JS JL JR`

Jump to the specified location if the first operand of a previous `CMP` instruction was less than or equal to its second operand using signed arithmetic.

JLT `JS JL JR`

Jump to the specified location if the first operand of a previous `CMP` instruction was less than its second operand using signed arithmetic.

JMP `JS JL JR`

Unconditionally jump to the specified location.

JNE `JS JL JR`

Jump to the specified location if the first operand of a previous `CMP` instruction was not equal to its second operand.

LAB `L`

Set the specified label to the machine address of the current position in the code area.

MV `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Move the second operand into the first.

MVB `AR VR GR MR LR DR DXR DXsR DXsBR`
`AK VK GK MK LK DK DXK DXsK DXsBK`

Move the least significant byte of the second operand into the memory byte location specified by the first.

MVH `AR VR GR MR LR DR DXR DXsR DXsBR`
`AK VK GK MK LK DK DXK DXsK DXsBK`

Move the least significant 16 bits of the second operand into the 16-bit memory location specified by the first.

MVSXB `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Move the sign extended byte value specified by the second operand into the first.

MVSXH `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Move the sign extended 16-bit value specified by the second operand into the first.

MVZXB `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Move the zero extended byte value specified by the second operand into the first.

MVZXH `RA RV RG RM RL RD RDX RDXs RDXsB`
`RR AR VR GR MR LR DR DXR DXsR DXsBR`
`RK AK VK GK MK LK DK DXK DXsK DXsBK`

Move the zero extended 16-bit value specified by the second operand into the first.

LEA `RA RV RG RM RL RD RDX RDXs RDXsB`

Load the register specified by the first operand with the absolute address of the memory location specified by the second operand.

LSH `RK RR`

Shift to the left the value in the register specified by the first operand by the amount specified by the second operand. If the second operand is a register is must be `C`. Vacated positions are filled with zeros. The effect is undefined if the shift distance is not in the range 0 to 31.

MUL `K R A V G M L D DX DXs DXsB`

Multiply register `A` by the operand placing the double length result in `D:A`. Signed arithmetic is used. Unsigned arithmetic is used. Immediate (`K`) operands may sometimes be packed in the data area.

NEG `R A V G M L D DX DXs DXsB`
Negate the value specified by the operand.

NOP `F`
Performs no operation.

NOT `R A V G M L D DX DXs DXsB`
Perform the bitwise complement of the value specified by the operand.

OR
```
   RA  RV  RG  RM  RL  RD  RDX  RDXs  RDXsB
RR AR  VR  GR  MR  LR  DR  DXR  DXsR  DXsBR
RK AK  VK  GK  MK  LK  DK  DXK  DXsK  DXsBK
```
Perform the bitwise OR of the second operand into the first.

POP `R A V G M L D DX DXs DXsB`
Pop one word off the stack placing it in the specified register or memory location.

PUSH `K R A V G M L D DX DXs DXsB`
Push the specified constant, register or memory location onto the stack.

RSH `RR RK`
Shift to the right the value in the register specified by the first operand by the amount specified by the second operand. If the second operand is a register is must be `C`. Vacated positions are filled with zeros. The effect is undefined if the shift distance is not in the range 0 to 31.

RTN `F`
This causes a return from the current function. The result, if any, should be in `A`.

SEQ `R`
Set the specified register to one if the first operand of a previous `CMP` instruction was equal to its second operand, otherwise set it to zero.

SGE `R`
Set the specified register to one if the first operand of a previous `CMP` instruction was greater than or equal to its second operand using signed arithmetic, otherwise set it to zero.

SGT `R`
Set the specified register to one if the first operand of a previous `CMP` instruction was greater than its second operand using signed arithmetic, otherwise set it to zero.

SLE `R`
Set the specified register to one if the first operand of a previous `CMP` instruction

was less than or equal to its second operand using signed arithmetic, otherwise set it to zero.

SLT R

Set the specified register to one if the first operand of a previous CMP instruction was less than its second operand using signed arithmetic, otherwise set it to zero.

SNE R

Set the specified register to one if the first operand of a previous CMP instruction was not equal to its second operand, otherwise set it to zero.

SUB RA RV RG RM RL RD RDX RDXs RDXsB
RR AR VR GR MR LR DR DXR DXsR DXsBR
RK AK VK GK MK LK DK DXK DXsK DXsBK

Subtract the second operand from the first, and set the condition code appropriately.

SUBC RA RV RG RM RL RD RDX RDXs RDXsB
RR AR VR GR MR LR DR DXR DXsR DXsBR
RK AK VK GK MK LK DK DXK DXsK DXsBK

Subtract the condition code carry bit and the second operand from the first, and set the condition code appropriately. Subtracting 1 from the 64-bit value held in B:A can be done by the code generated by:

```
mcRK( mc_sub,  mc_a, 1)  // Don't use DEC here!!
mcRK( mc_subc, mc_b, 0)
```

UDIV K R A V G M L D DX DXs DXsB

Divide the double length value in D:A by the specified operand. The result is left in A and the remainder in D. The UDIV instruction performs unsigned arithmetic.

UJGE JS JL JR

Jump to the specified location if the first operand of a previous CMP instruction was greater than or equal to its second operand using unsigned arithmetic.

UJGT JS JL JR

Jump to the specified location if the first operand of a previous CMP instruction was greater than its second operand using unsigned arithmetic.

UJLE JS JL JR

Jump to the specified location if the first operand of a previous CMP instruction was less than or equal to its second operand using unsigned arithmetic.

UJLT JS JL JR

Jump to the specified location if the first operand of a previous CMP instruction was less than its second operand using unsigned arithmetic.

UMUL `K R A V G M L D DX DXs DXsB`

Multiply register `A` by the operand placing the double length result in `D:A`. Unsigned arithmetic is used. Immediate (`K`) operands may sometimes be packed in the data area.

USGE `R`

Set the specified register to one if the first operand of a previous `CMP` instruction was greater than or equal to its second operand using unsigned arithmetic, otherwise set it to zero.

USGT `R`

Set the specified register or memory word to one if the first operand of a previous `CMP` instruction was greater than its second operand using unsigned arithmetic, otherwise set it to zero.

USLE `R`

Set the specified register to one if the first operand of a previous `CMP` instruction was less than or equal to its second operand using unsigned arithmetic, otherwise set it to zero.

USLT `R`

Set the specified register to one if the first operand of a previous `CMP` instruction was less than its second operand using unsigned arithmetic, otherwise set it to zero.

XCHG `RR RA RV RG RM RL RD RDX RDXs RDXsB`

Exchange the values specified by the two operands.

XOR
```
   RA RV RG RM RL RD RDX RDXs RDXsB
RR AR VR GR MR LR DR DXR DXsR DXsBR
RK AK VK GK MK LK DK DXK DXsK DXsBK
```
Exclusive OR the second operand into the first.

10.4 MC Debugging Aids

The primary debugging aid is to inspect the generated code and the is controlled by the DEBUG directive which sets the tracing level held in the global variable `mcDebug`. Assuming *bimc* are the least significant four bit of `mcDebug`, if $c = 1$, print comments compiled by `mcComment`. If $m = 1$, print MC instructions and directives. If $i = 1$, print the corresponding target instruction(s) and if $b = 1$, print the resulting binary code in hexadecimal. To fully understand this output it is, of course, necessary to have a good understanding of the target architecture being used.

A second important debugging aid is provided by the `mcPRF` function which compiler code to output the value of a specified register using a given `printf` format string. On return all registers including the condition code are preserved. A typical call of `mcPRF` is as follows.

```
mcPRF("The value of register A is %8x*n", mc_a)
```

As an aid to debugging MC packages themselves, there is a test program called `bcplprogs/mc/mcsystest.b` which systematically tests all MC instructions, directives and addressing modes generating error messages for each error found. Each such error message includes a test number which helps to locate the source of the of the problem. If `mcsystest` is given a test number as argument, it provides a detailed compilation trace of the specified test. This should provide sufficient information to locate the error in the package.

10.5 The n-queens demonstration

This section shows how the algorithm to solve the n-queens problem as described in Section 12.3 on page 160 can be reimplemented using the MC package. The MC version of the program is as follows.

```
GET "libhdr"
GET "mc.h"

MANIFEST {
 // Register mnemonics
 ld    = mc_a
 row   = mc_b
 rd    = mc_c
 poss  = mc_d
 p     = mc_e
 count = mc_f
}

LET start() = VALOF
{ // Load the dynamic code generation package
  LET mcseg = globin(loadseg("mci386"))
  LET mcb = 0

  UNLESS mcseg DO
  { writef("Trouble with MC package: mci386*n")
    GOTO fin
  }

  // Create an MC instance for 20 functions with a data space
  // of 100 words and code space of 4000
  mcb := mcInit(20, 100, 40000)

  UNLESS mcb DO
  { writef("Unable to create an mci386 instance*n")
    GOTO fin
  }

  mc := 0          // Currently no selected MC instance
  mcSelect(mcb)

  mcK(mc_debug, dlevel)

  FOR n = 1 TO 16 DO
  { mcComment("*n*n// Code for a %nx%n board*n", n, n)
```

```
    gencode(n) // Compile the code for an nxn board
  }

  mcF(mc_end)

  writef("Code generation complete*n")

  FOR n = 1 TO 16 DO
    writef("Number of solutions to %i2-queens is %i9*n",
            n, mcCall(n))

fin:
  IF mc DO mcClose()
  IF mcseg DO unloadseg(mcseg)

  writef("*n*nEnd of run*n")
}

AND gencode(n) BE
{ LET all = (1<<n) - 1
  mcKKK(mc_entry, n, 0, 0)

  try(1, n, all)

  mcRR(mc_mv, mc_a, count)       // return count
  mcF(mc_rtn)
  mcF(mc_endfn)
}

AND try(i, n, all) BE
{ LET L = mcNextlab()

  mcComment("*n// Start of code from try(%n, %n, %n)*n", i, n, all)

  IF i=1 DO
  { mcRK(mc_mv, ld,    0)        // At the outermost level
    mcRK(mc_mv, row,   0)        // initialise ld, row, rd and count
    mcRK(mc_mv, rd,    0)
    mcRK(mc_mv, count, 0)
  }

  mcRR(mc_mv,  poss, ld)         // LET poss = (~(ld | row | rd)) & all
  mcRR(mc_or,  poss, row)
  mcRR(mc_or,  poss, rd)
  mcR (mc_not, poss)
  mcRK(mc_and, poss, all)

  mcRK(mc_cmp, poss, 0)          // IF poss DO
  mcJL(mc_jeq, L)

  TEST i=n
  THEN { // We are on the final row and can place a queen
         mcR(mc_inc,  count)     //   count := count+1
       }
  ELSE { // We are not on the final row.
         LET M = mcNextlab()
```

```
          mcL (mc_lab,  M)        // { Start of REPEATWHILE loop

          mcRR(mc_mv,   p, poss)  //   LET p = poss & -poss
          mcR (mc_neg,  p)
          mcRR(mc_and,  p, poss)
          mcRR(mc_sub,  poss, p)  //   poss := poss - p


          mcR (mc_push, ld)       // call try((ld+p)<<1, row+p, (rd+p)>>1)
          mcR (mc_push, row)
          mcR (mc_push, rd)
          mcR (mc_push, poss)

          mcRR(mc_add,  ld,  p)
          mcRK(mc_lsh,  ld,  1)   //   ld  := (ld+p)<<1
          mcRR(mc_add,  row, p)   //   row := row+p
          mcRR(mc_add,  rd,  p)
          mcRK(mc_rsh,  rd,  1)   //   rd  := (rd+p)>>1

          try(i+1, n, all)

          mcR (mc_pop,  poss)
          mcR (mc_pop,  rd)
          mcR (mc_pop,  row)
          mcR (mc_pop,  ld)

          mcRK(mc_cmp,  poss, 0)
          mcJL(mc_jne, M)         // } REPEATWHILE poss
        }

        mcL(mc_lab, L)
        mcComment("// End   of code from try(%n, %n, %n)*n*n",
                   i, n, all)
}
```

In this implementation all the working variables are held in registers and all recursive calls are unwound knowing that the depth of recursion will be limited, in this case to no more than 16. The stack is used to save the state at the moment when a recursive call would have been made in the original program. An optimisation is done based on the knowledge that if a queen can be placed on the nth row of $n \times n$ board then the solution count can be incremented.

When running on a Pentium IV this implementation executes approximately 24 times faster than the normal interpretive Cintcode version and 25% faster than the corresponding optimised C version of the algorithm.

Chapter 11

Installation

The implementation of BCPL described in this report is is available free via my Home Page [3] 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 XP there are precompiled `.exe` files such as `wincintsys.exe` and `winrastsys.exe`. These files should be copied into the appropriate `bin` directory and renamed as `cintsys.exe` and `rastsys.exe`. For all the other architectures it is necessary to rebuild the system.

The simplest installation is for Linux machines.

11.1 Linux Installation

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

1) First create a directory named `BCPL` and copy either `bcpl.tgz` or `bcpl.zip` into it. These are available (free) via my home page [3] and both contain the same set of packed files and directories.

2) Enter the BCPL directory and extract the files of the BCPL Cintcode System by:

```
cd BCPL
tar zxvf bcpl.tgz
```

or unpack `bcpl.zip` using:

```
cd BCPL
unzip -v bcpl.zip
```

Some web browsers will have already decompressed the .tgz file, so you may have use the following command instead:

```
cd BCPL
tar xvf bcpl.tgz
```

This step 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).

3) In order to use the BCPL Cintcode system from another directory it is necessary to define the shell the variables `BCPLROOT`, `BCPLPATH` and `BCPLHDRS`. These must specify the absolute file names of the BCPL root directory, the directory containing the compiles commands and the directory containing the BCPL header files. The `BCPLROOT` directory should also be added to your `PATH`. This can be done by editing the file `BCPL/cintcode/setbcplenv`, if necessary, and running the command:

```
. setbcplenv          under bash
```

or

```
source setbcplenv     under the C-shell
```

This will execute commands similar to:

```
export BCPLROOT=$HOME/distribution/BCPL/cintcode
export BCPLPATH=$BCPLROOT/cin
export BCPLHDRS=$BCPLROOT/g
export PATH=$PATH:$BCPLROOT/bin
```

or

```
setenv BCPLROOT ${HOME}/distribution/BCPL/cintcode
setenv BCPLPATH ${BCPLROOT}/cin
setenv BCPLHDRS ${BCPLROOT}/g
setenv PATH     ${PATH}:${BCPLROOT}/bin
```

4) Now change directory to `cintcode` and attempt to re-build and enter the BCPL system:

```
cd cintcode
make clean
make
```

The line `make clean` is a recent addition to eliminate some commonly reported problems. There is something wrong if the output of the above make command does not end with something like:

```
...
bin/cintsys

BCPL Cintcode System (25 Jan 2007)
0>
```

To troubleshoot, try typing the following lines to a shell prompt:

```
cintsys -f -v
```

or

```
cintsys -f -V
```

and study the output, in conjunction with `sysc/cintsys.c` and `sysb/boot.b`. Hopefully, there will be enough information there to diagnose and correct the problem.

5) Now recompile all the system software and commands. This is done by typing:

```
        c compall
```

6) Next, 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/sysb` 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
```

7) 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`. The current version of this BCPL manual is available from my home page as a `.pdf` file. There is a demonstration script of commands in `cintcode/doc/notes`.

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

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

11.2 Command Line Arguments

The commands `cintsys` and `cintpos` that invoke the Cintcode interpreter can be given various arguments. These are:

`-m` *n*	Set the cintcode memory size to *n* words.
`-t` *n*	Set the tally vector size to *n* words.
`-s`	Enter the cintcode system giving the name of this file as the command for the CLI to run.
`-c` *text*	Enter cintsys with standard input setup to read the characters from *text* followed by an end-of-stream character.
`--` *text*	Enter cintsys with standard input setup to read the characters in text followed by the characters of the old standard input.
`-f`	Trace the use of environment variables in pathinput
`-v`	Trace the bootstrapping process
`-V`	As -v, but also include some Cincode level tracing
`-h`	Output some help information.

The rastering version of the interpreter `rastsys` can receive the same arguments.

11.3 Installation on Other Machines

Carry out steps 1 to 4 above. In the directory `BCPL/cintcode/sys` you will find directories for different architectures, e.g. ALPHA, MIPS, SUN4, SPARC, MSDOS, MAC, OS2, BC4, Win32, CYGWIN32 and shWinCE. These contain files that are architecture (or compiler) dependent, typically including `cintasm.s` (or `cintasm.asm`). For some old versions of Linux, it is necessary to change `_dosys` to `dosys` (or vice-versa) in the file `sys/LINUX/cintasm.s`.

Edit Makefile (typically by adding and removing comment symbols) as necessary for your system/machine and then execute make in the cintcode directory, e.g:

```
make
```

Variants of the above should work for the other architectures running Unix.

11.4 Installation for Windows XP

The files `wincintsys.exe` and `winrastsys.exe` are included in the standard distribution and should work under many versions of the Windows operating systems (such as Windows XP) just by typing the command:

```
wincintsys
```

It may be more convenient to move them into a different directory and rename them as `cintsys.exe` and `rastsys.exe`.

I have recently upgraded the Windows version of BCPL so that it can be compiled and run using the freely available Microsoft C compiler and libraries. On a new PC I installed the freely available .NET Framework 3.5 and the corresponding SDK 3.5. This provided amongst many other things a C compiler and all the relevant libraries.

I then created a shortcut on the desktop with

```
Target: %SystemRoot%\system32\cmd.exe /q /k VC9env.bat
```

and

```
Start in: D:\distribution\BCPL\cintcode
```

Double clicking on this shortcut opens a Shell window with the required environment variable all set up C compilation and the BCPL running environment. If they are not correct you may have to edit `VC9env.bat`. The BCPL system was then rebuilt by the commands:

```
nmake /f MakefileVC clean
nmake /f MakefileVC
```

This should recompile and link all the C code of the BCPL Cintcode system and then recompile all the standard BCPL system programs and commands. For good measure, once the BCPL Cintcode system has been entered, recompile all the BCPL code again by typing:

```
c compall
```

11.5 Installation using Cygwin

I recommend using the GNU development tools and utilities for Windows 95/98/NT/XP/etc that are available from `http://sourceware.cygnus.com/cygwin/`.

Edit the `cintcode/Makefile` to comment out the LINUX version

```
        #CC = gcc -O9 -DforLINUX
        #SYSM = ../cintcode/sys/LINUX
        #CINTASM = cintasm.o
        #ENDER = LITENDER
```

and enable the CYGWIN32 version

```
        CC = gcc -O9 -DforCYGWIN32
        CINTASM = cintasm.o
        SYSM = ../cintcode/sys/CYGWIN32
        ENDER = LITENDER
```

Then type:

```
        make
```

This should recompile the system and create the executable `cintsys.exe`.

Remember to include the cintcode directory in your `PATH` and `BCPLPATH` shell variables, so that the cintcode system can be run in any directory.

Careful inspection of the Makefile and directories in `cintcode/sys` will show that versions also exist that use Microsoft C++ 5.0 and Borland C4.0, but these are likely to be out of date and their use is not recommended.

11.6 Installation for Windows CE2.0

A version of the BCPL Cintcode System is available for handheld machines running Windows CE version 2.0. For installation details see the file `cintcode/sys/shWinCE/README`. This system provides ascrollable window for interaction with the CLI. It also provides a simple graphical facilities using a graphics window. The system has only been tested on an HP 620LX handheld machine.

11.7 The Native Code Version

A BCPL native mode system for 386/486/Pentium based machines is in directory `MCPL/native`. It can be re-built and test by changing to the directory `BCPL/natbcpl` and running `make`.

A version (64 bit) for the DEC Alpha is also available. To re-build this it is necessary to comment out the lines for LINUX and uncomment the lines for the ALPHA in `Makefile`, before running `make`.

Chapter 12

Example Programs

12.1 Coins

The following program prints out how many different ways a sum of money can be composed from coins of various denominations.

```
GET "libhdr"

LET coins(sum) = c(sum, (TABLE 200, 100, 50, 20, 10, 5, 2, 1, 0))

AND c(sum, t) = sum<0 -> 0,
                sum=0 -> 1,
                !t=0  -> 0,
                c(sum, t+1) + c(sum-!t, t)

LET start() = VALOF
{ writes("Coins problem*n")
  t(0); t(1); t(2); t(5); t(21); t(100); t(200)
  RESULTIS 0
}

AND t(n) BE writef("Sum = %i3  number of ways = %i6*n", n, coins(n))
```

12.2 Primes

The following program prints out a table of all primes less than 1000, using the sieve method.

```
GET "libhdr"

GLOBAL { count: ug  }

MANIFEST { upb = 999  }

LET start() = VALOF
{ LET isprime = getvec(upb)
   count := 0
   FOR i = 2 TO upb DO isprime!i := TRUE  // Until proved otherwise.

   FOR p = 2 TO upb IF isprime!p DO
   { LET i = p*p
      UNTIL i>upb DO { isprime!i := FALSE; i := i + p }
      out(p)
   }

   writes("*nend of output*n")
   freevec(isprime)
   RESULTIS 0
}

AND out(n) BE
{ IF count REM 10 = 0 DO newline()
   writef(" %i3", n)
   count := count + 1
}
```

12.3 Queens

The following program calculates the number of ways n queens can be placed on a $n \times n$ chess board without any two occupying the same row, column or diagonal.

```
GET "libhdr"

GLOBAL { count:200; all:201  }

LET try(ld, row, rd) BE TEST row=all

                        THEN count := count + 1

                        ELSE { LET poss = all & ~(ld | row | rd)
                               UNTIL poss=0 DO
                               { LET p = poss & -poss
                                 poss := poss - p
                                 try(ld+p << 1, row+p, rd+p >> 1)
                               }
                             }
```

```
LET start() = VALOF
{ all := 1

  FOR i = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("Number of solutions to %i2-queens is %i5*n", i, count)
    all := 2*all + 1
  }

  RESULTIS 0
}
```

12.4 Fridays

The following program prints a table of how often the 13th day of the month lies on each day of the week over a 400 year period. Since there are an exact number of weeks in 4 centuries, program shows that the 13th is most of a Friday!

```
GET "libhdr"

MANIFEST { mon=0; sun=6; jan=0; feb=1; dec=11 }

LET start() = VALOF
{ LET count       = TABLE 0, 0, 0, 0, 0, 0, 0
   LET daysinmonth = TABLE 31,  ?, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31
   LET days = 0

   FOR year = 1973 TO 1973+399 DO
   { daysinmonth!feb := febdays(year)
      FOR month = jan TO dec DO
      { LET day13 = (days+12) REM 7
         count!day13 := count!day13 + 1
         days := days + daysinmonth!month
      }
   }
   FOR day = mon TO sun DO
     writef("%i3 %sdays*n",
            count!day,
            select(day,
                  "Mon","Tues","Wednes","Thurs","Fri","Sat","Sun")
           )
   RESULTIS 0
}

AND febdays(year) = year REM 400 = 0 -> 29,
                    year REM 100 = 0 -> 28,
                    year REM 4   = 0 -> 29,
                    28

AND select(n, a0, a1, a2, a3, a4, a5, a6) = n!@a0
```

12.5 Lambda Evaluator

The following program is a simple parser and evaluator for lambda expressions.

```
GET "libhdr"

MANIFEST {
// selectors
H1=0; H2; H3; H4

// Expression operators and tokens
Id=1; Num; Pos; Neg; Mul; Div;Add; Sub
Eq; Cond; Lam; Ap; Y
Lparen; Rparen; Comma; Eof
}

GLOBAL {
space:200; str; strp; strt; ch; token; lexval
}

LET lookup(bv, e) = VALOF
{ WHILE e DO { IF bv=H1!e RESULTIS H2!e
               e := H3!e
             }
  writef("Undeclared name %c*n", H2!bv)
  RESULTIS 0
}

AND eval(x, e) = VALOF SWITCHON H1!x INTO
{ DEFAULT:     writef("Bad exppression, Op=%n*n", H1!x)
               RESULTIS 0
  CASE Id:     RESULTIS lookup(H2!x, e)
  CASE Num:    RESULTIS H2!x
  CASE Pos:    RESULTIS eval(H2!x, e)
  CASE Neg:    RESULTIS - eval(H2!x, e)
  CASE Add:    RESULTIS eval(H2!x, e) + eval(H3!x, e)
  CASE Sub:    RESULTIS eval(H2!x, e) - eval(H3!x, e)
  CASE Mul:    RESULTIS eval(H2!x, e) * eval(H3!x, e)
  CASE Div:    RESULTIS eval(H2!x, e) / eval(H3!x, e)
  CASE Eq:     RESULTIS eval(H2!x, e) = eval(H3!x, e)
  CASE Cond:   RESULTIS eval(H2!x, e) -> eval(H3!x, e), eval(H4!x, e)
  CASE Lam:    RESULTIS mk3(H2!x, H3!x, e)

  CASE Ap:   { LET f, a = eval(H2!x, e), eval(H3!x, e)
               LET bv, body, env = H1!f, H2!f, H3!f
               RESULTIS eval(body, mk3(bv, a, env))
             }
  CASE Y:    { LET bigf   = eval(H2!x, e)
               // bigf should be a closure whose body is an
               // abstraction eg Lf Ln n=0 -> 1, n*f(n-1)
               LET bv, body, env = H1!bigf, H2!bigf, H3!bigf
               // Make a closure with a missing environment
               LET yf  = mk3(H2!body, H3!body, ?)
               // Make a new environment including an item for bv
               LET ne  = mk3(bv, yf, env)
               H3!yf := ne // Now fill in the environment component
               RESULTIS yf // and return the closure
             }
}
```

```
// ***************  Syntax analyser **********************

// Construct       Corresponding Tree

// a ,.., z   -->  [Id, 'a'] ,..,  [Id, 'z']
// dddd       -->  [Num, dddd]
// x y        -->  [Ap, x, y]
// Y x        -->  [Y, x]
// x * y      -->  [Times, x, y]
// x / y      -->  [Div, x, y]
// x + y      -->  [Plus, x, y]
// x - y      -->  [Minus, x, y]
// x = y      -->  [Eq, x, y]
// b -> x, y  -->  [Cond, b, x, y]
// Li y       -->  [Lam, i, y]

LET mk1(x) = VALOF { space := space-1; !space := x; RESULTIS space }

AND mk2(x,y) = VALOF { mk1(y); RESULTIS mk1(x)  }

AND mk3(x,y,z) = VALOF { mk2(y,z); RESULTIS mk1(x)  }

AND mk4(x,y,z,t) = VALOF { mk3(y,z,t); RESULTIS mk1(x)  }

AND rch() BE
{ ch := Eof
  IF strp>=strt RETURN
  strp := strp+1
  ch := str%strp
}

AND parse(s) = VALOF
{ str, strp, strt := s, 0, s%0
  rch()
  RESULTIS nexp(0)
}
```

```
AND lex() BE SWITCHON ch INTO
{ DEFAULT:   writef("Bad ch in lex: %c*n", ch)
  CASE Eof:  token := Eof
             RETURN
  CASE ' ':
  CASE '*n' :rch(); lex(); RETURN

  CASE 'a':CASE 'b':CASE 'c':CASE 'd':CASE 'e':
  CASE 'f':CASE 'g':CASE 'h':CASE 'i':CASE 'j':
  CASE 'k':CASE 'l':CASE 'm':CASE 'n':CASE 'o':
  CASE 'p':CASE 'q':CASE 'r':CASE 's':CASE 't':
  CASE 'u':CASE 'v':CASE 'w':CASE 'x':CASE 'y':
  CASE 'z':
             token := Id; lexval := ch; rch(); RETURN

  CASE '0':CASE '1':CASE '2':CASE '3':CASE '4':
  CASE '5':CASE '6':CASE '7':CASE '8':CASE '9':
             token, lexval := Num, 0
             WHILE '0'<=ch<='9' DO
             { lexval := 10*lexval + ch - '0'
               rch()
             }
             RETURN

  CASE '-':  rch()
             IF ch='>' DO { token := Cond; rch(); RETURN }
             token := Sub
             RETURN
  CASE '+':  token := Add;    rch(); RETURN
  CASE '(':  token := Lparen; rch(); RETURN
  CASE ')':  token := Rparen; rch(); RETURN
  CASE '**': token := Mul;    rch(); RETURN
  CASE '/':  token := Div;    rch(); RETURN
  CASE 'L':  token := Lam;    rch(); RETURN
  CASE 'Y':  token := Y;      rch(); RETURN
  CASE '=':  token := Eq;     rch(); RETURN
  CASE ',':  token := Comma;  rch(); RETURN
}
```

```
AND prim() = VALOF
{ LET a = TABLE Num, 0
  SWITCHON token INTO
  { DEFAULT:     writef("Bad expression*n");    ENDCASE
    CASE Id:     a := mk2(Id, lexval);          ENDCASE
    CASE Num:    a := mk2(Num, lexval);         ENDCASE
    CASE Y:      RESULTIS mk2(Y, nexp(6))
    CASE Lam:    lex()
                 UNLESS token=Id DO writes("Id expected*n")
                 a := lexval
                 RESULTIS mk3(Lam, a, nexp(0))
    CASE Lparen: a := nexp(0)
                 UNLESS token=Rparen DO writef("')' expected*n")
                 lex()
                 RESULTIS a
    CASE Add:    RESULTIS mk2(Pos, nexp(3))
    CASE Sub:    RESULTIS mk2(Neg, nexp(3))
  }
  lex()
  RESULTIS a
}

AND nexp(n) = VALOF { lex(); RESULTIS exp(n) }

AND exp(n) = VALOF
{ LET a, b = prim(), ?

  { SWITCHON token INTO
    { DEFAULT:  BREAK
      CASE Lparen:
      CASE Num:
      CASE Id:   UNLESS n<6 BREAK
                 a := mk3(Ap,  a, exp(6));  LOOP
      CASE Mul:  UNLESS n<5 BREAK
                 a := mk3(Mul, a, nexp(5)); LOOP
      CASE Div:  UNLESS n<5 BREAK
                 a := mk3(Div, a, nexp(5)); LOOP
      CASE Add:  UNLESS n<4 BREAK
                 a := mk3(Add, a, nexp(4)); LOOP
      CASE Sub:  UNLESS n<4 BREAK
                 a := mk3(Sub, a, nexp(4)); LOOP
      CASE Eq:   UNLESS n<3 BREAK
                 a := mk3(Eq,  a, nexp(3)); LOOP
      CASE Cond: UNLESS n<1 BREAK
                 b := nexp(0)
                 UNLESS token=Comma DO writes("Comma expected*n")
                 a := mk4(Cond, a, b, nexp(0)); LOOP
    }
  } REPEAT
  RESULTIS a
}
```

```
AND try(expr) BE
{ LET v = VEC 2000
  space := v+2000
  writef("Trying %s*n", expr)
  writef("Answer: %n*n", eval(parse(expr), 0))
}

AND start() = VALOF
{ try("(Lx x+1) 2")
  try("(Lx x) (Ly y) 99")
  try("(Ls Lk s k k) (Lf Lg Lx  f x (g x)) (Lx Ly x) (Lx x) 1234")
  try("(Y (Lf Ln n=0->1,n**f(n-1))) 5")
  RESULTIS 0
}
```

12.6 Fast Fourier Transform

The following program is a simple demonstration of the algorithm for the fast fourier transform. Instead of using complex numbers, it uses integer arithmetic modulo 65537 with an appropriate Nth root of unity.

```
GET "libhdr"

MANIFEST {
modulus = #x10001  // 2**16 + 1

$$ln10  // Set condition compilation flag to select data size
//$$walsh

$<ln16 omega = #x00003; ln = 16 $>ln16  // omega**(2**16) = 1
$<ln12 omega = #x0ADF3; ln = 12 $>ln12  // omega**(2**12) = 1
$<ln10 omega = #x096ED; ln = 10 $>ln10  // omega**(2**10) = 1
$<ln4  omega = #x08000; ln = 4  $>ln4   // omega**(2**4)  = 1
$<ln3  omega = #x0FFF1; ln = 3  $>ln3   // omega**(2**3)  = 1

$<walsh  omega=1   $>walsh    // The Walsh transform

N       = 1<<ln    // N is a power of 2
upb     = N-1
}

STATIC   { data=0  }
```

```
LET start() = VALOF
{  writef("fft with N = %n and omega = %n modulus = %n*n*n",
                    N,          omega,     modulus)

   data := getvec(upb)

   UNLESS omega=1 DO   // Unless doing Walsh tranform
     check(omega, N)   //   check that omega and N are consistent

   FOR i = 0 TO upb DO data!i := i
   pr(data, 7)
// prints  -- Original data
//     0     1     2     3     4     5     6     7

   fft(data, ln, omega)
   pr(data, 7)
// prints   -- Transformed data
// 65017 26645 38448 37467 30114 19936 15550 42679

   fft(data, ln, ovr(1,omega))
   FOR i = 0 TO upb DO data!i := ovr(data!i, N)
   pr(data, 7)
// prints  -- Restored data
//     0     1     2     3     4     5     6     7
   RESULTIS 0
}

AND fft(v, ln, w) BE  // ln = log2 n    w = nth root of unity
{ LET n = 1<<ln
  LET vn = v+n
  LET n2 = n>>1

  // First do the perfect shuffle
  reorder(v, n)

  // Then do all the butterfly operations
  FOR s = 1 TO ln DO
  { LET m = 1<<s
    LET m2 = m>>1
    LET wk, wkfac = 1, w
    FOR i = s+1 TO ln DO wkfac := mul(wkfac, wkfac)
    FOR j = 0 TO m2-1 DO
    { LET p = v+j
      WHILE p<vn DO { butterfly(p, p+m2, wk); p := p+m }
      wk := mul(wk, wkfac)
    }
  }
}

AND butterfly(p, q, wk) BE { LET a, b = !p, mul(!q, wk)
                             !p, !q := add(a, b), sub(a, b)
                           }
```

```
AND reorder(v, n) BE
{ LET j = 0
  FOR i = 0 TO n-2 DO
  { LET k = n>>1
    // j is i with its bits is reverse order
    IF i<j DO { LET t = v!j; v!j := v!i; v!i := t }
    // k  =  100..00       10..0000..00
    // j  =  0xx..xx       11..10xx..xx
    // j' =  1xx..xx       00..01xx..xx
    // k' =  100..00       00..0100..00
    WHILE k<=j DO { j := j-k; k := k>>1 } //) "increment" j
    j := j+k                              //)
  }
}

AND check(w, n) BE
{ // Check that w is a principal nth root of unity
  LET x = 1
  FOR i = 1 TO n-1 DO { x := mul(x, w)
                        IF x=1 DO writef("omega****%n = 1*n", i)
                      }
  UNLESS mul(x, w)=1 DO writef("Bad omega**%n should be  1*n", n)
}

AND pr(v, max) BE
{ FOR i = 0 TO max DO { writef("%I5 ", v!i)
                        IF i REM 8 = 7 DO newline()
                      }
  newline()
}

AND dv(a, m, b, n) = a=1 -> m,
                     a=0 -> m-n,
                     a<b -> dv(a, m, b REM a, m*(b/a)+n),
                     dv(a REM b, m+n*(a/b), b, n)


AND inv(x) = dv(x, 1, modulus-x, 1)

AND add(x, y) = VALOF
{ LET a = x+y
  IF a<modulus RESULTIS a
  RESULTIS a-modulus
}

AND sub(x, y) = add(x, neg(y))

AND neg(x)    = modulus-x

AND mul(x, y) = x=0 -> 0,
                (x&1)=0 -> mul(x>>1, add(y,y)),
                add(y, mul(x>>1, add(y,y)))

AND ovr(x, y) = mul(x, inv(y))
```

Bibliography

[1] D.T. Ross et al. AED-0 programmer's guide and user kit. Technical report, Electronic Systems Laboratory M.I.T, 1964.

[2] C. Jobson and J.M. Richards. *BCPL for the BBC Microcomputer*. Acornsoft Ltd, Cambridge, 1983.

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

[4] M. Richards. *The Implementation of CPL-like programming languages*. Phd thesis, Cambridge University, 1966.

[5] M. Richards, A.R. Aylward, P. Bond, R.D. Evans, and B.J. Knight. The Tripos Portable Operating System for Minicomputers. *Software-Practice and Experience*, 9:513–527, June 1979.

[6] Christopher Strachey. A General Purpose Macrogenerator. *Computer Journal*, 8(3):225–241, 1965.

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: $-\boxed{\texttt{program}}-$, $-\boxed{\texttt{section}}-$, $-\boxed{\texttt{D}}-$, $-\boxed{\texttt{C}}-$ and $-\boxed{\texttt{En}}-$, 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 $-\boxed{\texttt{WHILE}}-$ and $-\boxed{\texttt{:=}}-$, it must match the next input token for the test to succeed. The test box $-\boxed{\texttt{eof}}-$ is only satisfied if the end of file has been reached. Sometimes the test box contains a side condition, as in $-\boxed{\texttt{REM n<6}}-$, in which case the side condition must also be satisfied. The only other test boxes are $-\boxed{\texttt{is call}}-$ and $-\boxed{\texttt{is name}}-$ which are only satisfied 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 $-\boxed{\texttt{En}}-$, we obtain the definitions of $-\boxed{\texttt{E0}}-$ to $-\boxed{\texttt{E8}}-$. Starting from the definition of $-\boxed{\texttt{program}}-$, 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.

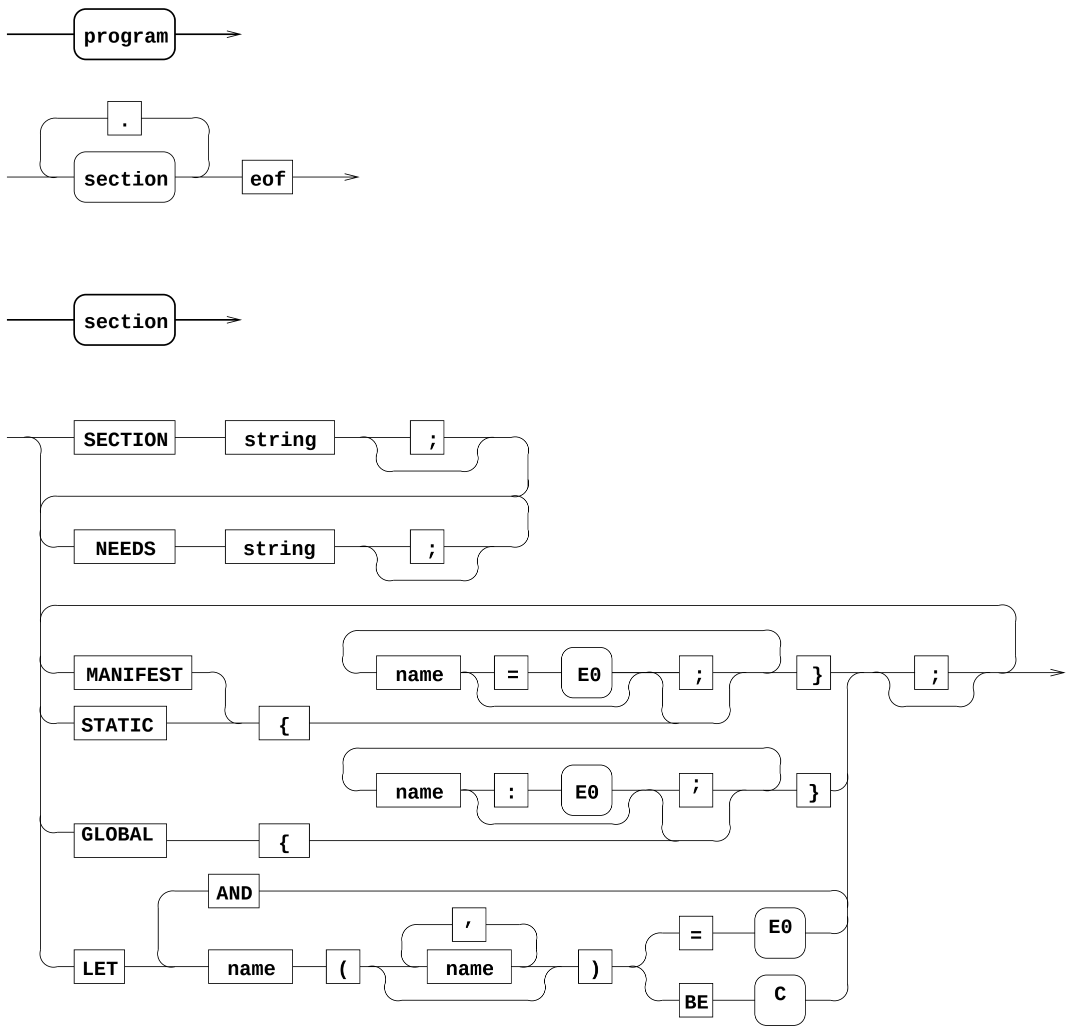

Figure A.1: Program, Section

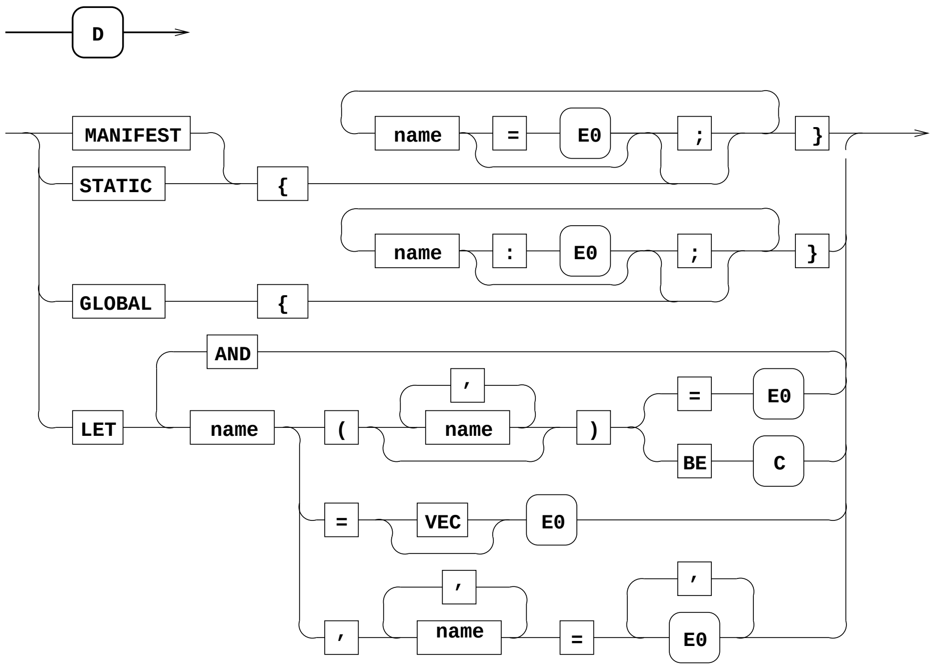

Figure A.2: Declarations

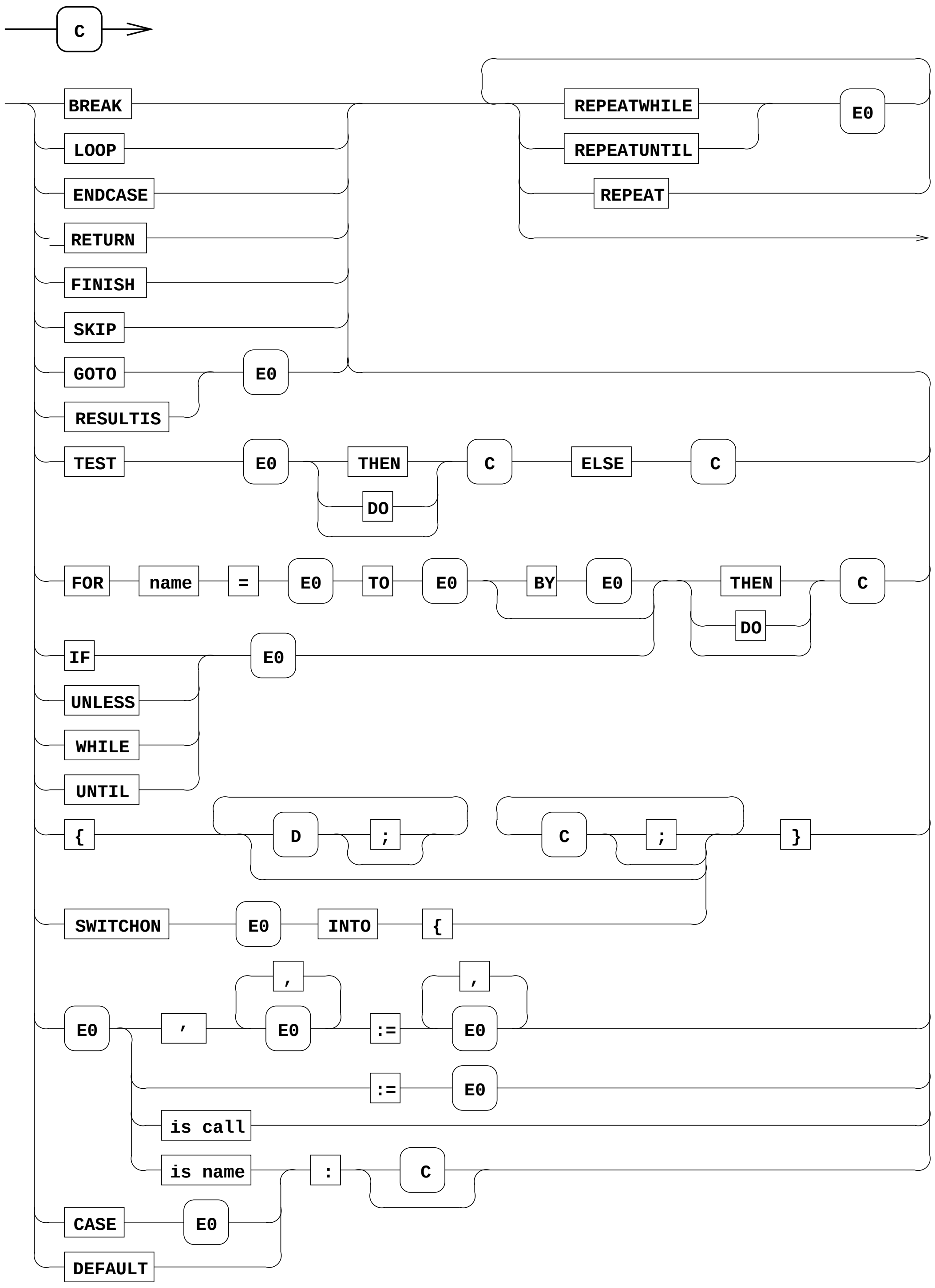

Figure A.3: Commands

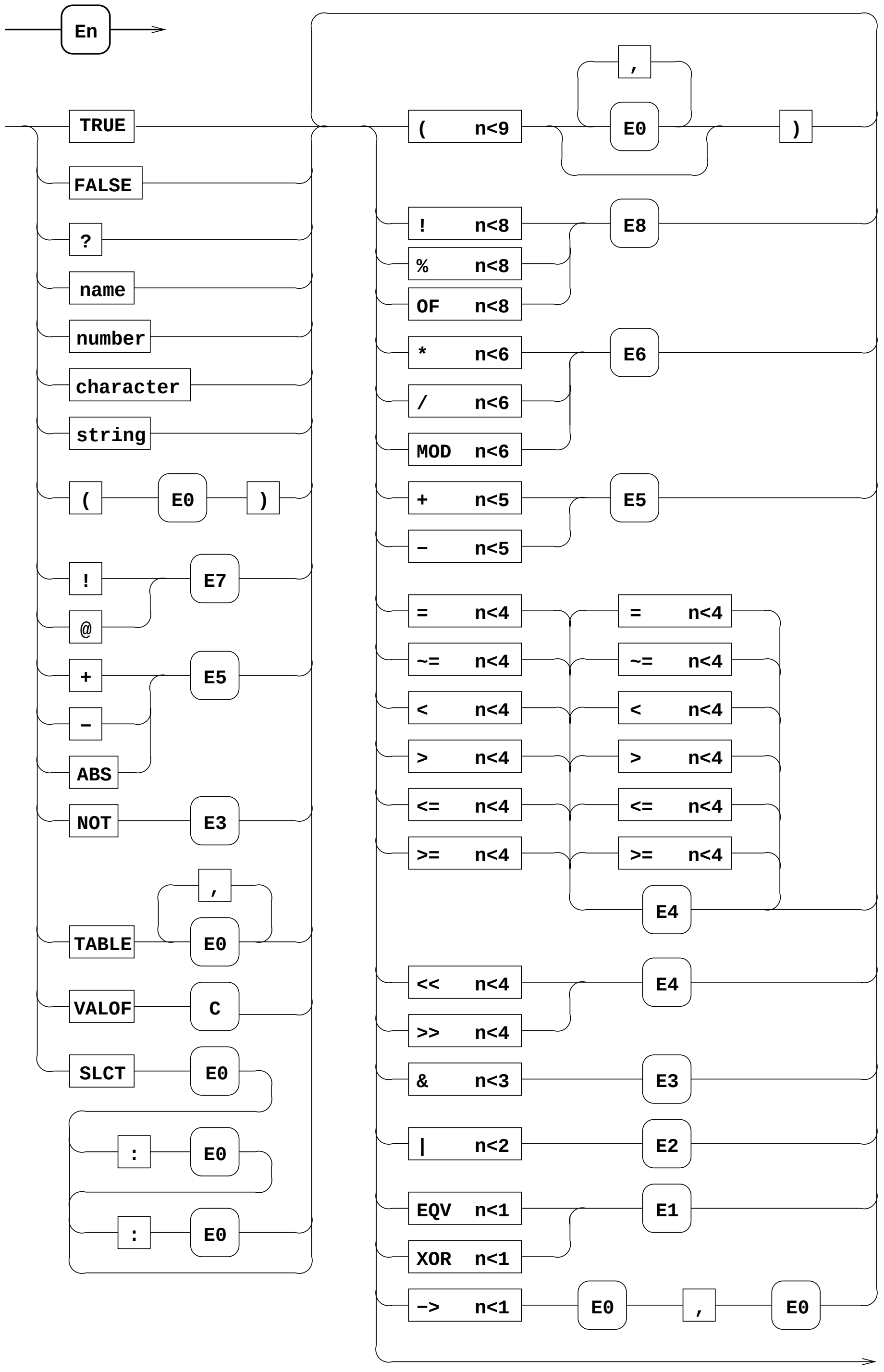

Figure A.4: Expressions