The BCPL Cintsys and Cintpos User Guide

by

Martin Richards

mr10@cl.cam.ac.uk

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

Computer Laboratory
University of Cambridge
Revision date: Mon Sep 22 09:28:49 AM BST 2025

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 both the single threaded BCPL Cintcode System (called Cintsys) and the Cintcode version of the Tripos portable operating system (called Cintpos). It gives the definition of standard BCPL including the recently added features such as floating point expressions and constructs involving operators such as `<>` and *op*`:=`. The language has recently been extended to include some of the pattern matching features of MCPL. This manual also describes the standard library and running environment and the native code version of the system based on Sial. Installation instructions are included. Since May 2013, the standard BCPL distribution supports both 32 and 64 bit Cintcode versions. Since August 2014, standard Cintcode BCPL includes floating point constants and operators, and since March 2018 it includes the FLT feature to make it easier to perform floating point calculations. Pattern matching was added in September 2021. These extensions are now in the standard BCPL distribution.

Keywords: Systems programming language, BCPL, Cintcode, 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. Recently I have decided to augment BCPL with some of the features of MCPL including particularly the pattern matching mechanism used in the definition of functions.

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, including, for instance, the Raspberry Pi.
- The MC package that allows machine independent dynamic compilation and execution of native machine code.

For many example BCPL programs see `bcpl4raspi.pdf` available from my home page.

MR

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, two example console sessions are presented to exhibit some of the key features of both the single threaded version of the system (Cintsys) and the interpretive version of Tripos (Cintpos).

1.1 A Cintsys Console Session

The BCPL Cintcode system can be entered using the `cintsys` shell command under the host operating system. If `cintsys` is called with the `-h` option it will output the following information about other possible options.

```
Valid arguments:

-h            Output this help information
-m n          Set Cintcode memory size to n words
-t n          Set Tally vector size to n words
-g n          Set the default global vector upb to n
-q            Set quiet mode. This stops the resident system
              fromm outputing text other than error messages and
              and debugging aids. It also stops the CLI from
              outputting prompts and stops the echoing of
              standar input (normally the keyboard).
-c args       Pass args to interpreter as CLI input
-- args       Pass args to interpreter as CLI input,
              then re-attach stdin
-s file args  Invoke the interpreter with this file as CLI input
-cin name     Set the pathvar environment variable name
-f            Trace use of environment variables in pathinput
-v            Trace the bootstrapping process
-vv           As -v, but include some Cincode level tracing
```

```
-d            Cause a dump of the Cintcode memory to DUMP.mem
              if a fault/error is encountered
-slow         Force the slow interpreter to always be selected
```

The Cintsys system is normally started using the `cintsys` shell command. This demonstration was run when I was logged in as user `mr` on a machine called `Cobham`. The BCPL Cintcode system was already properly installed. The demonstration was run in the root directory of the BCPL Cintcode system as was entered as follows.

```
mr@Cobham~$ cd $BCPLROOT
mr@Cobham~/distribution/BCPL/cintcode$
mr@Cobham~/distribution/BCPL/cintcode$ cintsys

BCPL 32-bit Cintcode System (18 Jul 2022)
0.000>
```

The characters `0.000>` are followed by a space character and is the command language prompt string inviting the user to type a command. The number gives the execution time in seconds of the preceding command. A program called `fact.b` in directory `cintcode/com` to compute factorials can be displayed using the `type` command as follows:

```
0.000> type com/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)
0.000>
```

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 function 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 a call of the library function `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` and `start` to be defined simultaneously allow `start` to call `fact`. This program can be compiled by using the following command:

```
0.000> bcpl com/fact.b to fact

32 bit BCPL (18 Jul 2022) with pattern matching, 32 bit target
Code size =   104 bytes 0f 32-bit little ender Cintcode
0.034>
```

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.

```
0.034> fact
fact(1) =    1
fact(2) =    2
fact(3) =    6
fact(4) =   24
fact(5) =  120
0.006>
```

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:

```
0.010> bcpl com/fact.b to fact d1

BCPL (3 Sep 2019) 32 bit with the FLT feature
   0:  DATAW 0x00000000
   4:  DATAW 0x0000DFDF
   8:  DATAW 0x6174730B
  12:  DATAW 0x20207472
  16:  DATAW 0x20202020
// Entry to:   start
  20: L10:
  20:     L1
  21:    SP3
  22: L12:
```

```
  22:    LP3
  23:     LF  L2
  25:     K9
  26:    SP9
  27:    LP3
  28:    SP8
  29:    LLL  L19920
  31:    K4G   94
  33: L15:
  33:     L1
  34:    AP3
  35:    SP3
  36:     L5
  37:    JLE  L12
  39: L14:
  39: L13:
  39:     L0
  40:    RTN
  44: L19920:
  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: L11:
  76:   JNE0  L16
  78:     L1
  79:    RTN
  80: L16:
  80:    LM1
  81:    AP3
  82:     LF  L11
  84:     K4
  85:    LP3
  86:    MUL
  87:    RTN
  88:  DATAW 0x00000000
  92:  DATAW 0x00000001
  96:  DATAW 0x00000014
 100:  DATAW 0x0000005E
Code size = 104 bytes 0f 32-bit little ender Cintcode
0.050>
```

This output shows the sequence of Cintcode instructions compiled for the functions `start` and `fact`. In addition 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 function name placed just before the entry point. As can be seen the name 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 opera-

tions on the registers and memory of the Cintcode machine. For instance, the first two instructions of `start` (`L1` and `SP3` at locations 20 and 21) 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 `L12` corresponding to location 22. The compilation of `fact(i)` is `LP3 LF L11 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 L19920`. 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 L12` jumps to label L12 if `i` is less than or equal to `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 L10`). 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 L11 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.

```
0.030> 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
0.006>
```

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

```
0.000> fact

!! BPT 1:        clihook
    A=            0 B=            0   25172:    K4G  1   (=G1)
*
```

This indicates that the breakpoint occurred when the Cintcode registers `A` and `B` were both zero, and that the program counter is set to 25172 where the next instruction to be obeyed is `K4G 1`. Single step exection can now be performed using the `\` command.

```
* \ A=            0 B=            0   60124:     L1
* \ A=            1 B=            0   60125:    SP3
* \ A=            1 B=            0   60126:    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:        60276        25174        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:        60276        25174        start            3
*
```

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

```
* \ A=            3 B=            1   60127:     LF  60180
* \ A=         fact B=            3   60129:     K9
* \ A=            3 B=            3   60180:   JNE0  60184
* \ A=            3 B=            3   60184:    LM1
* \ A=           -1 B=            3   60185:    AP3
* \ A=            2 B=            3   60186:     LF  60180
* \ A=         fact B=            2   60188:     K4
* \ A=            2 B=            2   60180:   JNE0  60184
* \ A=            2 B=            2   60184:    LM1
* \ A=           -1 B=            2   60185:    AP3
* \ A=            1 B=            2   60186:     LF  60180
* \ A=         fact B=            1   60188:     K4
* \ A=            1 B=            1   60180:   JNE0  60184
* \ A=            1 B=            1   60184:    LM1
* \ A=           -1 B=            1   60185:    AP3
* \ A=            0 B=            1   60186:     LF  60180
* \ A=         fact B=            0   60188:     K4
* \ A=            0 B=            0   60180:   JNE0  60184
* \ A=            0 B=            0   60182:     L1
* \ A=            1 B=            0   60183:    RTN
* \ A=            1 B=            0   60189:    LP3
* \ A=            1 B=            1   60190:    MUL
* \ A=            1 B=            1   60191:    RTN
* \ A=            1 B=            1   60189:    LP3
* \ A=            2 B=            1   60190:    MUL
* \ A=            2 B=            1   60191:    RTN
* \ A=            2 B=            1   60189:    LP3
* \ A=            3 B=            2   60190:    MUL
* \ A=            6 B=            2   60191:    RTN
* \ A=            6 B=            2   60130:    SP9
* \ A=            6 B=            2   60131:    LP3
* \ A=            3 B=            6   60132:    SP8
* \ A=            3 B=            6   60133:    LLL  60148
* \ A=        15037 B=            3   60135:    K4G  94   (=G94)
*
```

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(3) =    6
fact(4) =   24
fact(5) =  120
0.036>
```

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

```
0.010> bcpl com/bcpl.b to junk

32 bit BCPL (18 Jul 2022) with pattern matching, 32 bit target
Code size = 21824 bytes of 32-bit little ender Cintcode
Code size = 20544 bytes of 32-bit little ender Cintcode
Code size = 15832 bytes of 32-bit little ender Cintcode
0.569>
```

This shows that the total size of the compiler is 58,200 bytes and that it can be compiled (on a 2.17GHz CPU) in 0.569 seconds. Since this involves executing 54,579,958 Cintcode instructions, the rate is about 96 million Cintcode instructions per second with the current interpreter. This Cintcode execution rate can be confirmed by running the `sysinfo` command.

```
0.569> sysinfo

TGZDATE: Fri  3 Mar 16:55:54 GMT 2023
Build: Linux
Flags: SOUND CALLC
The hst is a little ender machine
Host address size = 64 bits
BCPL word size    = 32 bits
Execution rate    = 96,796,486 Cintcode instrctions per second

1.642>
```

1.2 A Cintpos Console Session

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

```
meopham$ cintpos

Cintpos System (09 Mar 2010)
0.000 1>
```

There is a directory called `com` that holds the BCPL source code of several Cintpos commands, such as `bcpl.b`, `bench100.b` and `fact.b`. We can inspect `fact.b` using the `type` command as follows.

```
0.000 1> type com/fact.b
SECTION "fact"

GET "libhdr"

LET f(n) = n=0 -> 1, n*f(n-1)

LET start() = VALOF
{ FOR i = 1 TO 10 DO
    writef("f(%i2) = %i8*n", i, f(i))
  RESULTIS 0
}
0.000 1>
```

It can be compiled and run as follows.

```
0.000 1> c bc fact
bcpl com/fact.b to cin/fact hdrs POSHDRS
```

```
BCPL (20 Oct 2009)
Code size =   120 bytes
0.020 1> fact
f( 1) =        1
f( 2) =        2
f( 3) =        6
f( 4) =       24
f( 5) =      120
f( 6) =      720
f( 7) =     5040
f( 8) =    40320
f( 9) =   362880
f(10) =  3628800
0.000 1>
```

There is a benchmark program called `bench100.b` which can be compiled and run as follows.

```
0.000 1> c bc bench100
bcpl com/bench100.b to cin/bench100 hdrs POSHDRS

BCPL (20 Oct 2009)
Code size =  1444 bytes
0.040 1> bench100

bench mark starting, Count=1000000

starting

finished
qpkt count = 2326410  holdcount = 930563
these results are correct
end of run
9.170 1>
```

The latest prompt (`9.170 1>`) indicates that the benchmark program took 9.17 seconds to run and that we are connected to the root command language interpreter running as task one.

When Cintpos starts these are six resident tasks which can be seen using the `status` command as follows.

```
0.000 1> status
Task  1: Root_Cli         running CLI         Loaded command: status
Task  2: Debug_Task       waiting DEBUG
Task  3: Console_Handler  waiting COHAND
Task  4: File_Handler     waiting FH0
Task  5: MBX_Handler      waiting MBXHAND
Task  6: TCP_Handler      waiting TCPHAND
0.010 1>
```

Task 2 is an interactive debugging aid, task 3 handles communication between tasks and the keyboard and display devices, task 4 handles communication between tasks and the filing system, task 5 provides a mailbox facility that allows

communication of short text messages between tasks and, finally, task 6 handles TCP/IP communication between tasks and the internet.

Tasks may be dynamically created and destoyed. For instance, the `run` command will create a new CLI task giving it a command to run.

```
0.010 1> run status
0.000 1> Task  1: Root_Cli         waiting CLI        No command loaded
Task  2: Debug_Task       waiting DEBUG
Task  3: Console_Handler  waiting COHAND
Task  4: File_Handler     waiting FH0
Task  5: MBX_Handler      waiting MBXHAND
Task  6: TCP_Handler      waiting TCPHAND
Task  7: Run_Cli          running CLI         Loaded command: status
```

Notice that the root CLI (task 1) completes the execution of the `run` command and issues a prompt (`0.000 1>`) before the newly created CLI (task 7) has had time to load and run the `status` command. As soon as task 7 finishes running the `status` command it commits suicide leaving the original 6 tasks.

The `bounce.b` program provides a demonstration of how communication between Cintpos tasks works.

```
0.000 1> type com/bounce.b
SECTION "bounce"

GET "libhdr"

LET start() BE qpkt(taskwait()) REPEAT
0.000 1>
```

It can be compiled and run as follows.

```
0.000 1> c bc bounce
bcpl com/bounce.b to cin/bounce hdrs POSHDRS

BCPL (20 Oct 2009)
Code size =    60 bytes
0.010 1> run bounce
0.000 1> status
Task  1: Root_Cli         running CLI         Loaded command: status
Task  2: Debug_Task       waiting DEBUG
Task  3: Console_Handler  waiting COHAND
Task  4: File_Handler     waiting FH0
Task  5: MBX_Handler      waiting MBXHAND
Task  6: TCP_Handler      waiting TCPHAND
Task  7: Run_Cli          waiting CLI         Loaded command: bounce
0.000 1>
```

The `status` output shows that the `bounce` program is running as task 7 and is suspended in `taskwait` waiting for another task to send it a packet. When it receives a packet it immediately returns it to the sender and waits for another to arrive. We can send a suitable packet to `bounce` using the `send` command whose source code is as follows.

```
0.000 1> type com/send.b
SECTION "send"

GET "libhdr"

GLOBAL { task: 200; count: 201 }

LET start() BE
{ LET pkt = VEC 2
  LET argv = VEC 50

  UNLESS rdargs("TASK/n,COUNT/n", argv, 50) DO
  { writef("Bad arguments for SEND*n")
    stop(20)
  }

  task, count := 7, 1_000_000
  IF argv!0 DO task  := !argv!0
  IF argv!1 DO count := !argv!1

  pkt!0, pkt!1, pkt!2 := notinuse, task, count

  writef("*nSending a packet to task %n, %n times*n", task, count)

  { LET k = pkt!2
    UNLESS k BREAK
    pkt!2 := k-1
    qpkt(pkt)
    pkt := taskwait()
  } REPEAT

  writes("Done*n")
}
0.010 1>
```

This program creates a packet consisting of a vector (one dimensional array) of three elements. The first is used by the system for chaining packets together and must be initialised the the special value `notinuse`. The next element of the packet (`pkt!1`) holds the destination task number and the final element (`pkt!2`) holds a value (initially 1000000) which is going to be used as a counter. The `REPEAT` loop decrements this counter field and sends the packet using `qpkt` to the bounce task suspending itself in `taskwait` until the packet returns. Control leaves the `REPEAT` loop when the counter reaches zero, causing `send` to output the message `Done`. We can compile and run `send` as follows.

```
0.010 1> c bc send
bcpl com/send.b to cin/send hdrs POSHDRS

BCPL (20 Oct 2009)
Code size =   252 bytes
0.020 1> send

Sending a packet to task 7, 1000000 times
Done
3.940 1>
```

This demonstration shows that a packet may be sent from one task to another 2 million times in 3.94 seconds. This corresponds to a rate of just over half a million times per second.

Chapter 2

The BCPL Language

The design of BCPL owes much to the work done 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 theses. Another problem (that became of particular interest to me) was that the implementation at Cambridge 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] designed by Strachey specifically for this task. It was a delightfully elegant work of art in its own right it is well

worth study. A variant of GPM, called BGPM, is included in the standard BCPL distribution.

BCPL was initially similar to this 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 special care is needed when commenting out fragments of program containing `//` comments and string constants. Comments are equivalent to white space and so may not occur within 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 named file. Early versions of the compiler only inserted the file up to the first occurring dot but now the entire file is inserted. By convention, `GET` directives normally appear on separate lines. If the filename does not end in `.h` or `.b` the extension `.h` is added.

The name is looked up by first searching the current directory and then the directories specified by the environment variable whose name is held in the `rtn_hdrsvar` of the rootnode, but this can be overridden using the `hdrs` compiler option. The default environment variable for BCPL headers is `BCPLHDRS` under Cintsys and `POSHDRS` under Cintpos. Header files are normally in the `g/` directory in the root directory of the current system. To check whether the environment variables are set correctly, enter `cintsys` or `cintpos` with the `-f` option as suggested in Section 3.6.

2.1.3 Conditional Compilation

A simple mechanism, whose implementation takes fewer than 20 lines of code in the lexical analyser allows conditional skipping of lexical symbols. It uses directives of the following form:

`$$`*tag*
`$<`*tag*
`$~`*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 `$$`*tag* directive. All the lexical tokens between `$<`*tag* and `$>`*tag* are skipped (treated as comments) unless the specified tag is set. All the lexical tokens between `$~`*tag* and `$>`*tag* are skipped unless the specified tag is not set. Skipping ends when a matching `$>` item is found or when a section separating dot ('.') encountered, or when the end of input is reached. GET files are not inserted and `$$` items have no effect while input is being conditially skipped.

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

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. Use of this mechanism is no longer recommended since it often leads to obscure programming errors. 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 2.2.11. BCPL expressions always vield values of the same bit length, normally 32 or 64 bits.

2.2.1 Names

A name is of a sequence of letters, digits, dots and underlines starting with a letter, but it must not one of the reserved words (such as `IF`, `WHILE` or `RESULTIS`). The use of dots in names is no longer recommended, and should be replaced by underscores. Double dots are no longer permitted in names because `..` is the range operator used in the pattern matching extension.

A name may be declared as a local variable, a static variable, a global variable, a manifest constant, a label or a function or routine. Since the language is typeless, the value of a name is just a bit pattern whose interpretation depends on how it is used. This is similar to the way values in central registers of most computers are used.

2.2.2 Constants

Integer constant consist of a sequence of digits. Underscores are allowed in integer anywhere after the first digit. Bit pattern binary, octal or hexadecimal constants are represented, respectively, using `#B`, `#O` or `#X` followed by digits of the appropriate sort. All letters in bit pattern constants can be in upper or lower case. The letter `'o'` in octal constants is optional. Underscores are allowed anywhere

in bit pattern constants after the # except between the # and b. o and x. The following are examples of valid numbers:

```
1234
1_234_456
#B_1011_1100_0110
#o377
#X3fff
#x_DEADC0DE
#_377
```

Since August 2014, floating point constants are allowed, such as the following:

```
1234.0
123.
.25
1.234_456e-5
10e6
but       1..10  is the same as 1 .. 10
and names may not contain consecutive dots
```

Note that 12.34 is a floating point number, but 12..34 is 12 followed by the range operator .. and 34. A floating point constant must contain a decimal point with at least one digit before or after the decimal point. It can be given an exponent consisting of e or E followed by a possibly signed exponent, as in 123e5, 123.e+5 or .123e-5,

BCPL uses the standard IEEE representation for floating point numbers using the same BCPL word length as the compiler. For 32-bit BCPL the floating point format is as follows. The left most bit is the sign with 1 representing negative. The next 8 bits hold an unsigned number e in the range 0 to 255 with 0 and 255 specifying some special values such as zero, infinity or various error values. The values between 1 and 254 specify binary exponents in the range -126 to +127 equal to $e - 127$. The remaining 23 bits are the fractional bits of the significand. For non zero values, the significand has 24 bits with its left most bit being 1 followed by these 23 fractional bits representing a value greater than or equal to 1.0 and just less than 2.0. Note that 1+8+23=32. The value of the floating point number is the significand multiplied by 2^{e-127}. As a special case, the number 0.0 is represented by a bit pattern of 32 zeroes.

If the BCPL word length is 64, the exponent has 11 bits and the significand has 52. Note that 1+11+52=64.

The compiler does not use any floating point operators or constants using, where necessary, calls of the form sys(Sys_flt,...) to perform any floating point calculations needed. This allows the compiler to be compiled using older versions of the compiler. Floating point constants are currently only compiled correctly if the BCPL word length of the compiler is the same as that of the target code.

`TRUE` and `FALSE` are reserved words that have values `-1` and `0`, respectively, representing the two truth values. They can be used in manifest constant expressions. Whenever a boolean test is made, the value is compared with with `FALSE` (`=0`). `BITSPERBCPLWORD` is also a reserved word whose value is 32 or 64 giving the BCPL word length currently being used. This constant was added on 16 May 2013 to allow the same header file to be used on both 32- and 64-bit BCPL systems. It is used in the MANIFEST declarations of constants such as `bytesperword` and `minint` that are word length dependent. If you are using an older BCPL compiler with the latest version of `libhdr.h` you will need to uncomment a line that declares `BITSPERBCPLWORD` as a MANIFEST constant with the appropriate value for the system you are using.

The commands `BREAK`, `LOOP`, `NEXT`, `EXIT` and `ENDCASE` are permitted within expressions and have the same effect as the corresponding commands described in Section 2.3.8.

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

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

Constants 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` to access fields of given length, shift and offset relative to a pointer as described in Section 2.2.6. The *len* and *shift* components are optional. Their omission has the following effect.

`SLCT` *shift*`:`*offset*	means	`SLCT 0:`*shift*`:`*offset*
`SLCT` *offset*	means	`SLCT 0:0:`*offset*

Character constants consist of a single character enclosed in single quotes (`'`). Character constants behave like integers typically in the range 0 to 255 corresponding to its normal ASCII encoding, but can be larger using unicode characters as describer below.

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 131.
`*#u`	Set the encoding mode explicitly 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 white space characters. In this context, comments introduced by `'//'` are treated as white space, but those introduced by `'/*'` are not.

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

the result is as follows

```
Unicode hex 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 2.2.12. The space for a static vector is allocated for the life time of the program and its elements are updateable.

2.2.3 Function Calls

Syntactically, a function call is an expression followed by a, possibly empty, argument list enclosed in paretheses as in the following examples.

```
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. If the function being called was defined by a routine definition, the result of the call will be undefined. The arguments are evaluated and laid out in consecutive stack locations where they become the initial values of the formal parameters of the called function or routine. There is no need for the number of arguments to be the same as the number of formal parameters. See Section 2.4.5 for more details. If the expression specifying the function to be called has the `FLT` tag the so does the result of the call.

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:

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

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 containing the possible functions to call. Element E of this vector is applied to the given set of arguments. E is normally a manifest constant. An example program illustrating method calls can be found in `BCPL/bcplprogs/demos/objdemo.b` in the BCPL distribution.

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 BCPL word sized location specified by E. E can only be a variable name or an expression with leading operator `!`. Pointers to consecutive locations are consecutive integers.

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, `~` return the bitwise complement of the value.

`FLOAT` *E* converts the integer E to its corresponding floating point value. `FIX` *E* converts the floating point value E to its closest integer representation. `#ABS` *E* returns the absolute value of the floating point number E, and `#+` and `#-` perform monadic plus and minus on floating point values.

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. Since words of memory have consecutive integer addresses, the expression E_1`!`E_2 is exactly equivalent to `!(`E_1`+`E_2`)`.

An expression of the form E_1`[`E_2`]` has recently been added. It is syntactically like a function call but is equivalent to E_1`!`E_2.

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 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 2.2.12) equal to `SLCT` *length*`:`*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 *length* and is *shift* bits from the right hand end of the word. A length of zero is interpreted

as the longest length possible consistent with *shift* and the word length of the implementation.

An `OF` expression can be used on right and left hand sides of assignments but not as the operand of `@`. When used in a right hand context the selected field is shifted to the right with vacated positions filled with zeros. A shift to the left is performed when a field is updated. Suppose `p!3` holds `#x12345678`, the expression `(SLCT 12:8:3) OF p` yields `#x456` and after the assignment:

```
(SLCT 12:8:3) OF p := #xABC
```

the value of `p!3` will be `#x123ABC78`.
The operator `..` is a synonym of `OF`.

An expression of the form E_1`<<`E_2 (or E_1`>>`E_2) evaluates E_1 and E_2 to yield a bit pattern, w say, and an integer, n say, and returns the result of shifting w to the left (or right) by n bit positions. Vacated positions are filled with zeroes. Shifts of the word length or more return 0, and negative shifts return undefined typically zero results, although on some versions of BCPL they reverse the direction of the shift.

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).

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 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.

The floating point operators `#*`, `#/`, `#+`, `#-`, `#=`, `\#~=`, `#<`, `#>`, `#<=`, `#>=` and `#->` have recently been added to standard BCPL. They have the same binding power as the corresponding integer operators. Beware that, with older versions of the BCPL compiler that do not implement the FLT feature, it is easy make mistakes such as `-1.2` which performs integer negation of the bit patterns representing `1.2`. The expression should have been written `#-1.2`. With the FLT feature `-` would have been automatically replaced by `#-` in this situation. See Section 2.7 for details.

2.2.7 Boolean Evaluation

Expressions used to control the flow of execution in conditional constructs such as `IF` and `WHILE` commands are evaluated in a *Boolean context*. This effects the treatment of the operators `~` `&` and `|` whose operands are evaluated in Boolean contexts. In a Boolean context, the operands of `&` and `|` are evaluated from left to right until the value of the condition is known, and `~` negates the condition.

2.2.8 `MATCH` Expressions

A `MATCH` expression has the following form:

`MATCH (` *args* `)`
`:` *P* `,..,` *P* `=>` *E*
...
`:` *P* `,..,` *P* `=>` *E*
`.`

It consists of the word `MATCH` followed by a list of arguments enclosed in parentheses, followed by a sequence of one or more match items terminated by an optional dot. The match items are applied to the arguments as described in Section 2.5, yielding the value of the expression in the first match item to be satisfied. If the `MATCH` expression is being evaluated in `FLT` mode, all its result expressions are evalated in `FLT` mode. If none are satisfied the result is zero (either 0 or 0.0).

Within a match item the command `NEXT` causes control to pass to the next match item, and `EXIT` causes the `MATCH` expression to terminate yielding the value 0 or 0.0.

2.2.9 `EVERY` Expressions

An `EVERY` expression has the following form:

`EVERY (` *args* `)`
`:` *P* `,..,` *P* `=>` *E*
...
`:` *P* `,..,` *P* `=>` *E*
`.`

It consists of the word `EVERY` followed by a list of arguments enclosed in parentheses, followed by a squence of one or more match items terminated by an optional dot. The match items are applied to the arguments as described in Section 2.5, yielding the sum of the values of the expressions of successful match items. IF the `EVERY` expression is being evaluated in `FLT` mode, all it result expressions are also evaluated in `FLT` mode and the sum performed using `#+`. If none are succesful the result is 0 or 0.0.

Within a match item the command NEXT causes control to pass to the next match item, and EXIT causes the EVERY expression to terminate yielding the sum accumulated so far.

2.2.10 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. VALOF expressions are often used as the bodies of functions.

2.2.11 Expression Precedence

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

The syntax of BCPL is specified by the diagrams in Appendix A, but a summary of the precendence of expression operators is given in table 2.1. The precedence values are in the range 0 to 10, 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 (a-b)-c, while a->x,b->y,z is right associative and so is equivalent to a->x,(b->y,z).
Notice that these precedence values imply that

```
          ! f (x,y)  means  ! (f (f,y))
          FLOAT v!i  means  FLOAT (v!i)
              ! @ x  means  ! (@ 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)
 b1-> x, b2 -> y,z   means  b1 -> x, (b2 -> y, z)
b1->  b2 -> x,y, z   means  b1 -> (b2 -> x, y), z
```

2.2.12 Manifest Constant Expressions

Manifest constant expressions are expressions whose values can be determined before the program is run. They may only consist of manifest constant names, numbers and character constants, TRUE, FALSE, BITSPERBCPLWORD, ?, the operators MOD, SLCT, FIX, FLOAT, *, /, +, -, ABS, the relational operators, #*, #/, #+,

10	Names, Literals, ?, TRUE, FALSE, BITSPERBCPLWORD, BREAK, LOOP, ENDCASE, NEXT, EXIT (E), Function and method calls Subscripted expressions using [and]	
10	SLCT :	SLCT constant
9L	! % OF	Dyadic
8	FLOAT FIX ! @	Monadic
8L	* / MOD #* #/ #MOD	
7	+ - ABS #+ #- #ABS	Monadic and Dyadic
6	= ~= <= >= < > #= #~= #<= #>= #< #>	Extended Relations
5L	<< >>	
4	~	Bitwise and Boolean operators
4L	&	
3L	|	
2L	EQV XOR	
1R	-> ,	Conditional expressions
0	MATCH EVERY VALOF TABLE	

Table 2.1: Operator precedence

#-, #ABS, #MOD, the floating point relational operators, <<, >>, NOT, ~, &, |, EQV, 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 OF.

Manifest constants are evaluated at compile time using arithmetic of the word length of the compiler. So on a 32 bit compiler, integers can only be represented correctly if they have no more than about 9 decimal digits and floating point constants will have a precision limited to 6 or 7 digits, and this will be true even when compiling for a 64 bit target. When using a 64 bit compiler integers may have up to 18 of 19 digits and the precision of floating point numbers is about 15 digits. If a 64 bit compiler has a 32 bit target the range and precision of constants is, of course, limited to what can be represented by 32 bit words.

2.3 Commands

The primary purpose of commands is to update variables, to perform input/output operations, and to control the flow of control. They are described in

the following sections.

2.3.1 Assignments

Simple assignments have the following possible forms:

$$L\texttt{:=}E$$
$$L\texttt{\#:=}E$$
$$L op\texttt{:=}E$$

where *op* is one of the following operators: `!`, `*`, `/`, `MOD`, `+`, `-`, `#*`, `#/`, `#MOD`, `#*`, `#-`, `<<`, `>>`, `&`, `|`, `EQV` or `XOR` and L is a variable name or an expression of one of the following forms: $E_1\texttt{!}E_2$, $\texttt{!}E$, $E_1\texttt{\%}E_2$, $\texttt{\%\%}E$ or K `OF` E. K is normally a selector of the form `SLCT` *`length:shift:offset`*. For `:=` and `#:=` assignments, the right hand side is evaluated and used to update the location specified by the left hand side. For *op*`:=` assignments, the value to assign is $Lop\texttt{:=}R$. If `#` is present a floating point assignment is performed. This causes the right hand side to be evaluated in FLT mode with the restriction that the left hand side must refer to a full word. See Section 2.7 for details. Typical simple assignments are as follows:

```
cg_x := 1000
v!i := x+1
!ptr := mk3(op, a, b)
str%k := ch
%strp := 'A'
SLCT 8:10:1 OF p +:= 1
p &:= #x7F
w!p #*:= a
```

Assignments are not permitted to start with any of the following keywords: `MATCH`, `EVERY`, `BREAK`, `LOOP`, `ENDCASE`, `NEXT` or `EXIT`.

A multiple assignment has the following possible forms:

$$L_1\texttt{,..,}L_n\ \texttt{:=}\ E_1\texttt{,..,}E_n$$
$$L_1\texttt{,..,}L_n\ \texttt{\#:=}\ E_1\texttt{,..,}E_n$$
$$L_1\texttt{,..,}L_n\ op\texttt{:=}\ E_1\texttt{,..,}E_n$$

These constructs allows a single command to make several assignments without the need to have to enclose them in section brackets. The assignments are done strictly from left to right and are exactly eqivalent to:

$$L_1\texttt{:=}E_1\ \texttt{;...;}\ L_n\ \texttt{:=}\ E_n$$
$$L_1\texttt{\#:=}E_1\ \texttt{;...;}\ L_n\ \texttt{\#:=}\ E_n$$
$$L_1 op\texttt{:=}E_1\ \texttt{;...;}\ L_n\ op\texttt{:=}\ E_n$$

This conversion is performed before the application of the rules of the FLT feature. See Section 2.7 for details.

2.3.2 Routine 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. Any results produced are discarded.

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` are synonyms and may be omitted whenever they are followed by a command keyword. To execute a conditional command, the expression E is evaluated in a Boolean context. If it yields a non zero value and C_1 if present is executed. If it yields zero and C_2 if present 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 `BY` K `DO` C
`FOR` N = E_1 `TO` E_2 `DO` C
`FOR` N = E_1 `BY` K `DO` C
`FOR` N = E_1 `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.

A `FOR` command declares the control variable N as a new local variable initialised with the value of E_1. The scope of the control variable is the body of the `FOR` command. The control variable may not be given the `FLT` tag. If `BY` is present, the step length is K which must be a manifest constants (see Section 2.2.12), but if omitted `BY 1` is assumed. If `TO` is present, the end limit is E_2, but if omitted an infinite end limit with the same sign as the step length is assumed. Until N moves beyond the end limit, the command C is executed and N increment by the step length (which can be negative).

2.3.5 SWITCHON command

A SWITCHON command has the following form:

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

Labels of the form DEFAULT: or CASE K: are permitted in the command sequence. E is evaluated and control is passed to the matching case label if it exists, otherwise a jump is made to the default label but, if that is not given, control passes to the point just after the switchon command.

2.3.6 MATCH Command

A MATCH command has the following form:

MATCH (*args*)
: P ,.., P BE C
...
: P ,.., P BE C
.

It consists of the word MATCH followed by a list of zero or more arguments enclosed in parentheses. This is followed by followed by one or more match items and optionally terminated by a dot. The match items are applied in turn to the arguments as described in Section 2.5 executing the command in the first match item to be satisfied. If none are satisfied the match command has no effect. Within a match item the the command NEXT causes control to pass to the next match item, if any, and the command EXIT causes the MATCH command to terminate.

2.3.7 EVERY Command

An EVERY command has the following form:

EVERY (*args*)
: P ,.., P BE C
...
: P ,.., P BE C
.

It consists of the word EVERY followed by a list of arguments enclosed in parentheses, followed by a sequence of one or more match items optionally terminated by a dot. The match items are applied in turn to the arguments as described in Section 2.5 executing the commands of every match item that is satisfied.

Within a match item the commands NEXT causes control to pass to the next match item, if any, and EXIT causes termination of the entire match list. If in

a MATCH expression or a function definition the result is zero, but if in an EVERY expression the result is the sum accumulated so far. MATCH and EVERY commands and routines do not return results.

2.3.8 Flow of Control

The commands in this section affect the flow of control.

RESULTIS E causes E to be evaluated and returned as the result of the smallest textually enclosing VALOF expression which must be within the current function or routine.

RETURN normally causes a return from the current routine, but if encountered in a function it returns with the value zero.

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). The destination of the jump must be within the current function or routine. For a REPEAT command, LOOP causes 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 which must be within the current function or routine.

ENDCASE causes execution to jump to the point just after the end of the smallest enclosing SWITCHON command which must be within the current function or routine.

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

NEXT is a newly added command that can only be used inside a the pattern or body of a match item. It causes control to pass to the start of the next match item, if any.

EXIT is a newly added command that can only be used inside a the pattern or body of a match item. It causes termination of the match construct.If in a Match expression or function body, it returns a zero result. If in an EVERY expresion, it returns the sum acccumulated so far. MATCH and EVERY commands and routine do not return results.

FINISH only remains in BCPL for historical reasons and should not be used. It is equivalent to the call stop(0, 0) which causes the current program to terminate. See the description of stop(code, res) page 73.

2.3.9 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:

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

where C_1 to C_m are commands. It is permissible to have no commands in a command sequence, thus `{}` is allowed and performs no commands.

Any semicolon occurring at the end of a line may be omitted. For this rule to work, infixed expression operators may never start a line (see Section 2.2.11).

A command sequence can also be formed using the symbol `<>` which behaves like semicolon but is more binding than `DO`, `THEN`, `ELSE`, `REPEATWHILE`, `REPEATUNTIL` and `REPEAT`. It purpose is to reduce the need for section brackets (`{` and `}`) as in

```
IF x<y DO t:=x <> x:=y <> y:=t
```

which is equivalent to:

```
IF x<y DO { t:=x; x:=y; y:=t }
```

This sequencing operator has been included since it was in the extended non standard version of BCPL at MIT and extensively used in the PAL compiler. See for instance `com/pal75.b`.

2.3.10 Blocks

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

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

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:

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

where N_1`,...,`N_n are names (see Section 2.2.1) and K_1`,...,`K_n are manifest constant expressions (see Section 2.2.12). Each name is declared to have the constant value specified by the corresponding manifest expression. The details have recently changed due to the introduction of the `FLT` feature, see Section 2.7 on page 40. Manifest names with the `FLT` tag have floating point values otherwise they are integers. If a value specification (`=`K_1) is omitted in the declaration of the first name, the value 1 or 1.0 is assumed. If a value specification (`=`K_i) is omitted in later declarations a value 1 or 1.0 greater than the value of the previous name is used. An automatic conversion between integer and floating point is performed if necessary. Thus, the declaration:

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

declares `A`, `B`, `C`, `D` and `E` to have manifest values `0`, `1`, `10.0`, `11` and `110`, respectively.

2.4.3 Global Declarations

The global vector is a permanently allocated region of memory that may be directly accessed by any (separately compiled) section of a program (see Section 2.6). It provides a mechanism for linking together separately compiled sections. A GLOBAL declaration allows a names to be explicitly associated with elements of the global vector. The syntax is as follows:

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

where $N_1, \ldots, N_n$ are names possibly prefixed by `FLT` (see Section 2.2.1) and $K_1, \ldots, K_n$ are manifest constants (see Section 2.2.12). Each constant specifies which global vector element is associated with each variable, and if `FLT` is specified the variable is assumed to hold a floating point number.

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 $N_1, \ldots, N_n$ are names possibly prefixed by `FLT` (see Section 2.2.1) and $K_1, \ldots, K_n$ are manifest constant expressions (see Section 2.2.12). 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 0 or 0.0 is implied. Thus, the declaration:

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

declares `A`, `B`, `C`, `D` and `E` to be static variables having initial values `0`, `0`, `10.0`, `0` and `100`, respectively.

2.4.5 `LET` Declarations

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

Local variable, local vector, function definitions can be combined using the word `AND`. The only effect of this is to extend the scope of names defined back to the word `LET`, thus allowing the definition of mutually recursive functions.

Local Variable Definitions

A local variable definition has the following form:

$$N_1\,,\ldots,\ N_n\ \texttt{=}\ E_1\,,\ldots,\ E_n$$

where $N_1\,,\ldots,N_n$ are names possibly prefixed by `FLT` (see Section 2.2.1) and $E_1\,,\ldots,E_n$ are expressions. The names, N_i, are allocated space in the current stack frame and are initialized with the corresponding values of E_i. Such variables are called *dynamic* variables since they are allocated when the definition is executed and cease to exist when control leaves their scope.

The variables $N_1\,,\ldots,N_n$ are allocated consecutive locations in the stack frame of the current function and so, for instance, the variable N_i may be accessed by the expression $(@N_1)!(\texttt{i}-\texttt{1})$. This feature is a recent addition to the language. When a local variable has the `FLT` tag it initial value expression is evaluated in `FLT` mode. For details see Section 2.7 on page 40.

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

Local Vector Definitions

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

where N is a name which may not be qualified by the `FLT` tag and K is a manifest constant. A location is allocated for N and initialized to point to a vector whose lower bound is `0` and whose upper bound is K. The variable N and the vector elements ($N\texttt{!0}$ to $N\texttt{!}K$) reside in the stack frame of the current function and only continue to exist while control remains within the function.

Function Definitions

These definitions have the following form:

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

where N is the name possibly prefixed by `FLT` of the function being defined, and $N_1\,,\ldots,N_n$ are its formal parameters, each of which may be prefixed by `FLT`. A function defined using `=` returns E as result, but one defined using `BE` and executes the command C and does not return a defined a result. Functions defined using `BE` are often called routines. If the function name has the `FLT` prefix, the result, if any, of a call is assumed to be a floating point value. For details see Section 2.7 on page 40.

Some example functions definitions are as follows.

```
LET wrpn(n) BE { IF n>9 DO wrpn(n/10)
                 wrch(n MOD 10 + '0')
```

```
                }
LET gray(n) = n XOR n>>1
LET next() = VALOF { c := c+1
                     RESULTIS !c
                   }
```

A function can be defined using pattern matching by giving its name, which may be prefixed by `FLT`, followed a one or more match items of the form:

: *Plist* `=>` *E*.

optionally followed by a dot. The way patterns work is described in on page 36.

A routine can be defined using pattern matching by giving its name followed a one or more match items of the form:

: *Plist* `BE` *C*.

optionally followed by a dot. The way patterns work is described in on page 36.

If the name of a pattern function has an `FLT` prefix, its result is assumed to be a floating point number.

If a function or routine is defined in the scope of a global variable with the same name, the global variable is given an initial value representing that function or routine (see section 2.6).

Since March 2023, functions, routine and pattern functions and routines to can have their names prefixed by `FLT`. Calls of functions with the `FLT` tag are assumed to return floating point results. If a function is declared having the FLT tag is in the scope of a global variable of the same name, the global must also have been declared with the FLT tag. If the function did not have the FLT tag, the global should also not have the tag. If a match expression is evaluated in FLT mode, all its result expressions are evaluated in FLT mode. Likewise, if an `EVERY` expression is evaluated in FLT mode the result is the floating point sum of its successful result expressions.

If a function is called as a command its result is thrown away, and if a routine is called when a result is required its result is undefined. See section 2.2.3 for information about the syntax of function and routine calls.

The arguments of a functions and routines behave like named elements of a dynamic vector and so exist only for the lifetime of the call. This vector has as many elements as there are formal parameters and they receive their initial values from the actual parameters of the call. Functions and routines are variadic; that is, the number of actual parameters need not equal the number of formals. If there are too few actual parameters, the missing ones are left uninitialized, and if there are too many actual parameters, the extra ones are evaluated and then discarded. Notice that arguments can be accessed by the expressions `(@x)!0`, `(@x)!1`, `(@x)!2`,... where `x` is the first argument. This feature is useful in the

definition of functions, such as `writef`, having a variable number of arguments. The scope of the formal parameters is the body of the function or routine.

Function and routine 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 Cintcode instruction for the call and one for the return.

There are two important restrictions concerning functions and routines. One is that a `GOTO` command cannot make a jump to a label not declared within the current function or routine, although such non local jumps can be made using `level` and `longjump`, described on page 65. The other is that dynamic free variables are not permitted.

2.4.6 Dynamic Free Variables

Free variables of a function or routine are those that are used but not declared in the function or routine, and they are restricted to be either manifest constants, static variables, global variables, functions, routines or labels. This implies that they are not permitted to be dynamic variables (ie local variables) of another function or routine. There are several reasons for this restriction, including the ability to represent a function or routine by a single BCPL word, the ability to provide a safe separate compilation with the related ability to assign functions and routines to variables. It also allows calls 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. Note that C adopted the same restriction, although in that language it is imposed by the simple expedient of insisting that all functions 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 Patterns

This section describes the MCPL style pattern matching mechanism that is now included in BCPL.

Pattern matching is an important feature since it provides a mechanism to select an outcome based on the values of locations in a structure referenced directly or indirectly from a set of arguments. Names, called pattern variables, can be associated with locations in the structure. Such variables have much in common with ordinary local variables. A name declared in the pattern of a match item can only be used in the pattern, expression or command in the match item.

Patterns are used in function and routine definitions and in `MATCH` and `EVERY` commands and expressions. They are applied to the arguments given by the function or routine calls or the arguments of `MATCH` and `EVERY` constructs.

Each match construct contains a list of one or more match items, each consisting of a pattern followed by either `=>` and an expression or `BE` and a command.

The list of match items is optionally terminated by a dot. If the first match item in the list uses the operator `=>` then all the subsequent items must also use `=>`, otherwise all the match items must use `BE`.

For functions, routines and `MATCH` expressions and commands, the patterns are tested in turn and the first to be successful causes the related expression or command to be evaluated.

For an `EVERY` command, the commands in all successful match items are executed, and for an `EVERY` expression the values of the expressions of all successful match items are summed and returned as the result.

A pattern is composed of terms and three pattern operators: comma (`,`), vertical bar (`|`) and juxaposition. The syntax specification is given in Figure A.4

on page 292. A term typically tests the contents of a memory location. It can be a relational operator such as `<=` or `#>` which compares the contents of the current location with the value of its right hand operand. It can be a possibly sign constant which is directly compared with the location. It can be a range test consisting of the operator `..` or `#..` applied to two operands which must be names or possibly signed numerical constants. This construct succeeds if the value in the current location is not less than the left operand and not greater than the right hand one.

The term query (`?`) always matches its location. If a term is just a manifest constant name it behaves as an explicit constant with that value. If it is a non manifest name, it is a local variable declaration associating the name with the current location and, as with other local variables, the name can have a `FLT` prefix. Such declarations always successfully match their locations.

A term can also be one of the escape commands `BREAK`, `LOOP`, `ENDCASE`, `NEXT` or `EXIT`. They provide an escape mechanism behaviing in exactly the same way as the corresponding commands.

If T_1 and T_2 are two terms, the juxtaposition T_1 T_2 matches if T_1 and T_2 both successfully match the current location. The pattern T_1`|`T_2 is successful if one or both terms match the current location. The pattern T_1`,`T_2 matches if T_1 matches the current location and T_2 matches the location whose address is one greater than that of the current location. Juxtaposition has the highest precedence and comma has the lowest. Vertical bar is less binding than juxtaposition but more binding than comma. A term may use parentheses to override the normal precedence of these operators.

The last form of term encloses a pattern in square brackets as in `[`P`]` where P is a pattern. This construct matches P with the location pointed to by the contents of current location.

The following examples illustrate how pattern matching can be used.

```
LET ways
: 0,          ?  => 1
: ?,        [0]  => 0
: s, coins[>s]  => ways(s, coins+1)
: s, coins[v ]  => ways(s, coins+1) + ways(s-v, coins)

LET eval
: [Id,     x], e => lookup(x, e)
: [Num,    k], ? => k
: [Mul, x, y], e => eval(x, e) * eval(y, e)
: [Div, x, y], e => eval(x, e) / eval(y, e)
: [Add, x, y], e => eval(x, e) + eval(y, e)
: [Sub, x, y], e => eval(x, e) - eval(y, e)
```

If a pattern variable is associated with an argument of the match construct it behaves exactly like an ordinary local variable addressing a location with a known offset relative to the P pointer. But if the variable is inside one or more square bracket terms its location is determined by a sequence of indirections. For example consider the following routine.

```
LET r : a, b[x, [y]], c BE { b:=c; x:=y }
```

Here `x` is treated as tt b!0 and `y` is equivalent to `b!1!0`, so the assignment `x:=y` is equivalent to `b!0:=b!1!0` which is executed after the assignment `b:=c`.

A more significant example is the function `rotleft` defined in `bcplprogs/patdemos/splay.b`. This function is worth diligent study. It performs a transformation of a binary tree represented by nodes of the form `[key,val,parent,left,right]`.

```
AND rotleft    // Promote right child      p          p
: n[key, val,                          //   |          |
    np[?,?,?,npl,npr],                 //   n    =>    r
    nx,                                //  / \        / \
    nr[?,?,nrp,nry[?,?,nryp,?,?],nrz]  // x   r      n   z
   ] BE                                //    / \    / \
                                       //   y   z  x   y
{ LET y = nry
  // The order of the assigments was chosen with great care.

  TEST np              // Test if n has a parent.
  THEN TEST n=npl
       THEN npl := nr  // Update the parent's left branch.
       ELSE npr := nr  // Update the parent's right branch.
  ELSE root := nr      // n has no parent, so r is the new root.
  IF nry DO nryp := n  // If y exists, its parent should be n.

  nrp := np
  nry := n
  np  := nr
  nr  := y
}
```

2.6 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 functions and routines 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 can be called from the function `start` defined in `demomain.b`.

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

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

Although the global vector mechanism has disadvantages, particularly in the organisation of library packages, there are some compensating benefits arising from its extreme simplicity. One is that the output of the compiler is available directly for execution without the need for a link editing step. Sections may also be loaded and unloaded dynamically during the execution of a program using the library functions `loadseg` and `unloadseg`, and so arbitrary overlaying schemes can be organised easily. An example of where this is used is in the implementation of the Command Language Interpreter described in Chapter 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, disc space was very limited and modern day linkage editors had not been invented; however, some of its advantages are still relevant today.

2.7 The FLT Feature

BCPL was originally designed for the implemention of compilers and other system software such as text editors, pagination programs and operating systems. These applications typically did not require floating arithmetic and so the language did not include such features. Indeed, many early machines on which BCPL ran had word lengths of 16 or 24 bits which were of insufficient for useful floating point numbers. Even on 32-bit machines the precision of floating point numbers is limited to about 6 decimal digits which is insufficient for serious scientific calculation. For 50 years I resisted putting floating point into BCPL but have recently given in. This is mainly due to the need to use 32-bit floating point in the BCPL interface with the OpenGL graphics library described `bcplraspi.pdf`. The same document also describes a flight simulator where the inaccuracy of 32-bit floating point can be put down to random turbulence of the air through which the aircraft is flying. Single precision floating point is also useful when representing samples in digital sound.

The FLT feature was added to BCPL in March 2018 to make computations involving floating point numbers more convenient. This feature may look as if data types have been added to BCPL, but the language still retains its typeless nature in the sense that all expression values are the same size and the compiler generates no error messages relating to data types. The sole effect of this feature is to cause some integer operators such as `+` and `-` to be replaced automatically by their floating point versions `#+` and `#-` and to automatically replace some integer constants by floating point ones when required. These conversions are specified by some simple rules, but before applying them it is assumed that all simultaneous assignments and variable definitions have been automatically replaced by sequences of non simultaneous constructs.

Some expressions such are the operands of `#+` are evaluated in *FLT mode* meaning that they are expected to yield floating point results. Expressions may also have the *FLT tag* when they are believed to return floating point values. The rules relating to FLT tag and FLT evaluation are as follows.

(1) Global, static and manifest, local variable and formal parameter names can be prefixed by `FLT` giving them the `FLT` tag. All other names, namely `FOR` loop control variables, vectors and program labels, may not be given the `FLT` tag. A global variable with an `FLT` tag is assumed to hold a floating point value, or if it holds a function its result is assumed to be a floating point value.

(2) An expression has the FLT tag if it is a name declared with the FLT tag, a floating point constant or it has one of the following leading operators FLOAT, #ABS, #*, #/, #MOD, #+, #- and #->.

(3) If one of the operators ABS, *, /, MOD, +, -, =, ~=, <, <=, >, >=, :=, ABS:=, *:=, /:=, MOD:=, += or -= has an operand with the FLT tag, the operator is replaced by the corresponding floating point version. The operator -> is replaced by #-> if its second or third operand has the FLT tag.

(4) Expressions are evaluated in either FLT or non-FLT mode. Expressions are evaluated in FLT mode if they are operands of FIX, #ABS, #*, #/, #MOD, #+, #-, #=, #~=, #<, #<=, #> or #>=. The second and third operands of #-> and the second operand of #:=, #ABS:=, #*:=, #/:=, #MOD:=, #+:= and #-:= are evaluated in FLT mode. Expressions giving the values of static, manifest and local variable names with the FLT tag are also evaluated in FLT mode. All other expressions are evaluate in non-FLT mode.

(5) If the leading operator of an expression evaluated in FLT mode is one of ABS, *, /, MOD, +, - or -> it is replaced by the floating point version. If an integer constant is evaluated in FLT mode it is replaced by the corresponding floating point constant.

(6) If a function is declared in the scope of a global of the same name they must either both have FLT tags or neither should have that tag.

These rules are applied repeatedly until there is no further change. As an example, the following function:

```
LET f(a, FLT b, c) = VALOF
{ a, b, c := 1/2, 1/2, 2 * (a + b)
  RESULTIS a + 2.0 * c + (0|#x3840000)
}
```

is automatically converted to:

```
LET f(a, FLT b, c) = VALOF
{ { a := 1/2; b #:= 1.0#/2.0; c #:= 2.0 #* (a #+ b) }
  RESULTIS a #+ 2.0 #* c #+ (0|#x3840000)
}
```

Notice that the user almost never needs to use # to specify floating point operations. Note also that the expression (0|#x3840000|) is a way to protect an expression (#x38400000) from being evaluated in FLT mode. See com/xcmpltest.b and bcpl4raspi.pdf from my home page for examples of the use of the FLT feature.

To see why the name in a vector declarations may not be given the FLT tag, consider LET v = VEC 5. This initialises v with a pointer to a vector with 6

elements and the expression `v+1` would point to the element at subscript position one. Had `v` been given the `FLT` tag, `v+1` would have been automatically converted to `v#+1.0` which is clearly meaningless.

The `FLT` tag is not permitted to qualify `FOR` loop control variables since, due to floating point truncation and rounding errors, the number of iterations of a loop such as `FOR FLT x = 0.0 TO 10.0 BY 0.1 DO...` is properly defined since 0.1 cannot be represented precisely by a floating point number.

2.8 The `objline1` Feature

If a file named `objline1` is found in the current directory or the other directories searched by `GET` directives, its first line is copied as the first line of the compiled Cintcode module. This will typically put a line such as:

```
#!/usr/local/bin/cintsys -c
```

as the first line of the compiled object module. This line is ignored by the CLI but under Linux it allows Cintcode programs to be called directly from a Linux shell. If `objline1` cannot be found no such line is inserted at the start of the object module.

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 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:y, 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 66.

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 58.

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

To aid debugging, the entire Cintcode memory is initialised to `deadcode`. Typically `deadcode=#xDEADC0DE`.

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`.

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

Constants of the form `fl_...` are mnemonics for the floating point operations performed by the call `sys(Sys_flt, op, ...)` as described near page 76.

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 66.

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. The default value is 0 but can be incremented using the `-v` option. Larger values of `boottace` generate more tracing information.

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 169.

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_days CIN:y, POS:y, NAT:n

This field holds the number of days since 1 January 1970. It is updated by the interpreter normally within a milli-second of the date changing.

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 accesible to the debugger and to programs that inspect Cintcode memory dumps.

rtn_dcountv CIN:y, POS:y, NAT:n

This holds a pointer to the debug count vector. These counters can be incremented by calls of the form `sys(Sys_incdcount, n)` or by similar calls in C within the Cintcode interpreter. The zeroth element of `dcountv` holds it upper bound which is typically 511.

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. Some devices are handled by polling in the interpreter thread based on the count of Cintcode instructions obeyed. Currently the clock (device `-1`) and `ttyout` (device `-3`) are handled in this way. This improved the performance of output to the screen and causes the clock to have a resolution of about 1 milli-second although the actual clock precision is usually limited by the underlying operating system.

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`. The environment variable held in `envlist` are distinct from those such as `BCPLROOT` held by the underlying operating system but have a similar purpose.

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". See Section 3.6 for more details.

rtn_idletcb CIN:n, 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`. The task number of the IDLE task is zero but it is not a proper task and does not have an entry in the task table. The Cintpos scheduler gives it control when all other tasks are suspended.

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_gvecsize CIN:y, POS:y, NAT:n

This field holds the size of global vectors created from now on. The default size is now 2000 but it can be set using the `-g` argument when entering `cintsys` or `cintpos`.

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_laststst 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_msecs CIN:y, POS:y, NAT:n

This field holds the number of milli-seconds since midnight. It is repeatedly updated by the interpreter and its value is normally correct to the nearest milli-second.

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". See Section 3.6 for more details.

rtn_quietflag CIN:y, POS:y, NAT:n

This field holds `TRUE` if `cintsys` or `cintpos` was entered with the `-q` option. This implies that the system is running in quiet mode.

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". See Section 3.6 for more details.

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". See Section 3.6 for more details.

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_system CIN:y, POS:y, NAT:y

This rootnode field holds 1 when Cintsys is running or 2 when Cintpos is running. It is otherwise zero.

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 153.

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

Under Cintpos, this rootnode field holds the Cintpos task table. The zeroth entry is the table's 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. If not in Cintpos this field holds zero. If not in Cintpos this field holds zero.

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

This is the upperbound of the rootnode. It value is typically 80.

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 holds 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 buffer of bytes, allocated by `getvec`, 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 the stream. It is given the SCB as its argument and it returns `TRUE` if the call is successful. It otherwise returns `FALSE` with an error code in `result2`.

scb_fd scb_fd1 CIN:y, POS:y, NAT:n

These SCB fields hold a machine dependent file or mailbox descriptor which is often implemented as a native machine code address. On some machines machine addresses are 64 bits long and so cannot be held in a variable of a 32 bit version of BCPL. So the BCPL system allocates two consecutive words to hold such values. In order to allow the same header files be used for 32 and 64 bit BCPL and for 32 and 64 bit machines, two words are allocated even when one word would be sufficient. How a machine addresse is packed in a pair of BCPL words is implementation dependent except that null pointers cause both words to be set to zero. This mechanism is used whenever native machine addresses are held in BCPL variables.

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. This field is updated every time a character is transferred to or from a stream.

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 an 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 holds 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 too short.

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

This word occurs near the start of a section of code just before a compiled string of 11 character representing the section name if a section name is specified in the source code. If the name is less than 11 characters long it is padded with spaces at the end. If the name has more than 11 characters, the string consists of the first and last five separated by a prime (’). Typically `sectword=#x0000FDDF`.

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

As an aid to debugging, 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 73.

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 80.

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

This constant no longer exists since time is now measured in milli-seconds (and dates in days). In both Cintsys and Cintpos, delays measured in milli-seconds can be achieved using `delay(msecs)` and delays until a specified absolute time can be done using `delayuntil(days, msecs)`. Under Cintpos, the clock device now takes packets that specify absolute times (in days since 1 January 1970 and milli-second since midnight) for their release. For example, `sendpkt(notinuse, -1, 0, 0, 0, days, msecs)` will resume execution when the time specified by `days` and `msecs` is reached. The second argument (`-1`) specifies the clock device.

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`.

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.

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

This holds the list of currently existing coroutines.

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

This is a variable used by command language interpreters.

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

This points to the currently executing coroutine.

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

This is a string holding the name of the current working directory.

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

This variable is in global zero and holds the size of the global vector. Its value is normally 1000.

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

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

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

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

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

Under Cintpos when running in multi-event mode, `pktlist` contains mapping from packets to their corresponding coroutines.

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

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

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

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

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

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

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

This points to the rootnode.

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.

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

Under Cintpos this is the identifier of the currently executing task. It in not available under Cintsys.

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

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

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.

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 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 := appendstream(*scb*) CIN:y, POS:y, NAT:y

This function sets the position of stream *scb* to the end so that anything written to the stream will be appended. The result is FALSE if *scb* is not an inout stream or cannot be positioned for other reasons. It returns TRUE otherwise.

ch := binrdch() CIN:y, POS:y, NAT:y

This call behaves like rdch() but does not skip over carriage return ('*c') characters.

ch := binwrch(ch) CIN:y, POS:y, NAT:y

This call behaves like wrch(ch) but does treat ch as a special character and so does not call deplete at the end of lines and does not insert carriage return ('*c') characters.

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. The definition of callco is in blib.b and is as follows.

```
LET callco(cptr, a) = VALOF
{ IF cptr!co_parent DO abort(110)
  cptr!co_parent := currco
  RESULTIS changeco(a, cptr)
}
```

callco always leaves the global currco is set to point to the target coroutine. This is done by the Cintcode instruction CHGCO invoked by changeco.

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 := changeco(*val*, *cptr*) CIN:y, POS:y, NAT:y

This function is used in the functions that implement the coroutine mechanism. In callco, resumeco and cowait, it causes the current coroutine to become suspended and store *cptr* in the global currco before giveing control to the specified coroutine. Strangely, execution continues just after the call of changeco but with the P pointer pointing to the stack frame of the function that caused the target coroutine to become suspended. The call of changeco in each of callco, cowait and resumeco is immediately followed by a RETURN statement which causes the corresponding function to

return with result *val*. The only other use of `changeco` is in `createco`. This is more subtle but can be understood by looking at the description of `createco` on page 58.

res `:= changepri(`*taskid, pri*`)` CIN:n, POS:y, NAT:n

This Cintpos function attempts to change the priority of the specified task to *pri*. It moves the specified task control block to its new position in the priority chain. If the specified task is runnable and of higher priority than the current task, it is given control leaving the current task suspended in RUN state. The result is non zero if successful, otherwise it is zero with `result2` set to 101 if *taskid* is invalid or to 102 if the change would cause two tasks to have the same priority.

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

This function simply calls `start(`*arg*`)` 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	Binary 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. The definition of cowait is in blib.b and is as follows.

```
LET cowait(a) = VALOF
{ LET parent = currco!co_parent
  currco!co_parent := 0
  RESULTIS changeco(a, parent)
}
```

cowait always leaves the global currco is set to point to the resumed coroutine. This is done by the Cintcode instruction CHGCO invoked by changeco.

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.

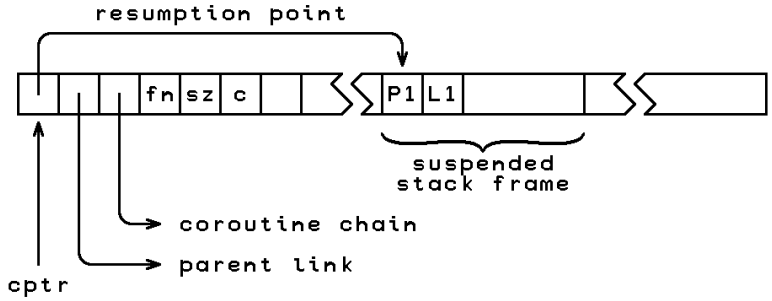


Figure 3.1: A coroutine stack

The resumption point is P pointer belonging to the function 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

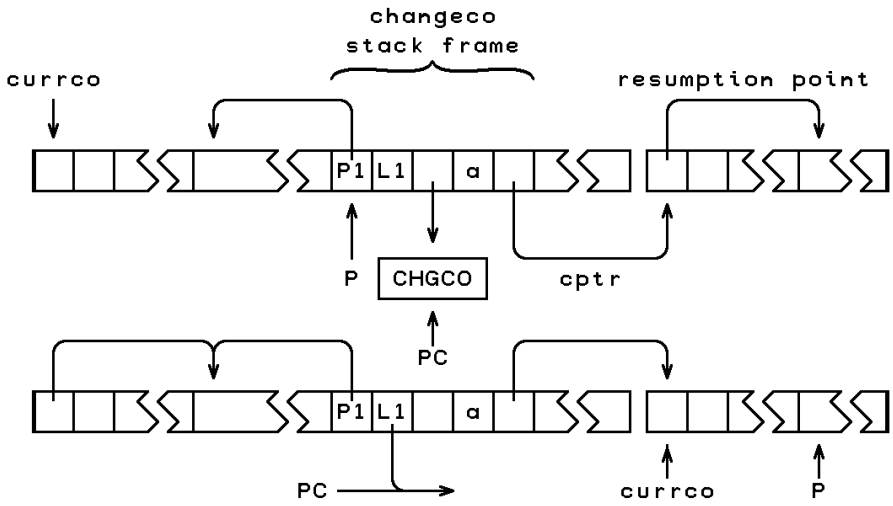


Figure 3.2: The effect of `changeco(a, cptr)`

`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 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` used by `cintsys` and `cintpos` and its body consists of the single Cintcode instruction `CHGCO`. 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. In the native code version of BCPL `changeco` is defined in `mlib.s`

The definition of `createco` is in `blib.b` and is as follows.

```
LET createco(fn, size) = VALOF
{ LET c = getvec(size+6)
  UNLESS c RESULTIS 0
  FOR i = 6 TO size+6 DO c!i := stackword

  c!0 := c<<B2Wsh // resumption point
  c!1 := currco   // parent link
  c!2 := colist   // colist chain
  c!3 := fn       // the main function
  c!4 := size     // the coroutine size
  c!5 := c        // the new coroutine pointer

  colist := c  // insert into the list of coroutines

  changeco(0, c)

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

The function `createco` creates a new coroutine by allocating its stack by the call

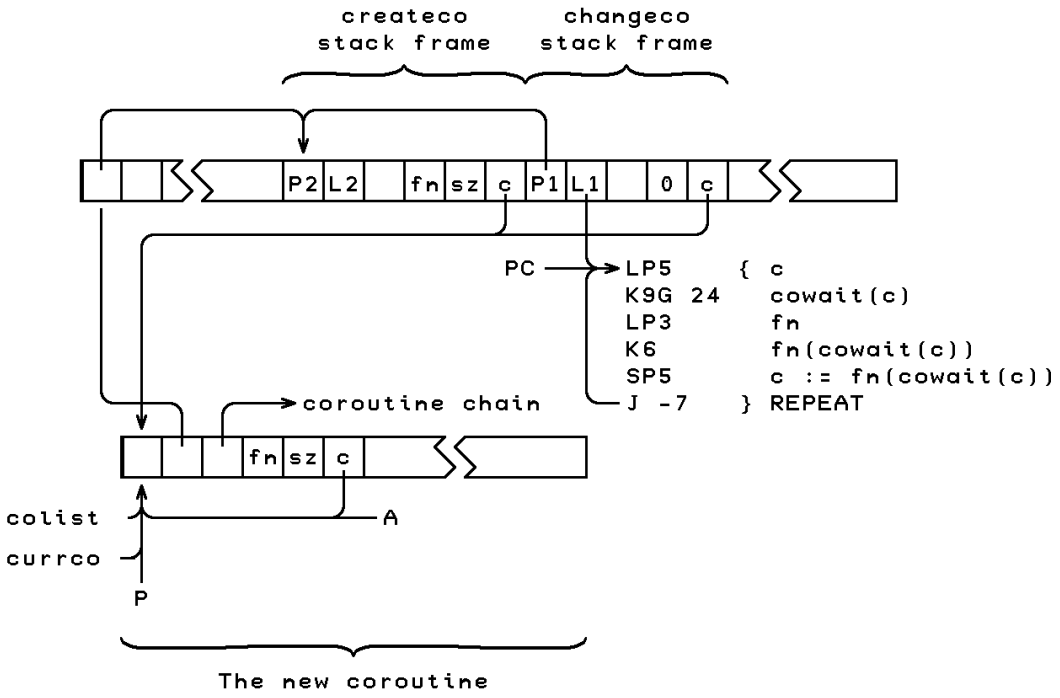


Figure 3.3: The state just after `changeco(0,c)` in `createco`

`gevec(size+6)`. The variable `c` holds a pointer to the new coroutine stack and, as can been seen, its first six words are initialised to hold system information, as follows.

- `c!0` resumption point
- `c!1` parent link
- `c!2` colist chain
- `c!3` *fn* – the main function
- `c!4` *size* – the coroutine size
- `c!5` c – the new coroutine pointer

The coroutine list `colist` is also set to `c`.

The call `changeco(0, c)` causes the P pointer to be set to `c!0` which has been initialied to the address of the base of the new coroutine stack. Execution continues just after the call, namely at the REPEAT loop in the body of `createco`, but in the coroutine environment of the newly created coroutine. The compiled code for this loop will assume `fn`, `size` and `c` reside in positions 3, 4 and 5 relative to P, ie in memory locations `c!3`, `c!4` and `c!5` so execution behaves as (naively) expected. The first time `cowait(c)` is called in this loop, execution returns from `createco` with the result `c` pointing to the newly created coroutine.

When control is next transferred 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.

To help to fully understand the subtle effect of effect of `changeco(0,c)`, look at figure 3.3 which shows the state just after `changeco` transfers control to the newly created coroutine. At this moment the newly created coroutine immediately suspends it self by calling `cowait` in the loop:

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

at the end of `createco`.

devid `:= createdev(`*dcb*`)` CIN:n, POS:y, NAT:n

This Cintpos function creates a device using the first available slot in `devtab`. The device control block *dcb* must have already been initialised and linked to its device driver. If successful it returns a negative value identifying the device. On failure it returns zero with `result2` set to 104 if the `devtab` is full, or to 106 if device initialisation failed.

res `:= createtask(`*seglist, stsize, pri*`)` CIN:n, POS:y, NAT:n

This Cintpos function creates a task using the first free slot in the task table. It allocates space for the new task control block (TCB) and a copy of the specified segment list, and initialises them both. It inserts the new TCB in priority chain of tasks and returns the id of the newly created task if successful. It is left in DEAD state with no stack or global vector and no packets on its work queue. If there is an error, it returns zero with `result2` set to 102 if there is already a task with priority *pri*, or to 103 if there is insufficient memory or to 105 if the task table is full. A segment list is a small vector whose zeroth element holds its upperbound and whose other elements hold lists of sections of code typically loaded by `loadseg`.

`datstamp(`*datv*`)` CIN:y, POS:y, NAT:y

This sets *datv*`!0` to the number of days since 1 January 1970, and *datv*`!1` to the number of milli-seconds since midnight, and for compatability with the older version of datstamp *datv*`!2=-1` indicating the new date and time format is being used.

`dat_to_string(`*datv, v*`)` CIN:y, POS:y, NAT:y

This call causes the time stamp in *datv* to be converted to three strings *v*, *v*`+5` and *v*`+10`. The string at *v* is set to the date in the form `dd-mmm-yyyy`. The string at *v*`+5` is set to the the current time in the form hh:mm:ss, and the string at *v*`+10` is set to the day of the week. The upper bound of *v* should be at least 14 to be safe. The time stamp is typically obtained by a call of `datstamp(datv)` which sets `datv!0` to the number of days since 1 January 1970, `datv!1` to the number of milli-seconds since midnight and `datv!2` to -1 indicting that the new date and time format is being used.

`delay(`*msecs*`)` CIN:y, POS:y, NAT:y

This call suspends execution for at least *msecs* milli-seconds. Under Cintpos, this is achieved by sending a suitable packet to the clock device (using `sendpkt`) and waiting for it to be returned.

`delayuntil(`*days*`, `*msecs*`)` CIN:y, POS:y, NAT:y

This call suspends execution until the specified date and time is reached. *days* specifies the date as the number of days since 1 January 1970 and *msecs* is the number of milli-seconds since midnight. Under Cintpos, the delay is achieved by sending a suitable packet to the clock device (using `sendpkt`) and waiting for it to be returned.

`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.

dcb `:= deletedev(`*devid*`)` CIN:n, POS:y, NAT:n

This Cintpos function closes down the specified device and deallocates it device identifier, but it does not return its device control block (DCB) to free store. It returns any packets still on its work queue to the requesting tasks with both the `pkt_res1` and `pkt_res2` fields set to `-1`. If successful, it returns the DCB of the deleted device. On failure, it returns zero with `result2` set to 101 indicating that *devid* was invalid. If any of the released packets cause a higher priority task to become runnable, the control passes to the highest priority one leaving the current task suspended in RUN state. The clock device has identifier `-1` and is permanently resident and cannot be deleted.

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

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

res `:= deleteself(`*pkt*`, `*seg*`)` CIN:n, POS:y, NAT:n

This Cintpos function first calls `qpkt` to return the packet if *pkt* is non zero, then calls `unloadseg(`*seg*`)`~ if *seg* is non zero, before deleting the current task. This function is defined in `klib` since it would be unsafe for it to be in a segment that may be unloaded while it is being executes. It returns a non zero value if successful but, of course, this value will never be seen! On failure, it return zero with `result2` set to 108 indicating that the current task is not deletable.

res `:= deletetask(`*taskid*`)` CIN:n, POS:y, NAT:n

This Cintpos function attempts to delete the specified task which must have an empty work queue and be either the current task or in DEAD state. Its task control block (TCB) is unlinked from the priority chain and removed from `tasktab`. Finally its segment list and the TCB itself returned to free store. It returns a non zero value if successful. On failure, it returns zero with `result2` set to 101 if *taskid* is invalid, or to 108 if the task is not deletable.

res `:= dqpkt(`*id*`, `*pkt*`)` CIN:n, POS:y, NAT:n

This Cintpos function attempts to dequeue the given packet from the task or device specified by *id*. If not found there, it may have already been returned to the current task so its work queue is searched. The result is the id of the task or device whose work queue contained the packet. If there is an error, the result is zero with `result2` set to 101 for invalid id or 109 if the packet was not found. The id field of the packet is set to the id of the task or device whose work queue contained the packet provided that this is not the id of the current task.

`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)`.

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

This function opens an output stream specified by the file name *name* in append mode causing all output to be appended onto the end of the file. If the file name is relative and the prefix string is set, it is prepended to the name before attempting to open the stream. If the file does not exist a zero length file of the given name is created. If there is an error the result is zero.

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`. During matching all letters are converted to uppercase, but this convention may change in future.

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 file name is relative and the prefix string is set, it is prepended to the name before attempting to open the stream. If the stream cannot be opened the result is zero. See Section 3.3.2 for information about the treatment of filenames.

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

This function opens a stream specified by the device or file name *name* that can be used for both input and output. If *name* is the string `"*"` then output is normally to the screen and input comes from the keyboard. If the file name is relative and the prefix string is set, it is prepended to the name before attempting to open the stream. 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 file name is relative and the prefix string is set, it is prepended to the name before attempting to open the stream. If the stream cannot be opened, the result is zero. See Section 3.3.2 for information about the treatment of filenames.

res := get_record(*v*, *recno*, *scb*) CIN:y, POS:y, NAT:y

This attempts to read the record numbered *recno* from the file whose stream control block is *scb* into the vector *v*. The record length must have been set already by a call of setrecordlength. If get_record is successful it returns TRUE, otherwise it returns FALSE possibly because the end of file was reached before the whole record had been read.

v := getlogname(*logname*) CIN:y, POS:y, NAT:y

This function searches the list of logical variables held in the root node and returns its value if found, otherwise it returns zero.

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

This function allocates 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*.

res := hold(*taskid*) CIN:n, POS:y, NAT:n

This Cintpos function sets the HOLD bit in the task control block of the specified task. It returns a non zero value if successful. If there is an error, it returns zero with result2 set to 101 if *taskid* was invalid, and 110 if the specified task was already in HOLD state. If the task holds itself control is given to next lower priority runnable task.

cptr := initco(*fn*, *size*, *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*) CIN:y, POS:y, NAT:y

This function provides a convenient method of creating and initialising 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)
  result2 := 0
  IF cptr DO result2 := callco(cptr, @a)
  RESULTIS cptr
}
```

A coroutine with main function *fn* and given size is created and, if successful, it is

initialised by `callco(`*cptr*`, @`*a*`)`. Thus, *fn* should expect a vector containing up to 11 elements. 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 function returns the number of Cintcode instructions 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.

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

This call returns the current stack frame pointer 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 80 for details.

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

This call causes execution to resume at label *L* in the body of a function or routine 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 function or routine can be performed using the `GOTO` command, so `level` and `longjump` are only needed for non local jumps.

res `:= memoryfree()` CIN:y, POS:y, NAT:n

This function checks that the free store chain is valid, outputting a error message and calling `abort(999)` if not. If the chain is valid, it returns the current number of unused words, and sets `result2` to the memory size. This function can assist debugging and helps with the detection of space leaks.

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. The result is also undefined if any of a, b or c is the largest negative integer.

This version of `muldiv` is defined in the hand written Cintcode library `syslib` and invokes the `MDIV` Cintcode instruction which is implemented efficiently. The older version is invoked by `sys(Sys_muldiv,a,b,c)` and uses binary long division implemented in C. Both versions are believed to produce identical results except possibly when `c=0`.

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)
```

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.

res `:= note(`*scb*`, `*posv*`)` CIN:y, POS:y, NAT:y

If *scr* is a file stream, this function sets *posv*`!0` and *posv*`!1` to the current block number and position within that block. For RAM streams, *posv*`!0` and *posv*`!1` are set to zero and the position within the stream buffer. The result is `TRUE` if *scb* is a file or RAM stream, and `FALSE` otherwise.

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. The directories specified by the shell variable are separated by either semicolons or colons, although under Windows only semicolons are allowed. If the prefix string is non null and the filename, possibly prefixed by a directory name, is relative then the prefix string is prepended before the file is opened. If the file cannot be opened `pathfindinput` returns zero.

res `:= point(`*scb*`, `*posv*`)` CIN:y, POS:y, NAT:y

This function sets the position of stream *scb* to that specified in *posv* whose elements are *scb*`!0` the block number and *scb*`!1` the byte position within the block. If the new position is in a different block the contents of the buffer buffer may have to be written out and data from the new block read in. `point` may therefore fail if the stream was not opened using `findinput` or `findinoutput`. It returns `TRUE` if successful, even if positioned just after the last block of the file, ie `block=lblock+1` and `pos=end=0`. It returns `FALSE`, otherwise, possibly because the stream is not pointable or the *posv* is out of range. It is advisable to test the result of `point` every time it is used.

For RAM streams *posv*`!0` should be zero and *posv*`!1` should be a position in the buffer (which is entirely held in RAM).

res `:= put_record(`*v*`, `*recno*`, `*scb*`)` CIN:y, POS:y, NAT:y

This attempts to write a record numbered *recno* to the file whose stream control block is *scb* taking data from the vector *v*. The record length must have been set already by a call of `setrecordlength`. If put_record is successful it returns `TRUE`, otherwise it returns `FALSE`. If the last record of a file has number *n*, it is permissible to extend the file by writing record *n+1*, but not one with a larger record number.

res := qpkt(*pkt*) CIN:n, POS:y, NAT:n

This Cintpos function queues the given packet on the end of the work queue of the destination task or device (specified by pkt_id!*pkt*). If this field is positive it refers to a task, if it is -1 it refers to the clock device and other negative values refer to other devices. If the packet is queued successfully this field is updated to hold the current task's identifier and the result is non zero, otherwise the result is zero with result2 set to 101 if the destination id is invalid, and to 111 if pkt_link was not equal to notinuse (=-1). If the destination was a runnable task of higher priority than the current one, then the current task immediately becomes suspended in RUN state and control is given to the destination, otherwise the current task continues to run normally. Interaction with the resident Cintpos devices is described in Chapter 6.

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) MOD 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 arguments and these are easily read using rdargs.

The first argument (*keys*) specifies the argument format. The second and third arguments provide a vector (*argv*) with a given upper bound (*upb*) into which the decoded arguments will 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.

The string *keys* holds the list of argument keywords separated by commas (,). Alternative keywords for a given argument are separated by equal signs (=). The expected number of arguments is one more than the number of commas in the key string. If rdargs returns successfully, this number of elements at the start of *argv* will hold the decoded arguments.

Arguments can have qualifiers of the form /A, /K, /N, /S and /P. The qualifier letters can be in either upper or lower case. The qualifier /A means that the argument must be given. /K means that, if the argument is given, it must include its keyword. /N specifies that the argument must be a number. /S indicates that the argument is a switch parameter set to TRUE by its keyword. /P indicates that a prompt will be given for the argument if it has not already been set. Prompting only happens if the currently selected input and output streams are both connected to an interactive terminal. If the prompt is for a switch argument (/S) it expects a yes/no response. Typing yes or y is treated as yes, any other response is treated as no. If rdargs returns successfully *argv*!0, *argv*!1 etc will hold the arguments settings. A setting of

zero means the argument was not given. A setting of `-1` means the argument was a switch set the `TRUE`. Otherwise, if `/N` was specified the setting will point to a word in *argv* where the decoded integer is stored. If a `/N` was not specified, the setting will be a BCPL string with its characters packed into *argv*. Note that an argument should not have both `/N` and `/S` specified.

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=AS/K,DATA/N/P,N/S", argv, 50) DO
{ writef("Bad arguments for: FROM/A,TO=AS/K,DATA/N/P,N/S*n")
  ...
}
```

In this example, there are four possible arguments and their values will be placed in the first four elements of `argv`. The first argument has keyword `FROM` and must receive a value because of the qualifier `/A`. The second has alternative keywords `TO` and `AS` with qualifier `/K` that insists the argument is introduced by one of its keywords. The third argument has the qualifiers `/N` and `/P` indicating that it expects a number and that it will be prompted for if not already given, and the last argument has the qualifier `/S` indicating that it is a switch that can be set by the presence of its keyword.

Table 3.4 shows the values in placed in `argv` and the result when the call:

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

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

Arguments	argv!0	argv!1	argv!2	argv!4	Result
abc TO xyz	"abc"	"xyz"	0	0	~=0
to xyz from abc	"abc"	"xyz"	0	0	~=0
as xyz abc n	"abc"	"xyz"	0	-1	~=0
abc xyz	-	-	-	-	=0
"from" to "to"	"from"	"to"	0	0	~=0
abc data 123 to "to"	"abc"	"to"	->123	0	~=0
data 123 to junk	-	-	-	-	=0

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

To consolidate your understanding of `rdargs`, try compiling and running the program: `bcplprogs/tests/tstrdargs.b`.

res `:= rdargs2(`*keys1, keys2, argv, upb*`)` CIN:y, POS:y, NAT:y

This function behaves just like `rdargs`, specified above, except it uses key data that

is the concatenation of strings `keys1` and `keys2` thus allowing the key data to have up to than 510 characters.

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 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
`=`		5
`;`		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 double quote character.

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

This reads an optionally signed floating point number 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.

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 := recordnote(*scb*) CIN:y, POS:y, NAT:y

This call returns the number of the record containing the character pointed to by the file position pointer of stream *scb*. The record length must have already been set by a call of setrecordlength. The result is -1 if the stream is not suitable.

res := recordpoint(*scb*, *recno*) CIN:y, POS:y, NAT:y

This call sets the file position pointer of stream *scb* to point to the first byte of the record whose number is *recno*. The record length must have already been set by a call of setrecordlength. It returns TRUE if successful and FALSE otherwise.

res := release(*taskid*) CIN:n, POS:y, NAT:n

This Cintpos function will clear the HOLD bit in the specified task thus making it potentially runnable. It returns a non zero value if successful. If the specified task does not exist it returns zero with 101 in result2. If the released task has higher priority and is runnable it gains control leaving the current task suspended in RUN state. This function is also called unhold.

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.

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. The definition of resumeco is in blib.b and is as follows.

```
LET resumeco(cptr, a) = VALOF
{ LET parent = currco!co_parent
  currco!co_parent := 0
  IF cptr!co_parent DO abort(111)
  cptr!co_parent := parent
  RESULTIS changeco(a, cptr)
}
```

resumeco always leaves the global currco is set to point to the resumed coroutine. This is done by the Cintcode instruction CHGCO invoked by changeco.

res := rewindstream(*scb*) CIN:y, POS:y, NAT:y

This function set the position of stream scb to its start, returning TRUE if successful, and FALSE otherwise.

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. The character is echoed to the screen unless the system is running in quiet mode.

`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*`,`*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

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.

res `:= setbit(`*bitno*`,` *bitvec*`,` *state*`)` CIN:y, POS:y, NAT:y

This function sets the specified bit in *bitvec* to 1 or 0 depending on whether *state* is `TRUE` or `FALSE`, respectively. It returns a non-zero value if and only if the previous setting of the bit was a one. See `testbit` below.

res `:= setflags(`*taskid*`,` *flags*`)` CIN:n, POS:y, NAT:n

This Cintpos function sets the specified flags in the task control block of the specified task. If successful it returns a non zero value with `result2` set to the previous setting of the flags field, otherwise it returns zero with `result2` set to 101 indicating that *taskid* was invalid. For more information about flags see `testflags` described below.

oldlogname `:= setlogname(`*logname*`,` *logvalue*`)` CIN:y, POS:y, NAT:y

This sets the value of logical variable *logname* to the *logvalue*. By convention *logvalue* should be a string. The list of logical name-value pairs is held in the root node.

prevseed `:= 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.

`sxpushval(`*sxv*`,` *val*`)` CIN:y, POS:y, NAT:y

This pushes value *val* into the self expanding vector *sxv*. *sxv* points to the two word control block of the self expanding vector. Initially both elements must be zero. When non empty *sxv*`!1` will be a vector, `v` say, containing the elements with `v!0` will be the subscript of the latest element in `v`. *sxv*`!0` holds the upper bound of `v`. When the self expanding vector needs more space, it allocates a new vector `v` using `newvec` freeing the previous one after copying its elements into the new one. Clearly pointers to elements

of v may become invalid after any call of sxpushval. When the self expanding vector is no longer needed, freevec(v) must be called.

srchwk(*tcb*) CIN:n, POS:y, NAT:n

This function is the Cintpos scheduler which is normally only called from within one of the klib library functions or from the interrupt service routine. Its argument points to the highest priority task control block that could possibly run. It searches down the priority chain from this point until it finds the highest priority runnable task. After setting the globals tcb and taskid appropriately, it gives this task control using a call of sys(Sys_rti,...).

res := stackfree(hwm) CIN:y, POS:y, NAT:n

If hwm=TRUE, this function returns the number of unused stack words above the high water mark, otherwise it returns the number of words between the current stack frame pointer and the end of stack. In either case it sets result2 to the stack size.

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

This function is, by convention, the main function 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 function 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

res := stepstream(*scb, n)* CIN:y, POS:y, NAT:y

This function advances the position of stream *scb* by *n* words, returning TRUE if successful, and FALSE otherwise.

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

This function is provided to stop the execution of the current command running under control of the CLI. The arguments code and reason are placed in the CLI globals cli_returncode and cli_result2 where they can be inspected by commands such as if and why.

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. The result is negative or zero if str%1='-'. This function is no longer recommended, string_to_number should be used instead.

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

This attempts to set result2 to the integer represented by the string *str*. It returns TRUE is successful and FALSE otherwise. The following are examples of acceptable strings: "'A'", "123", "-99", "+63", "#377", "-#x7FF" and "+#b1011011".

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

On non standard machines such as the GP2X gaming machine there are buttons that can be pressed. This call returns a bit pattern indicating which buttons are currently pressed.

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

This makes the call `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).

res `:= sys(Sys_callc, c_name2ipaddr,` *`a1`*`)` CIN:y, POS:y, NAT:y

The name or dotted decimals of a host is given in *a1* and the result is its IP address or -1 if there is an error.

res `:= sys(Sys_callc, c_name2port,` *`a1`*`)` CIN:y, POS:y, NAT:y

The name or decimals of a port is given in *a1* and the result is its IP address or -1 if there is an error.

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

The result is the file descriptor of a new socket or -1 if there is an error.

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

The file descriptor of a socket is given in *a1*. Id *a2*=`1` the specified socket may be reused. If there is an error the result is -1.

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

This sets the send buffer size of socket *a1* to *a2* bytes. If there is an error the result is -1.

res := sys(Sys_callc, c_setrcvbufsz, *a1*, *a2*) CIN:y, POS:y, NAT:y

This sets the receive buffer size of socket *a1* to *a2* bytes. If there is an error the result is -1.

res := sys(Sys_callc, c_bind, *a1*, *a2*, *a3*) CIN:y, POS:y, NAT:y

This bind socket *a1* to remote IP address *a2* and remote port *a3*. If there is an error the result is -1.

res := sys(Sys_callc, c_tcpconnect, *a1*, *a2*, *a3*) CIN:y, POS:y, NAT:y

This make a TCP/IP connection through socket *a1* to remote IP address *a2* and remote port *a3*. If there is an error the result is -1.

res := sys(Sys_callc, c_tcplisten, *a1*, *a2*) CIN:y, POS:y, NAT:y

This causes socket *a1* to wait for a TCP/IP connection to be requested by a remote host. The maximum number of connections waiting to be accepted is given in *a2*. If there is an error the result is -1.

res := sys(Sys_callc, c_tcpaccept, *a1*) CIN:y, POS:y, NAT:y

This accepts a TCP/IP connection through socket *a1*. The result is the socket number to be used for this connection or -1 if there is an error.

res := sys(Sys_callc, c_tcpclose, *a1*) CIN:y, POS:y, NAT:y

This closes socket *a1*. The result is -1 if there is an error.

res := sys(Sys_callc, c_fd_zero, *a1*) CIN:y, POS:y, NAT:y

This clear every bit in the bit vector *a1*. The result is -1 if there is an error.

res := sys(Sys_callc, c_fd_set, *a1*, *a2*) CIN:y, POS:y, NAT:y

This sets bit *a1* in the bit vector *a2*. The result is -1 if there is an error.

res := sys(Sys_callc, c_fd_isset, *a1*, *a2*) CIN:y, POS:y, NAT:y

This inspects bit *a1* in the bit vector *a2*. The result is 1 if the bit was set and 0 otherwise.

res := sys(Sys_callc, c_fd_select, *a1*, *a2*, *a3*, *a4*, *a5*) CIN:y, POS:y, NAT:y

This inspects bit *a1* in the bit vector *a2*. The result is 1 if the bit was set and 0 otherwise.

The number of the bits to test is in *a1*. The bit vector identifying read sockets of interest is in *a2*, The bit vector identifying write sockets of interest is in *a3*, The bit vector identifying other sockets of interest is in *a4*. A pointer to two words holding the timeout in seconds and microseconds is in *a5*. The result is the number of sockets that can now be read or written to, or 0 if the timeout period has elapsed before any sockets are ready. A result of -1 indicate an error.

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_datstamp, *datv*) CIN:y, POS:y, NAT:y

This sets *datv*!0 to the number of days since 1 January 1970, and *datv*!1 to the number of milli-seconds since midnight, and for compatability with the older version of datstamp *datv*!2=-1 indicating the new date and time format is being used.

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

In both Cintsys and Cintpos this call suspends Cintcode execution until the time period has elapsed. It is normally better to use the library functions delay(msecs) or delayuntil(days, msecs).

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

This deletes the file whose name is given by name. See page 94 for information about the treatment of file names.

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

This is used in Cintpos to send commands from the interpreter thread to Cintpos device threads.

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

This call will dump the whole of Cintcode memory to the file DUMP.mem in a compacted form that is typically inspected by either the commands dumpsys or dumpdebug. By convention, context = 1 if SIGINT has been received, context = 2 if SIGSEGV has been received, context = 3 if the dump was caused by BOOT detecting a fault, context = 4 if the dump by the user call sys(Sys_quit, -2), context = 5 if the dump by a non zero return code from the interpreter, context = 6 if the dump by the D command in the interactive debugger.

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

This sets the elements of the time stamp vector *datv* to represent the date and time of the last modification of the file given by *name* returning TRUE if successful. The first element *datv*!0 holds the number of days since 1 January 1970, *datv*!1 is the number of milli-seconds since midnight and *datv*!2=-1 indicating that the new date format is being used. If the file does not exist the call returns FALSE and setting the three elements of *datv* to 0, 0 and -1, respectively.

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_flt, *op* ,...) CIN:y, POS:y, NAT:y

This call provides all the floating point operations available to BCPL. The required

operation is specified by *op* normally using a manifest constant (declared in `libhdr`) such as `fl_mk`, `fl_add` or `fl_sin`. All such operations are described below. BCPL floating point numbers must fit in BCPL words and so are typically only 32 bits long causing their precision and range to be somewhat limited. On 64-bit implementations of BCPL, floating point numbers are much more precise.

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

This call attempts returns `-1` if all the `Sys_flt` operations are available. It otherwise return zero.

res `:= sys(Sys_flt, fl_mk,` *a*`,` *e*`)` CIN:y, POS:y, NAT:y

This call attempts to return a floating point approximimation to the number $a \times 10^e$ where a and e are signed integers.

res `:= sys(Sys_flt, fl_unmk,` *a*`)` CIN:y, POS:y, NAT:y

This call decomposes the floating point number a returning the signed integer mantissa and leaving the decimal exponent in `result2`. For example, `sys(Sys_flt, fl_unmk, 1234.5678)` might return `12345678` leaving `-4` in `result2`. However, the result may vary depending on the BCPL word length and the floating point representation used.

res `:= sys(Sys_flt, fl_float,` *a*`)`
res `:= sys(Sys_flt, fl_fix,` *a*`)` CIN:y, POS:y, NAT:y

The first call returns a floating point approximation of the integer a, and the second attempts to return the closest integer to the floating point number a.

res `:= sys(Sys_flt, fl_abs,` *a*`)`
res `:= sys(Sys_flt, fl_pos,` *a*`)`
res `:= sys(Sys_flt, fl_neg,` *a*`)`
res `:= sys(Sys_flt, fl_mul,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_div,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_add,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_sub,` *a*`,` *b*`)` CIN:y, POS:y, NAT:y

The first three calls return, respectively, the absolute value of a, the value of a and the negated value of a where a is a floating point number. The last four calls perform floating point multiplication, division, addition and subtraction on their arguments.

res `:= sys(Sys_flt, fl_eq,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_ne,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_ls,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_gr,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_le,` *a*`,` *b*`)`
res `:= sys(Sys_flt, fl_ge,` *a*`,` *b*`)` CIN:y, POS:y, NAT:y

These six calls return `TRUE` if the corresponding floating point comparisons are satisfied. Otherwise the result is `FALSE`.

res := sys(Sys_flt, fl_acos, *a*)
res := sys(Sys_flt, fl_asin, *a*)
res := sys(Sys_flt, fl_atan, *a*) CIN:y, POS:y, NAT:y

These calls return floating point approximations to the arc cosine, arc sine and arc tangent of em a. The argument a is in radians and for `acos` the result is between 0 and π. For `asin` and `atan` it is between $-\pi/2$ and $\pi/2$.

res := sys(Sys_flt, fl_atan2, *y*, *x*) CIN:y, POS:y, NAT:y

This call return the angle in radians between x-axis and the line from the origin to the point with cartesian coordinates (x, y). The result lies between $-\pi$ and π.

res := sys(Sys_flt, fl_cos, *a*)
res := sys(Sys_flt, fl_sin, *a*)
res := sys(Sys_flt, fl_tan, *a*) CIN:y, POS:y, NAT:y

These calls return the cosine, sine and tangent of a.

res := sys(Sys_flt, fl_cosh, *a*)
res := sys(Sys_flt, fl_sinh, *a*)
res := sys(Sys_flt, fl_tanh, *a*) CIN:y, POS:y, NAT:y

These calls return the hyperbolic cosine, sine and tangent of a.

res := sys(Sys_flt, fl_exp, *a*)
res := sys(Sys_flt, fl_log, *a*)
res := sys(Sys_flt, fl_log10, *a*) CIN:y, POS:y, NAT:y

The first call returns an approximation to e^a where e is the base of natural logarithms. The second call return the natural logarithm of a, and the third call returns log to the base 10 of a.

res := sys(Sys_flt, fl_frexp, *a*)
res := sys(Sys_flt, fl_ldexp, *f*, *n*) CIN:y, POS:y, NAT:y

The first call splits a floating-point number (a) into a fraction (f) and exponent (n) such that a is approximately equal to $f \times 2^n$. If possible the absolute value of f will be between 0.5(inclusive) and 1.0(exclusive). The call returns f and stores n in `result2` as an integer. The second call is the inverse of `frexp` returning an approximation to $f \times 2^n$.

res := sys(Sys_flt, fl_modf, *a*)
res := sys(Sys_flt, fl_mod, *x*, *y*) CIN:y, POS:y, NAT:y

The first call returns the fractional part (f) of a storing the integer part (i) as a floating-point number in `result2`. The sign of both f and i is the same as the sign of a and a will equal $i + f$.

The second call returns f such that f has the same sign as x, the absolute value of f is less than the absolute value of y, and there exists and integer k such that $k \times y + f$ equals x.

res := sys(Sys_flt, fl_pow, *a*, *b*)
res := sys(Sys_flt, fl_sqrt, *a*) CIN:y, POS:y, NAT:y

The first call returns an approximation to a^b, and the second call attempts to return the non negative square root of a.

res := sys(Sys_flt, fl_ceil, *a*)
res := sys(Sys_flt, fl_floor, *a*) CIN:y, POS:y, NAT:y

The first call returns the smallest floating-point number not less than a whose value is an exact integer and the second call returns the largest floating-point number not greater than a whose value is an exact integer.

res := sys(Sys_flt, fl_F2N, *s*, *x*) CIN:y, POS:y, NAT:y

This returns the integer part of $s \times x$. This is the scaled fixed point representation of x when s is the scaled value representing 1.0. For example:

```
sys(Sys_flt, fl_F2N, 1_000, -1.234) =>    -1234
```

res := sys(Sys_flt, fl_N2F, *s*, *n*) CIN:y, POS:y, NAT:y

This returns the floating point value corresponding to n/s. This is the floating point number representing the fixed point scaled value n when the scaled number s represents 1.0. For example:

```
sys(Sys_flt, fl_N2F, 1_000, 1_234) = 1.234
```

res := sys(Sys_flt, fl_radius2, *a*, *b*)
res := sys(Sys_flt, fl_radius3, *a*, *b*, c) CIN:y, POS:y, NAT:y

The first call returns the square root of $a^2 + b^2$ and the second returns the square root of $a^2 + b^2 + c^2$. For example:

```
sys(Sys_flt, fl_radius2, 3.0, 4.0)      =>    5.000
sys(Sys_flt, fl_radius3, 1.0, 2.0, 2.0) =>    3.000
```

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

If *ptr* is zero it does nothing, otherwise it returns to free store the space pointed to by *ptr* which must have previously been allocated by sys(Sys_getvec,...). It checks that the block is not already free and attempt to check that it has not been corrupted.

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 a pointer to prefix string which is in space allocated when Cintsys or Cintpos was started. See sys(Sys_setprefix,...) on page 85.

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

This function return the contents of the machine memory location whose word address is *addr* which may be outside the normal range of the Cintcode memory.

res := sys(Sys_gettrval, *count*) CIN:y, POS:y, NAT:n

This returns a value from the low level trace buffer. See Sys_trpush for more details.

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 returns zero if the request cannot be satisfied. A word is allocated just before the start of the vector to hold its size, and several (typically 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 defined 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 operations 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_incdcount, *n*) CIN:y, POS:y, NAT:y

This function increments the specified counter held in the vector pointed to by the field rtn_dcountv in the rootnode. This operation is also available to the interpreter code written in C.

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*. On return the result is a return code indicating why the interpreter returned, and the elements of *regs* hold the final state of the Cintcode registers. These registers are described in the chapter on the design of Cintcode starting on page 193 and the correspondence between the elements of *regs* and the Cintcode registers is given on page 84. The return codes are given on page 84.

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 just returns FALSE.

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

This attempts to load a Cintcode module from file *name* looking first in the current directory. If a valid module is not found there and *name* is a relative file name, it searches through the directories specified by the environment variable whose name

is in the rtn_pathvar element of the rootnode. This name is normally BCPLPATH under Cintsys and POSPATH under Cintpos. See Section 3.6 for more information about environment variables.

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. 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 function "start". The body of start is compiled into one word (7B1C2310) 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

The remaining 4 words contain global initialisation data that is read backwards during global initialisation invoked by sys(Sys_globin,...). 0000001C (=28) is the highest global variable referenced by this section. The pair 00000001 00000024 specifies that the entry point at position 36 is the initial value of global 1, and the next entry (00000000) marks the end of the global initialisation data.

The manifest constants t_hunk, t_reloc, t_end, t_hunk64, t_reloc64, t_end64, t_bhunk, and t_bhunk64 are declared in libhdr for the convenience of programs that generate or read Cintsys and Cintpos object modules. The example above shows t_hunk loading n 32-bit words encoded in hex bytes. Although the BCPL compiler used in both Cintsys and Cintpos generates position independent code and has no need to modify the loaded words of a hunk, other languages may need to perform relocation. This can be done using t_reloc which is followed by a 32-bit word n encoded in hex followed by a further n words which each give the position of a word in the most recently loaded hunk that needs to be modified by the addition of the base address of the hunk. The code t_bhunk is similar to t_hunk only the data words (not the length field) are provided in

binary rather than hex characters. Such hunks are thus about half the size of character based ones. The code `t_end` marks the end of an object module, but end-of-file has the same effect. Those codes containing the characters `64` provide equivalent facilities for 64-bit versions of BCPL. Neither `t_reloc` nor `t_reloc64` are currently available in Cintsys or Cintpos.

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

Under cintpos, this call disables interrupts.

`sys(Sys_memmovebytes,` *dest*`,` *src*`.  n)` CIN:y, POS:y, NAT:y

This copies n bytes from BCPL byte address *src* to BCPL byte address *dest*. The source and destination regions may overlap. This function behaves as if the source region is first copied to a non overlapping place before copying it to the destination.

`sys(Sys_memmoveword,` *dest*`,` *src*`.  n)` CIN:y, POS:y, NAT:y

This copies n words from BCPL word address *src* to BCPL word address *dest*. The source and destination regions may overlap. This function behaves as if the source region is first copied to a non overlapping place before copying it to the destination.

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 a more efficient `muldiv` function is now defined in `syslib` invoking the Cintcode instruction `MDIV`, see section 3.3.

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

This function opens an output stream specified by the file name *name* in append mode causing all output to be appended onto the end of the file. If the file does not exist a zero length file of the given name is created. If successful it returns the file pointer to the given file, otherwise it returns zero.

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 94 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 Section 3.3.2 for information about the treatment of file names and Section 3.4 for information about random access files.

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 94 for information about the treatment of file names.

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

This returns a machine dependent value indicating under which architecture Cintsys or Cintpos is running.

res := sys(Sys_pollsardch) CIN:y, POS:y, NAT:y

This returns the next character from standard input if it is immediately available, otherwise it returns pollingch (=-3). If the input stream is exhausted it returns endstreamch (=-1). The character is not echoed to the standard output stream.

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

This function sets atomically the contents of the machine memory location whose word address is *addr* to val returning its previous setting. The address may point to system work space outside the normal Cintcode memory.

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 *fp* must have been created by a call of sys(Sys_openread,...) or sys(Sys_openreadwrite,...). The number of bytes actually read is returned as the result.

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). Unlessrunning in quietmode 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 13.2 on page 271.

`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

This will set the file position pointer of the opened file whose descriptor is *fd* to *pos*. The file descriptor is normally in the `scb_fd` field of the stream control block for that file. The value of *pos* must be between zero and the current number of bytes in the file. See Section 3.4 for more information about random access files.

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

Both interpreter `cinterp` and `fasterp` returns when a fault such as division by zero occurs or when a call of `sys(Sys_quit,...)` is made. Before returning, the interpreter save the Cintcode registers in `regs`. 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 existent Cintcode instruction
2	BRK instruction encountered
3	Count has reached zero
4	PC set to a negative value
5	Division by zero
10	Single step interrupt from the fast interpreter (debugging)
11	The value of the watched location in the Cincode memory has changed in the course of executing the previous instruction
12	Indirect address out of range
13	SIGINT received

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. The prefix string is held in space that was allocated when the system started. 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 147.

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

There is a variant of `cintsys` called `rastsys` that provides a way to generate data for time-memory images, and `cintpos` has a similar variant called `rastpos`. These systems can also generate bit streams that can be converted in sound related to the execution of programs. 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, 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. If $n = 4$ and *arg=1*, the system generated a bit stream file based on the fifth bit of the address of every access to the Cintcode memory. This file can later be converted to sound using the command `rast2wav`. The generated sound is somewhat similar to that generated by the Edsac 2 computer in Cambridge in the early 1960s.

When not representing bit stream data, the raster file contains text using run length encoding to represent raster lines. 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 150 for more information on how to use the rastering facility. See also the command `bits2ps`.

res `:= sys(Sys_settrcount,` ***count***`)` CIN:y, POS:y, NAT:n

This sets the private variable `trcount` used by the low level tracing mechanism to the specified value returning it previous setting. Setting it to a negative value disables the tracing mechanism. See `Sys_trpush` for more details.

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. Note that it may be necessary to run `alsamixer` to enable the sound device and adjust its volume setting. The available sound functions have mnemonic names declared in `g/sound.h` and are described below.

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

This returns `TRUE` is the `Sys_sound` functions are available on the current system.

res `:= sys(Sys_sound, snd_waveInOpen,` ***a1, a2 , a3 , a4*** `)` CIN:y, POS:y, NAT:y

This opens a sound wave device for input. *a1* is typically `"/dev/dsp"`, `"/dev/dsp1"` or a small integer, *a2* is the sample format, eg 16 for S16_LE, 8 for U8. *a3* is the number of channels, typically 1 or 2 and *a4* is the number of samples per second, typically 44100. The result is the file (or device) descriptor of the opened device or -1 if error.

res `:= sys(Sys_sound, snd_waveInPause,` ***a1***`)` CIN:y, POS:y, NAT:y

This will pause sound wave sampling from device *a1*. Recently read samples can still be read (to flush the buffered data).

res `:= sys(Sys_sound, snd_waveInRestart,` ***a1***`)` CIN:y, POS:y, NAT:y

Restart sound wave sampling.

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

Read samples from a sound wave input device *a1*, returning immediately. *a2* is the buffer in which to receive the samples and *a3* is the number of bytes to read. The result is the number of bytes actually transferred into the buffer.

res `:= sys(Sys_sound, snd_waveInClose,` ***a1***`)` CIN:y, POS:y, NAT:y

This closes sound wave input device *a1*.

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

This opens a sound wave device for output. *a1* is typically `"/dev/dsp"`, `"/dev/dsp1"` or a small integer, *a2* is the sample format, eg 16 for S16_LE, 8 for U8. *a3* is the number of channels, typically 1 or 2 and *a4* is the number of samples per second, typically 44100. The result is the file (or device) descriptor of the opened device or -1 if error.

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

Write samples from a sound wave output device *a1*. *a2* is the buffer holding the

samples and *a3* is the number of bytes to be written. The result is the number of bytes actually transferred from the buffer.

res `:= sys(Sys_sound, snd_waveOutClose,` *a1*`)` CIN:y, POS:y, NAT:y
This closes sound wave output device *a1*.

res `:= sys(Sys_sound, snd_midiInOpen,` *a1*`)` CIN:y, POS:y, NAT:y
This opens MIDI device for input specified by *a1* which is typically `"/dev/midi"`, `"/dev/dmmidi1"` or a small integer. The result is the file (or device) descriptor of the opened device or -1 if error.

res `:= sys(Sys_sound, snd_midiInRead,` *a1*`,` *a2* `,` *a3* `)` CIN:y, POS:y, NAT:y
This reads bytes from MIDI input device *a1* into buffer *a2*. *a3* is the number of MIDI bytes to read. The result is the actual number of bytes transferred or -1 if there was an error.

res `:= sys(Sys_sound, snd_midiInClose,` *a1*`)` CIN:y, POS:y, NAT:y
This close MIDI input device *a1*.

res `:= sys(Sys_sound, snd_midiOutOpen,` *a1*`)` CIN:y, POS:y, NAT:y
This opens a MIDI device for output. *a1* is typically `"/dev/midi"`, `"/dev/dmmidi1"` or a small integer. The result is the file (or device) descriptor of the opened device or -1 if error.

res `:= sys(Sys_sound, snd_midiOutWrite1,` *a1*`,` *a2*`)` CIN:y, POS:y, NAT:y
This writes a one byte MIDI message (*a2*) to MIDI device *a1*.

res `:= sys(Sys_sound, snd_midiOutWrite2,` *a1*`,` *a2* `,` *a3* `)` CIN:y, POS:y, NAT:y
This writes a two byte MIDI message (*a2 a3*) to MIDI device *a1*.

res `:= sys(Sys_sound, snd_midiOutWrite3,` *a1*`,` *a2* `,` *a3* `,` *a4* `)` CIN:y, POS:y, NAT:y
This writes a three byte MIDI message (*a2 a3 a3*) to MIDI device *a1*.

res `:= sys(Sys_sound, snd_midiOutWrite,` *a1*`,` *a2* `...)` CIN:y, POS:y, NAT:y
This write *a3* MIDI bytes from buffer *a2* to MIDI output device *a1*. The result is the number of bytes actually sent.

res `:= sys(Sys_sound, snd_midiOutClose,` *a1*`)` CIN:y, POS:y, NAT:y
This closes MIDI output device *a1*.

`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 while execution is 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

This causes the command *comstr* to be executed by the command language shell of the operating system under which Cintsys or Cintpos is running.

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 13.2) 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

This returns the current file position pointer of the opened file whose descriptor is *fd*. The file descriptor is normally in the scb_fd field of the stream control block for that file. See Section 3.4 for more information about random access files.

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.

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

There is a low level circular trace buffer that can hold 4096 values, and a private variable trcount that holds the number of values currently pushed into this buffer. If trcount<0, low level tracing is disabled, but otherwise trpush pushes val into the buffer at position trcount MOD 4096 and increments trcount. The call sys(Sys_settrcount, *count*) sets trcount to the specified value (possibly disabling tracing) and returns its previous setting. The call sys(Sys_gettrval, *count*) gets the value in the trace buffer at position trcount MOD 4096. Normally this function is only called when tracing is disabled. Under both Cintsys and Cintpos, trpush can also be called from the parts of the system implemented in C.

This tracing mechanism is available both to the BCPL user and parts of the system such as cintpos.c, cinterp.c and devices.c. Under Cintpos these low level tracing functions use a mutex to control access to trcount and the circular buffer. It is thus thread safe and so can be used to help debug subtle timing problems in the system software. For an example of the use of this tracing mechanism see the command com/testtr.b.

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

This call is typically only made from the body of the Cintpos Idle task. It suspends the interpreter until either some Cintpos device issues an interrupt request or the specified timeout occurs. It is typically implemented by waiting with a timeout on a host operating system condition variable. When a device thread wishes to interrupt the interpreter it send a signal via the appropriate condition variable. Unfortunately some operating systems may take hundreds of milliseconds to reschedule the interpreter thread. A possible but selfish solution is for the Idle task to execute a busy loop instead of calling `waitirq`.

`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,...)` or `sys(Sys_openreadwrite,...)`. The result is the number of bytes transferred, or zero if there was an error.

pkt `:= taskwait()` CIN:n, POS:y, NAT:n

If there is a packet in the task's queue it is dequeued and returned as the result. If there was no packet on the work queue this task is suspended in WAIT state and control given to a lower priority task.

res `:= testbit(`*bitno*`,` *bitvec*`)` CIN:y, POS:y, NAT:y

This function returns a non zero value if and only if the specified bit in *bitvec* is a one. The bits are numbered from zero starting at the least significant bit of `bitvec!0`. `bitvec!0` holds bits 0 to `bitsperword-1`, `bitvec!1` holds bits `bitsperword` to `2*bitsperword-1`, etc.

res `:= testflags(`*flags*`)` CIN:n, POS:y, NAT:n

This Cintpos function tests and clears specified flags in the task control block of the current task. Flags are bits in the `tcb_flags` field of the task control block, and they are normally called `A`, `B`, etc corresponding to consecutive bits from the least significant end of the field. A flag is set if the corresponding bit is a one. The argument *flags* is a bit pattern identifying which flags are being inspected. The result is `FALSE` if none of the specified flags were set, and `TRUE` if at least one was, in which case `result2` is set to a bit pattern representing the flags that were set and have now been cleared.

`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 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.

Item	Substitution
%s	Write the next argument as a string using writes.
%*n*t %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 a in UTF-8 or GB2312 character using codewrch.
%*n*b %b*n*	Write the next argument as a binary number in a field width of *n* characters using writebin.
%*n*o %o*n*	Write the next argument as an octal number in a field width of *n* characters using writeoct.
%*n*x %x*n*	Write the next argument as a hexadecimal number in a field width of n characters using writehex.
%*n*i %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.
%*n*u %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 %.
%pc	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.
%*n*.*m*f	Write the next argument as a floating point number in a field with of *n* with *m* digits after the decimal point. The output is generated using writeflt.
%*n*.*m*e	Write the next argument as a floating point number in exponential form in a field with of *n* with *m* digits after the decimal point. The output is generated using writee.
%m	The next argument 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

```
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 `sysb/blib.b`) is a good example of how a variadic function can be defined.

`writeflt(`*x*`,` *w*`,` *p*`)` CIN:y, POS:y, NAT:y

This routine outputs the floating point number x to the currently selected output stream in a field of width w with p digits after the decimal point.

`writee(`*x*`,` *w*`,` *p*`)` CIN:y, POS:y, NAT:y

This routine outputs the floating point number x to the currently selected output stream in exponential form in a field of width w with p digits after the decimal point.

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 element `buf` is either zero or holds the stream's byte buffer which must have been allocated using `getvec` and must be freed using `freevec` when the stream is closed. 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, output or both and `work` is private work space for the action function `rdfn`, `wrfn` which are called, repectively, when the byte buffer becomes empty on reading or full on output. The function `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`

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 is given in figure 3.8.

```
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`

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 ch<'*s' & cos!scb_type<0 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 operating system and so some details such as the maximum length of file names are machine dependent. Previously, BCPL used to follow the syntax of target machine files names, but recently BCPL attempts to be more machine independent by mainly adopting the Linux style of names and converting them to target machine form at runtime. The target machine format is set by a configuration parameter set when the system was installed. The formats currently available are for Unix, Windows and VMS.

Within BCPL file names slashs (`/`) and back slashes (\) are regarded as separators between the components of file names. File names may start with a colon prefix consisting of letters and digits followed by a colon, as in `TCP:shep.cl.cam.ac.uk:9000` or `G:test.b`. Such prefixes allow access to special features such as URLs used in

TCP/IP communication or to other filing systems. These are often dependent on the host operating system.

A file name starting '/' or '\' or containing a colon is treated as an absolute name; all others are relative names and are interpreted relative to the current directory. A file name consisting of a single asterisk (*) is special and represents standard input (normally the keyboard) or standard output (normally the screen) depending on context. Within a file name, the components dot (.) and double dot (..) represent the current and parent directories, respectively. As an example, the file name `../bcplprogs/demos/queens.b` is valid and automatically converted when used to `..\bcplprogs\demos\queens.b` under Windows or to `[-.bcplprogs.demos]queens.b` under VMS.

Some operating systems such as Windows CE2.0 have no concept of a current working directory. For such systems there is a feature that allows users to specify a character string to be automatically prepended to any relative (non absolute) file name before it is used. The prefix string is stored in static Cintcode space allocated when Cintsys or Cintpos starts up. It can be inspected and changed using the calls: `sys(Sys_getprefix)` and `sys(Sys_setprefix, prefix)`, or the CLI command `prefix` described on page 147. The prefix string is only used with relative file names not already prefixed with directories given by path variables such as `BCPLPATH` or `POSPATH`.

3.4 Random Access

Disk files can be regarded as potentially huge vectors of bytes with the first byte being at position zero of the file. An opened stream to or from a file has a file position pointer that holds the position relative to the start of where the next byte will be transferred. For any such stream this position can be read using `note(scb, posv)` or updated using `point(scb, posv)`. For read-write streams it is possible to read or write data at any position in the file.

Disk files can also be regarded as potentially huge collections of fixed length records. The user must specify the record size by calling `setrecordlength`. The records of a file are given consecutive numbers starting with zero, and can be read or written using `get_record` and `put_record`. The record number of the next record to be transferred can be obtained by calling `recordnote` and can be set using `recordpoint`.

3.5 RAM streams

A special form of random access stream is a RAM stream which can be created by the call `findinoutput("RAM:")`. RAM streams hold all the data in main memory in the stream buffer. As data is written to a RAM stream, its buffer is automatically enlarged as needed. The data can be read back by calling `rewindstream` followed by calls of `rdch`. Alternatively it can be accessed from the buffer held in `scb!scb_buf`. The number of valid bytes in the buffer is `scb!scb_end`. When a RAM stream is closed its buffer and scb are returned to free store.

3.6 Environment Variables

Most operating systems allow the user to set environment variables whose names consist of letters and digits and whose values are arbitrary character strings. Both Cintsys and Cintpos use such variables to specify directories to be searched when looking up files in certain contexts. These directories are separated by semicolons or colons, but when running under Windows only semicolons are allowed.

In the standard Cintsys system the environment variable `BCPLROOT` holds the file name of the root directory of the system. `BCPLPATH` holds a list of directories that are searched when attempting to load the Cintcode compiled form of a BCPL program. `BCPLHDRS` holds the directories to be searched when the BCPL compiler is processing a `GET` directive and `BCPLSCRIPTS` specified the directories to be searched when the `c` command is looking for a command-command script.

In the standard Cintpos system these variables are called `POSROOT`, `POSPATH`, `POSHDRS` and `POSSCRIPTS`. It is sometimes convenient to use other names, for instance, `NBCPLROOT`, `NBCPLPATH`, `NBCPLHDRS` and `NBCPLSCRIPTS` might be used when developing a new version of Cintsys. To make this possible the system allocates static space to hold the names and provides the command `setroot` described on page 152 to allow the user to change them. These names may be up to 63 characters long are accessible to commands such as `bcpl`, `c` and `setroot` via the rootnode fields `rtn_rootvar`, `rtn_pathvar`, `rtn_hdrsvar` and `rtn_scriptsvar`.

When Cintsys (or Cintpos) starts up it requires a valid setting of `rtn_pathvar` in order to locate Cintcode modules such as `BOOT` and `BLIB`. The default setting of this field is `BCPLPATH` (or `POSPATH`) but can be changed using the `-cin` argument at startup as in

```
cintsys -cin NBCPLPATH
```

After loading the resident system control is passed to `BOOT` which updates the variable names appropriately for the system being run. It is unlikely that the user will want change them using `setroot` although it might be useful to use `setroot` to see what names are currently being used.

If the value of an environment variable represents a list of directories, they should be given using Linux style slash (`/`) separators and the directories separated by semicolons (rather than the Linux style colons). This allows colon prefixes such as `G:` to be used in, for instance, Windows version of the system. For compatibility with older systems, colons may be used as an alternative to semicolons when not running under Windows.

When Cintpos starts up the process is similar except the setting of `rtn_pathvar` is `POSPATH` unless explicitly changed using `-cin`.

When installing `cintsys` or `cintpos` for the first time it is common to fail to set the environment variables correctly. To help repair such mistakes, use the `-f` option when calling `cintsys` or `cintpos`. This will output a trace of every time any file is looked up using an environment variable. Even more information is generated if the `-v` argument is also given (or even `-vv`). Until the system is working correctly it is recommended that it is started using

```
cintsys -f
```

or

```
cintpos -f -v
```

3.7 Coroutine examples

This section contains examples that use the coroutine mechanism.

3.7.1 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 currfreq = freq            // These only change at the
    LET curramplitude = amplitude  // start of a complete cycle.
    LET q4 = rate*1000
    LET q2 = q4/2
    UNTIL x > q2 DO { cowait(+curramplitude) // First half cycle
                      x := x + currfreq
                    }
    UNTIL x > q4 DO { cowait(-curramplitude) // Second half cycle
                      x := x + currfreq
                    }
    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)
  ...
}
```

The frequency and amplitude can be changed by assignments such as:

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

Note that the new frequency and amplitude take effect at the start of the next complete cycle.

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

3.7.2 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 an 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.

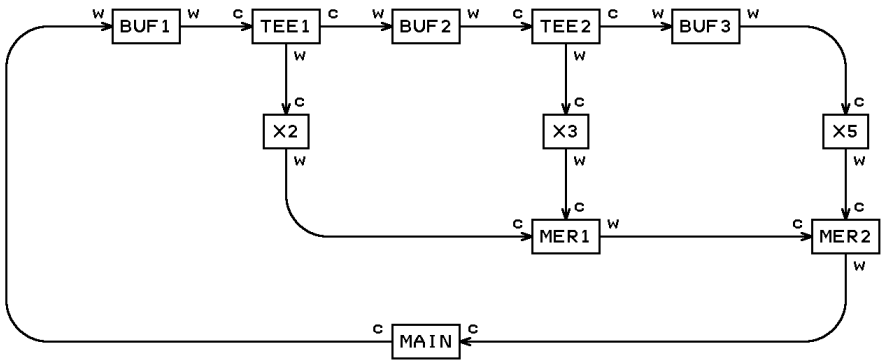


Figure 3.10: Coroutine data flow

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)` executed in coroutine `X2` will yield a value that `TEE1` extracted from `BUF1`, after sending a copy 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`, respectively. `MER1` merges two monotonically increasing streams of numbers produced by `X2` and `X3`. The resulting monotonic 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 MOD 201)
                      q := q+1
                    }
               ELSE { IF p=q+201 DO writef("Buffer full*n")
                      v!(p MOD 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 MOD 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
}
```

3.7.3 A Discrete Event Simulator

This is a benchmark test for a discrete event simulator using coroutines. It simulates a network of n nodes which each receive, queue, process and transmit messages to other nodes. The nodes are uniformly spaced on a straight line and the network delay is assumed to be proportional to the linear distance between the source and the destination. When a message arrives at a node it is queued if the node was busy, otherwise it is processed immediately. After processing the message for random time, it is sent to another randomly chosen node. After dispatching the message, the node dequeues its next message and processes it if there is one, otherwise the node becomes suspended. Initially every node is processing a message and every queue is empty. There are n coroutines to simulate the progress of each message and the discrete event priority queue is implemented using the heapsort heap structure. The simulation stops at a specified simulated time. The result is the number of messages that have been processed. A machine independent random number generator is used so the resulting value should be independent of implementation language and machine being used.

The program is given below. When it is run using the default settings, it executes 435,363,350 Cintcode instructions and has 2,510,520 coroutine changes.

```
SECTION "cosim"

GET "libhdr"

GLOBAL {
  priq:ug    // The vector holding the priority queue
  priqupb    // The upper bound
  priqn      // Number of items in the priority queue
  wkqv       // The vector of work queues
  count      // count of messages processed
  nodes      // The number of nodes
  ptmax      // The maximum processing time
  stopco     // The stop coroutine
  cov        // Vector of message coroutines
  ranv       // A vector used by the random number generator
  rani; ranj // subscripts of ranv
  simtime    // Simulated time
  stoptime   // Time to stop the simulation
  tracing

// Functions
  rnd;  initrnd;  closernd;  prq;  insertevent;  upheap
  downheap;  getevent;  waitfor;  prwaitq;  qitem;  dqitem
  stopcofn;  messcofn
}

// ################### Random number generator ########################

// The following random number generator is based on one given
// in Knuth: The art of programming, vol 2, p 26.
LET rnd(n) = VALOF
{ LET val = (ranv!rani + ranv!ranj) & #x_FFF_FFFF
  ranv!rani := val
  rani := (rani + 1) MOD 55
  ranj := (ranj + 1) MOD 55
  RESULTIS val MOD n
}

AND initrnd(seed) = VALOF
{ LET a, b = #x_234_5678+seed, #x_536_2781
  ranv := getvec(54)
  UNLESS ranv RESULTIS FALSE
  FOR i = 0 TO 54 DO
  { LET t = (a+b) & #x_FFF_FFFF
    a := b
    b := t
    ranv!i := t
  }
  rani, ranj := 55-55, 55-24  // ie: 0, 31
  RESULTIS TRUE
}

AND closernd() BE IF ranv DO freevec(ranv)
```

```
// ################### Priority Queue functions ######################

AND prq() BE
{ FOR i = 1 TO priqn DO writef(" %i4", priq!i!0)
  newline()
}

AND insertevent(event) BE
{ priqn := priqn+1        // Increment number of events
  upheap(event, priqn)
}

AND upheap(event, i) BE
{ LET eventtime = event!0
//writef("upheap: eventtime=%n i=%n*n", eventtime, i)

  { LET p = i/2           // Parent of i
    UNLESS p & eventtime < priq!p!0 DO
    { priq!i := event
      RETURN
    }
    priq!i := priq!p      // Demote the parent
    i := p
  } REPEAT
}

AND downheap(event, i) BE
{ LET j, min = 2*i, ? // j is left child, if present
  IF j > priqn DO
  { upheap(event, i)
    RETURN
  }
  min := priq!j!0
  // Look at other child, if it exists
  IF j<priqn & min>priq!(j+1)!0 DO j := j+1
  // promote earlier child
  priq!i := priq!j
  i := j
} REPEAT

AND getevent() = VALOF
{ LET event = priq!1      // Get the earliest event
  LET last  = priq!priqn  // Get the event at the end of the heap
  UNLESS priqn>0 RESULTIS 0 // No events in the priority queue
  priqn := priqn-1        // Decrement the heap size
  downheap(last, 1)       // Re-insert last event
  RESULTIS event
}

AND waitfor(ticks) BE
{ // Make an event item into the priority queue
  LET eventtime, co = simtime+ticks, currco
  insertevent(@eventtime) // Insert into the priority queue
  cowait()                // Wait for the specified number of ticks
}
```

```
// ###################### Queueing functions ########################

AND prwaitq(node) BE
{ LET p = wkqv!node
  IF -1 <= p <= 0 DO { writef("wkq for node %n: %n*n", node, p); RETURN }
  writef("wkq for node %n:", node)
  WHILE p DO
  { writef(" %n", p!1)
    p := !p
  }
  newline()
}

AND qitem(node) BE
// The message has reached this node
// It currently not busy, mark it as busy and return to process
// the message, other append it to the end of the work queue
// for this node.
{ // Make a queue item
  LET link, co = 0, currco
  LET p = wkqv!node
  UNLESS p DO
  { // The node was not busy
    wkqv!node := -1  // Mark node as busy
    IF tracing DO
      writef("%i8: node %i4: node not busy*n", simtime, node)
    RETURN
  }
  // Append item to the end of this queue
  IF tracing DO
    writef("%i8: node %i4: busy so appending message to end of work queue*n",
           simtime, node)
  TEST p=-1
  THEN wkqv!node := @link     // Form a unit list
  ELSE { WHILE !p DO p := !p  // Find the end of the wkq
         !p := @link          // Append to end of wkq
       }
  cowait() // Wait to be activated (by dqitem)
}

AND dqitem(node) BE
// A message has just been processed by this node and is ready to process
// the next, if any.
{ LET item = wkqv!node // Current item (~=0)
  UNLESS item DO abort(999)
  TEST item=-1
  THEN wkqv!node := 0                  // The node is no longer busy
  ELSE { LET next = item!0
         AND co   = item!1
         wkqv!node := next -> next, -1 // De-queue the item
         callco(co)                    // Process the next message
       }
}
```

```
// ######################## Coroutine Bodies ##########################

AND stopcofn(arg) = VALOF
{ waitfor(stoptime)
  IF tracing DO
    writef("%i8: Stop time reached*n", simtime)
  RESULTIS 0
}

AND messcofn(node) = VALOF
{ qitem(node)   // Put the message on the work queue for this node

  { // Start processing the first message
    LET prtime   = rnd(ptmax)     // a random processing time
    LET dest     = rnd(nodes) + 1 // a random destination node
    LET netdelay = ABS(node-dest) // the network delay

    IF tracing DO
      writef("%i8: node %i4: processing message until %n*n",
             simtime, node, simtime+prtime)
    waitfor(prtime)
    count := count + 1 // One more message processed
    IF tracing DO
      writef("%i8: node %i4: message processed*n",
             simtime, node, dest, simtime+netdelay)
    dqitem(node) // De-queue current item and activate the next, if any
    IF tracing DO
      writef("%i8: node %i4: sending message to node %n to arrive at %n*n",
             simtime, node, dest, simtime+netdelay)

    waitfor(netdelay)
    node := dest      // The message has arrived at the destination node
    IF tracing DO
      writef("%i8: node %i4: message reached this node*n",
             simtime, node)
    qitem(node)   // Queue the message if necessary
    // The node can now process the first message on its work queue
  } REPEAT
}

// ######################## Main Program ##########################

LET start() = VALOF
{ LET seed = 0
  LET argv = VEC 50

  UNLESS rdargs("-n/n,-s/n,-p/n,-r/n,-t/s", argv, 50) DO
  { writef("Bad arguments for cosim*n")
    RESULTIS 0
  }

  nodes, stoptime, ptmax := 500, 1_000_000, 1000
  IF argv!0  DO nodes    := !(argv!0) // -n/n
  IF argv!1  DO stoptime := !(argv!1) // -s/n
  IF argv!2  DO ptmax    := !(argv!2) // -p/n
  IF argv!3  DO seed     := !(argv!3) // -r/n
  tracing := argv!4                   // -t/s
```

```
  writef("*nCosim entered*n*n")
  writef("Network nodes:       %n*n", nodes)
  writef("Stop time:           %n*n", stoptime)
  writef("Max processing time: %n*n", ptmax)
  writef("Random number seed:  %n*n", seed)
  newline()

  UNLESS initrnd(seed) DO
  { writef("Can't initialise the random number generator*n")
    RESULTIS 0
  }
  stopco := 0
  wkqv, priq, cov := getvec(nodes), getvec(nodes+1), getvec(nodes)
  UNLESS wkqv & priq & cov DO
  { writef("Can't allocate space for the node work queues*n")
    GOTO ret
  }

  FOR i = 1 TO nodes DO wkqv!i, cov!i := 0, 0
  priqn := 0  // Number of events in the priority queue
  count := 0  // Count of message processed
  simtime := 0 // Simulated time

  IF tracing DO writef("%i8: Starting simulation*n", simtime)

  // Create and start the stop coroutine
  stopco := createco(stopcofn, 200)
  IF stopco DO callco(stopco)
  // Create and start the message coroutines
  FOR i = 1 TO nodes DO
  { LET co = createco(messcofn, 200)
    IF co DO callco(co, i)
    cov!i := co
  }
  // Run the event loop

  { LET event = getevent()      // Get the earliest event
    UNLESS event BREAK
    simtime := event!0          // Set the simulated time
    IF simtime > stoptime BREAK
    callco(event!1)
  } REPEAT

  IF tracing DO writef("*nSimulation stopped*n*n")
  writef("Messages processed: %n*n", count)
ret:
  FOR i = nodes TO 1 BY -1 IF cov!i DO deleteco(cov!i)
  IF cov    DO freevec(cov)
  IF wkqv   DO freevec(wkqv)
  IF priq   DO freevec(priq)
  IF stopco DO deleteco(stopco)
  closernd()
  RESULTIS 0

fail:
  writef("Unable to initialise the simulator*n")
  GOTO ret
}
```

3.8 The BMP Graphics Library

The graphics library provides facilities for drawing pictures and outputing them to file. This library is designed to generate `.bmp` files representing potentially large images using 8-bit or 24-bit coloured pixels. It is designed to create files representing 2 dimensional rectangular images using the `.bmp` file format. It should not be confused with the SDL and GL libraries (described later) used to generate images on the display screen suitable for interactive graphics typically used in computer games.

This library is initialised by a call of `opengraphics` which specifies the size of the image to be drawn and whether 8 or 24 bit pixels are to be used. It also sets up a palette of 256 colours if 8 bit pixels are to be used. The graphics header file is in `g/graphics.h`. It declares the constants `mode8bit`, `mode8bitalt` and `mode24bit` for use in the call of `opengraphics`. It also declares several variables starting at position `g_grbase` which is declared with value 450 in `libhdr.h` but can be redefined, if necessary, before inserting `graphics.h`. The graphics global variables are as follows.

xsize, **ysize**
These hold the the number of pixels in each row and column of the canvas.

bmpmode
This is set by `opengraphics` to `mode8bit`, `mode8bitalt` or `mode24bit`.

bpp
This holds the the number of bytes (1 or 3) per pixel in `canvas`.

rowlen
This holds `bpp*xsize`, the number of bytes in `canvas` to represent a row.

canvassize
This holds the number of bytes (`rowlen*ysize`) in `canvas`.

canvasupb
This holds the UPB of `canvas` in words.

canvas
This holds the vector, allocated by `getvec(canvasupb)`, of pixel bytes to represent the image. Each pixel is either an 8-bit byte identifying a colour in the palette or 3 bytes giving the blue, green and red components of the colour directly.

palettev
If 8-bit pixels are being used, this holds the palette vector of 256 colours. The colours are specified by values of the form `#Xrrggbb` in the least significant 24 bits of each element of `palettev`. `palettev` is set to zero when 24 bit pixels are being used.

col_white, **col_majenta**, **col_blue**, **col_cyan**, **col_green**, **col_yellow**, **col_red**, **col_black**
These variables hold either various 8 or 24 bit colour values.

currx, **curry**, **currcolour**

These variables hold the current pixel location and the current 8 or 24 bit colour. This library uses the convention that the bottom leftmost pixel has coordinate `(0,0)`. The direction of the x axis is to the right and the direction of the y axis is up. The primary use of `currx` and `curry` is their use in `drawto`, `drawby` and `drawch` to make drawing sequences of lines and characters more convenient.

3.8.1 The Graphics Functions

The BMP global functions are defined in `g/graphics.b`. They are as follows. Except for `opengraphics`, `closegraphics` and `wrgraph` they are the same as those in the SDL library.

`opengraphics(`*`xsize, ysize, mode`*`)`

This function sets `bmpmode` to *mode* and allocates the vector `canvas`. If 8 bit pixels are specified by *mode* it allocates `palettev` and fills it with one of two sets of palette colours. It also initialised all the other graphics variables. The canvas is initially filled with white pixels (like a blank sheet of paper).

`closegraphics()`

This function closes the graphics library returning `canvas` to freestore and if `palettev` was allocated it is also returned.

`drawpoint(`*`x, y`*`)`

This function places a pixel with colour specified by `currcolour` at position (x, y) on the canvas.

`drawpoint33(`*`x, y`*`)`

This function places a 3x3 square of pixels with the colour specified by `currcolour` centred at position (x, y) on the canvas.

`drawch(`*`ch`*`)`

This function draws a 8x12 array of pixels representing the given character. Its colour is specified by `currcolour` on a white background. The bottom leftmost pixel of the character is at (`currx`,`curry`). If *ch* is `'*n'`, `currx` is set to 10 and `curry` is decremented by 14, otherwise `currx` is incremented by 9.

`drawstr(`*`x, y,str`*`)`

This function calls `drawch` for each character in the given string starting at position (x, y),

`moveto(`*`x, y`*`)`

This function sets `currx` and `curry` to x and y, respectively.

`moveby(`*`dx, dy`*`)`

This function increments `currx` and `curry` by dx and dy, respectively.

`drawto(`*`x, y`*`)`

This function draws a straight line of colour `currcolour` from (`currx`, `curry`) to (x, y). It leaves (`currx` and `curry`) to x and y.

`drawby(`*`dx, dy`*`)`
This function draws a straight line of colour `currcolour` from (`currx`, `curry`) to (`currx`+dx, `curry`+dy). It then increments `currx` and `curry` by dx and dy, respectively.

`drawrect(`*`x, y, w, h`*`)`
This function draws the outline of the rectangle of width w and height h with the bottom left corner at (x, y) using `currcolour`.

`drawrndrect(`*`x, y, w, h, radius`*`)`
This function draws the outline of the rectangle of width w and height h with its bottom left corner at (x, y) with rounded corners of given radius. Its colour is specified by `currcolour`. If *radius* is less than or equal to zero the corners are square, and if *radius* is greater than half the shorter side length it is reduced to this value. `currx` and `curry` are set to $x1$ and $y1$, respectively.

`fillrect(`*`x, y, w, h`*`)`
This function draws a rectangle of width w and height h with its bottom left corner at (x, y). It is filled with the colour specified by `currcolour`.

`fillrndrect(`*`x, y, w, h, radius`*`)`
This function draws the rectangle of width w and height h with rounded corners of given radius. The bottom left corner is at (x, y). It colour is specified by `currcolour`. If *radius* is less than or equal to zero the corners are square, and if *radius* is greater than half the shorter side length it is reduced to this value.

`drawcircle(`*`x, y, radius`*`)`
This function draws a circle centred at (x, y) with given radius. Its colour is specified by `currcolour`.

`fillcircle(`*`x, y, radius`*`)`
This function draws a filled circle centred at (x, y) with given radius. Its colour is specified by `currcolour`.

`drawellipse(`*`x, y, rx, ry`*`)`
This function draws an ellipse centred at (x, y) with given `x` and `y` radii. Its colour is specified by `currcolour`.

`fillellipse(`*`x, y, rx, ry`*`)`
This function draws a filled ellipse centred at (x, y) with given `x` and `y` radii. Its colour is specified by `currcolour`.

`wrgraph(`*`filename`*`)`
This function writes the image held in `canvas` to the given file in `.bmp` format. The image is (currently) scaled to 300 DPI which corresponds to 11811 pixels per metre. At this scale the size of an A4 page is 2490x3510 pixels.

There are two programs to illustrate how this graphics library can be used. They are `bcplprogs/tests/grtst.b` and `bcplprogs/tests/grpalette.b`. If you are using BCPL under Linux you can compile and run them as follows.

```
cd ~/distribution/BCPL/bcplprogs/test
cintsys
c b grtst
grtst
ctrl-c
gimp grtst.bmp
ctrl-c
cintsys
c b grpalette
grpalette b8
ctrl-c
gimp palette.bmp
ctrl-c
cintsys
grpalette b24
ctrl-c
gimp palette.bmp
```

The image displayed by the last call of `gimp` is shown in figure 3.11 illustrating some of the colours available when using 24 bit pixels.

3.9 The SDL Graphics Library

The SDL Graphics Library implemented in C is available for many platforms including Linux, Windows and OSX and BCPL has an interface with this library allowing the user to create a window on the screen and repeatedly draw simple images allowing simple interactive games to be implemented. It also provides access to the keyboard, the mouse and joysticks. In due course this interface will allow the generation of sound.

To include these features it is necessary to install the SDL libraries on you machine and then build `cintsys` using a Makefile such as `MakefileSDL` or `MakefileRaspiSDL`.

The SDL operations are invoked by calls of the form `sys(Sys_sdl,...)`. There is a header file (`g/sdl.h`) declaring the various constants and globals available, and `g/sdl.b` contains the definitions of several functions providing the interface. The constant `g_sdlbase` is set in `libhdr.h` to be the first global used in the SDL library. It can be overridden by re-defining `g_sdlbase` before GETting `sdl.h`.

A program using the SDL library should start with the following lines.

```
GET "libhdr"
MANIFEST { g_sdlbase=nnn  } // Only used if the default setting of 450 in
```

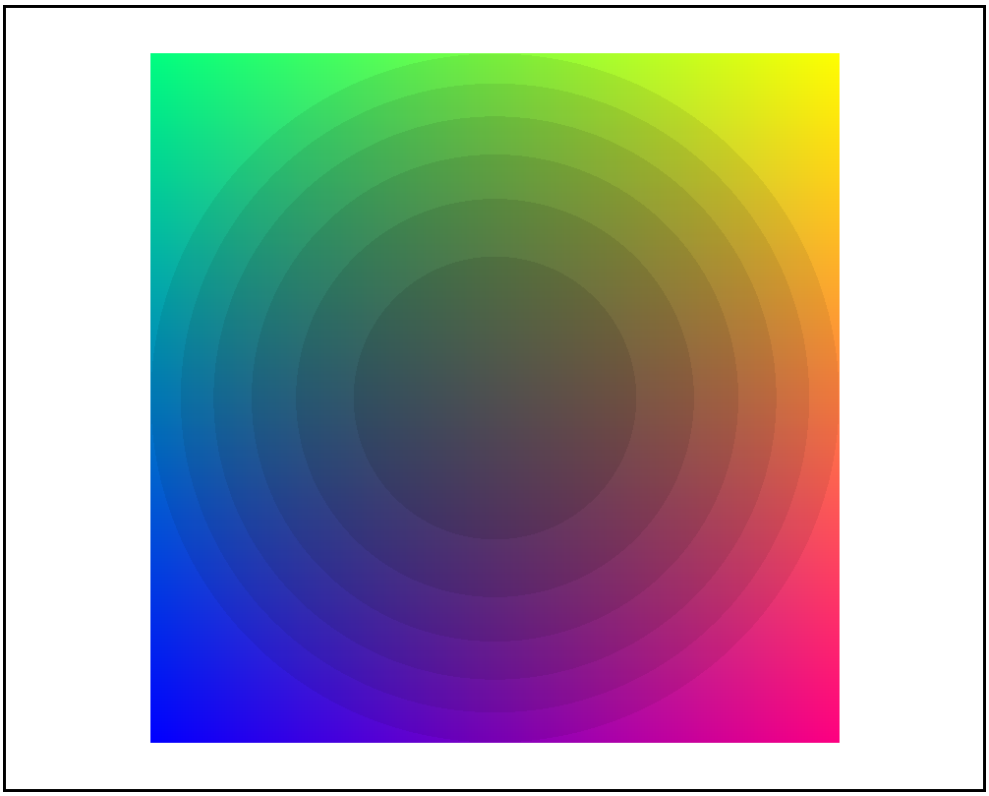


Figure 3.11: The image created by `grpalette b24`

```
                              // libhdr is not suitable.
GET "sdl.h"
GET "sdl.b"                   // Insert the library source code
.
GET "libhdr"
MANIFEST { g_sdlbase=nnn  } // Only used if the default setting of 450 in
                              // libhdr is not suitable.
GET "sdl.h"
```

There are several programs that use SDL described in Chapter 4 of `bcpl4raspi.pdf` available from my home page.

3.9.1 `sdl.h` details

The BCPL SDL functions make use of functions provided by the SDL library implemented in C. These sometimes use machine addresses pointing to structures such as those representing windows and surfaces. Such addresses are stored in BCPL in a pair of words as described in the description of `scb_fd` on page 51.

The global variables `screen`, `screen1`, `currsurf`, `currsurf1`, `format`, `format1`, `joystick` and `joystick1` hold address pairs for The SDL window, the currently selected surface, the format structure for that window and a joystick.

A call of `mkscreen` sets the variables `screenxsize` and `screenysize` to the number of pixels in the width and height of the created window. It alse sets `fscreenxsize` and `fscreenysize` to the floating point versions of these variables. The variable `fscreencentrex` and `fscreencentrey` are the floating point coordinates of the center of the screen.

The width and height of the currently seclected surface is held in `currxsize` and `currysize`.

The vectors `leftxv`, `leftzv`, `rightxv` and `rightzv` are used by the functions such as `drawtriangle` and `drawtriangle3d` defined in `g/sdl.b` that draw filled objects. The variables `miny` and `maxy` are also used by these functions.

The vector `depthv` and variables `miny` and `maxy` are used by functions in `g/sdl.b` and should not be touched by the user. The 3D drawing functions use `depthv` to implement the hiding of pixels that are further from the eye than other pixel at the same position on the screen. The upperbound of `depthv` is `depthvupb` (=`screenxsize*screenysize-1`). The elements of `depthv` are scaled integers with `zfac` units of depth corresponding to a distance of one pixel. The variab;e `zfac` is actually a floating point number since the 3D drawing functions typiclally take floating point pixel coordinates. If `zfac` is too small the the line of intersection of two nearly parallel plane can become inaccurate. The maximum allowable scaled depth is held in `maxdepth` (`=-1_000_000_000`).

Whenever anything is drawn it is given the colour held in the variable `currclour`. The variables `currx` and `curry` give the starting position of 2D lines and characters. They are updated after each line or characte is drawn. This allows long sequences of 2D lines or charaters to be drawn conveniently.

There is a similar mechanism for drawing 3D lines using the variables `currx3d`, `curry3d` and `currsz3d`. These hold the integer `x` and `y` pixel coordindates of the next 3D line to be drawn with `currsx3d` being its scaled integer depth.

The variables `mousex` and `mousey` hold the current pointer position 0n the screen, and `mousebuttons` is a bitpattern indicating which mouse buttons are currently pressed.

Some or all of the variables `eventtype`, `eventa1`, `eventa2`, `eventa3`, `eventa4` and `eventa5` are set by the function `pollevents` as described below.

```
sdle\_active
sdle\_keydown
sdle\_keyup
sdle\_mousemotion
sdle\_mousebuttondown
sdle\_mousebuttonup
sdle\_joyaxismotion
sdle\_joyballmotion
sdle\_joyhatmotion
sdle\_joybuttondown
```

```
sdle\_joybuttonup
sdle\_quit
sdle\_syswmeven
sdle\_videoresize
sdle\_userevent

sdle\_arrowup
sdle\_arrowdown
sdle\_arrowright
sdle\_arrowleft

sdl\_init_everything

sdl\_SWSURFACE   // Surface is in system memory
sdl\_HWSURFACE   // Surface is in video memory

sdl\_ANYFORMAT  // Allow any video depth/pixel-format
sdl\_HWPALETTE  // Surface has exclusive palette
sdl\_DOUBLEBUF  // Set up double-buffered video mode
sdl\_FULLSCREEN  // Surface is a full screen display
sdl\_OPENGL      // Create an OpenGL rendering context
sdl\_OPENGLBLIT  // Create an OpenGL context for blitting
sdl\_RESIZABLE  // This video mode may be resized
sdl\_NOFRAME  // No window caption or edge frame
```

3.9.2 Functions defined in `sdl.b`

This section describes all the functions defined in `g/sdl.b` in alphabetical order.

`alloc2dvecs()`
this function is only used in `drawtriangle`. It allocates and initialises the vectors `leftxv` and `rightxv`.

`alloc3dvecs()`
this function is only used in `drawtriangle2d`. It allocates and initialises the vectors `leftxv`, `lefttszv`, `rightxv` and `rightszv`. .

`blitsurf(srcptr, dstptr, x, y)`
This copies the source surface into the specified position of the destination surface.

`bltsurfrect(srcptr, srcrect, dstptr, x, y)`
Copy the specified rectangle from the source surface to the specified position in the destination surface.

rc `:= closesdl()`
This closes down the SDL library returning all allocated space to freestore.

`crossprod(v1, v2, v3)`
This computes the cross product of `v1` and `v2` which are both vectors with three floating point elements. The result is thus a vector orthogonal to `v1` and `v2` whose length is the sine of the angle between the vectors multiplied by the product of their lengths. The vectors `v1`, `v1` and `v1` are in right handed orientation.

`drawby(dx, dy)`
This is just calls `drawto(currx+dx, curry+dy)`.

`drawby3d(FLT dx, FLT dy, FLT dz)`
This is just calls `drawto3d(currx+dx, curry+dy, currz+dz)`.

`drawch(ch)`
Draw a 12x8 character at the position specified b y and `curry` and increment `curry` by 9. If `ch` was `'*n'` set `currx` to 10 and decrement `curry` by 11.

`drawcircle(x0,y0, radius)`
Not yet described

`drawf(x, y, form, a, b, c,..., t)`
Not yet described

`drawfillcircle(x, y, radius)`
Not yet described

`drawfillrect(x0,y0, x1,y1)`
Not yet described

`drawfillrndrect(x0, y0, x1, y1, radius)`
Not yet described

`drawpoint(x, y)`
This draws a point at location `(x,y)` on the currently selected surface. It colour is te one set by the most recent call of `setcolour`.

`drawpoint3d(FLT x, FLT y, FLT z)`
This draws a point at location `(x,y)` on the currently selected surface. If the `z` value at that position is greater than `z` the pixel is not updated.

`drawpoint3di(x, y, sz)`
Not yet described

`drawquad(x1,y1, x2,y2, x3,y3, x4,y4)`
Not yet described

`drawquad3d(FLT x1, FLT y1, FLT z1, FLT x2, FLT y2, FLT z2, FLT x3, FLT y3, FLT z3, FLT x4, FLT y4, FLT z4)`
Not yet described

`drawrect()`
Not yet described

`drawrndrect()`
Not yet described

`drawstr()`
Not yet described

`drawto()`
Not yet described

`drawto3d()`
Not yet described

`drawto3d()`
Not yet described

`drawto3di()`
Not yet described

`drawtriangle()`
Not yet described

`drawtriangle3d(FLT x1, FLT y1, FLT z1, FLT x2, FLT y2, FLT z2, FLT x3, FLT y3, FLT z3)`
Not yet described

`drawwrch(ch)`
Not yet described

`fillsurf(col)`
Not yet described

`freesurface(surfptr)`
Not yet described

`getevent()`
Not yet described

`getmousestate()`
Not yet described

`hidecursor()`
Not yet described

rc `:= initsdl()`
This initialises the SDL library and sets the global variables used by `sdl.b`. It

must be called before any other SDL operations can be performed. It returns `TRUE` if successful.

res `:= inprod(v1, v2)`
This returns the inner product of `v1` and `v2` which are both vectors with three floating point elements. The result is thus the cosine of the angle between the vectors multiplied by the product of their lengths.

colour `:= maprgb(r,g,b)`
This return a value representing the colour specified by its `r`, `r` and `r` components. It uses the colour represention chosen during the call of `mkscreen`.

rc `:= mkscreen(title, xsize, ysize)`
This creates a window of specified size with the given title. It returns `TRUE` if sucessful.

`mksurface(w, h, surfptr)`
Not yet described

`moveby(dx, dy)`
This is just calls `moveto(currx+dx, curry+dy)`.

`moveby3d(FLT dx, FLT dy, FLT dz)`
This is just calls `moveto3d(currx+dx, curry+dy, currz+dz)`.

`moveto(x, y)`
This selects position `(x,y)` in the currently selected surface used by subsequent calls of `drawch`, `drawto` and `drawby`. Its depth coordinate is given the value zero.

`moveto3d(FLT x, FLT y, FLT z)`
This selects position `(x,y,z)` in the currently selected surface used by subsequent calls of `drawch`, `drawto` and `drawby`. By convention smaller values of z are deeper into the screen.

`sdldelay(msecs)` This causes a real time delay of the specified number of milliseconds.

`sdlmsecs()`
This return the number of real time milliseconds since the current command was entered.

`selectsurface(surfptr, xsize, ysize)`
This selects a surface for use in subsequent drawing commands. The arguments `xsize` and `ysize` specify the size of the surface in pixels. The pair of BCPL word used to represent the machine address of the surface is pointed to by `surfptr`. See `scb_fd` on page **??** for more details.

rc `:= setcaption(title)`

This resets the title of the window created by `mkscreen`. It returns `TRUE` if successful.

`setcolour(col)`

This sets the colour to be used in subsequent drawing commands. The colour should be one returned by a call of `maprgb`.

`setcolourkey(col)`

This sets the colour that has the special property that when an attempt is made to to draw a pixel with this colour the pixel is left with its previous colour. This mechanism is used, for example, when displaying the moving coloured circles by the program `bcplprogs/raspi/bucket.b`.

`setlims(x0,y0, x1,y2)` This function is used by `drawtriangle` which draws a filled 2D triangle. It updates entries in `leftxv` and `rightxv` for each value of `y` between `y0` and `y1`.

`setlims3d(x0,y0,sz0, x1,y2,sz1)` This function is used by `drawtriangle3d` when drawing a filled 3D triangle. It updates entries in `leftxv`, `leftszv`, `rightxv` and `rightszv` for each value of `y` between `y0` and `y1`. The arguments are all integers with the `z` components being scaled to cause `zfac` units to correspond to a distance of one pixel.

`showcursor()`

This causes the cursor to be displayed.

`standardize(v)`

This divides the three floating point elements of the vector `v` by length of the vector leaving the elements of `v` set to the direction cosines of the given vector.

`updatescreen()`

This causes the surface that is currently being drawn to be copied the display screen.

`write_ch_slice(x, y, ch, line)`

This function is used by `drawch` to plot a row pixels of a 8x12 character.

The pixels to be drawn on the screen by the next call of `updatescreen` are held in a vector pointed to by the global variable `screen`. In order to implement hidden surface removal there is a second vector depthv holding the `z` coordinates of pixels in `screen`. This vector is only used by the 3D drawing functions. The depth values were held as scaled integers with `zfac` units corresponding to a distance of one pixel.

This eliminated some of the problems caused by using integers to represent depth which was particularly noticable when two nearly parallel 3D plane interected. Unfortunately floating point numbers caused other problems mainly due to their lack of precision so integers are again going to be used but 64 units of an element of `depthv` will now represent a distance of one pixel. Even though these units are used in `depthv`,

all the 3D drawing functions will use the convention that one unit in `x`, `x` and `x` will represent a distance of one pixel. The fractional values in `depthv` only occur when drawing 3D lines and triangles. The vertices of lines and triangles always occur on pixel boundaries.

3.9.3 `sys(Sys_sdl,...)` calls

rc := `sys(Sys_sdl, sdl_avail`
This return `TRUE` if the SDL facilities are available.

rc := `sys(Sys_sdl, sdl_init)`
This ...

rc := `sys(Sys_sdl, sdl_setvideomode, width, height, bbp, flags)`

rc := `sys(Sys_sdl, sdl_quit)`

rc := `sys(Sys_sdl, sdl_locksurface, surfptr)`

rc := `sys(Sys_sdl, sdl_unlocksurface, surfptr)`

rc := `sys(Sys_sdl, sdl_getsurfaceinfo, surfptr, ptr)`

rc := `sys(Sys_sdl, sdl_getfmtinfo.  fmtptr)`

rc := `sys(Sys_sdl, sdl_geterror, str)`

rc := `sys(Sys_sdl, sdl_updaterect, surfptr, left, top, right, bottom)`

rc := `sys(Sys_sdl, sdl_loadbmp, filename of a .bmp image)`

rc := `sys(Sys_sdl, sdl_blitsurface.  src, srcrect, dest, destrect)`

rc := `sys(Sys_sdl, sdl_setcolourkey, surfptr, flags, colorkey)`

```
rc := sys(Sys_sdl, sdl_freesurface, surfptr)

rc := sys(Sys_sdl, sdl_setalpha, surfptr, flags, alpha)

rc := sys(Sys_sdl, sdl_imgload, filename)

rc := sys(Sys_sdl, sdl_delay, msecs)

rc := sys(Sys_sdl, sdl_flip, surfptr)

rc := sys(Sys_sdl, sdl_displayformat, surfptr)

rc := sys(Sys_sdl, sdl_waitevent, pointer)

rc := sys(Sys_sdl, sdl_pollevent, pointer)

rc := sys(Sys_sdl, sdl_getmousestate, pointer)

rc := sys(Sys_sdl, sdl_loadwav, file, spec, buff, len)

rc := sys(Sys_sdl, sdl_freewav, buffer)

rc := sys(Sys_sdl, sdl_wm_setcaption, string)

rc := sys(Sys_sdl, sdl_videoinfo, v)

rc := sys(Sys_sdl, sdl_maprgb.  formatptr, r, g, b)

rc := sys(Sys_sdl, sdl_drawline,)

rc := sys(Sys_sdl, sdl_drawhline, )
```

```
rc := sys(Sys_sdl, sdl_drawvline,)

rc := sys(Sys_sdl, sdl_drawcircle, )

rc := sys(Sys_sdl, sdl_drawrect, )

rc := sys(Sys_sdl, sdl_drawpixel, )

rc := sys(Sys_sdl, sdl_drawellipse, )

rc := sys(Sys_sdl, sdl_drawfillellipse, )

rc := sys(Sys_sdl, sdl_drawround.  )

rc := sys(Sys_sdl, sdl_drawfillround, )

rc := sys(Sys_sdl, sdl_drawfillcircle, )

rc := sys(Sys_sdl, sdl_drawfillrect, )

rc := sys(Sys_sdl, sdl_fillrect, )

rc := sys(Sys_sdl, sdl_fillsurf, )

rc := sys(Sys_sdl, sdl_numjoysticks)

rc := sys(Sys_sdl, sdl_joystickopen, index, jpyptr)

rc := sys(Sys_sdl, sdl_joystickclose, index)

rc := sys(Sys_sdl, sdl_joystickname, index)
```

```
rc := sys(Sys_sdl, sdl_joysticknumaxes, joyptr)

rc := sys(Sys_sdl, sdl_joysticknumbuttons, joyptr)

rc := sys(Sys_sdl, sdl_joysticknumballs, joyptr)

rc := sys(Sys_sdl, sdl_joysticknumhats, joyptr)

rc := sys(Sys_sdl, sdl_joystickeventstate, aeg_

rc := sys(Sys_sdl, sdl_getticks)

rc := sys(Sys_sdl, sdl_showcursor)

rc := sys(Sys_sdl, sdl_hidecursor)

rc := sys(Sys_sdl, sdl_mksurface)

rc := sys(Sys_sdl, sdl_setcolourkey)

rc := sys(Sys_sdl, sdl_joystickgetbutton)

rc := sys(Sys_sdl, sdl_joystickgetaxis)

rc := sys(Sys_sdl, sdl_joystickgetball)

rc := sys(Sys_sdl, sdl_joystickgethat)
```

3.10 The GL Graphics Library

This library is still under development

OpenGL is a sophisticated graphics library allowing 3D images to be drawn on the screen efficiently using the full power of the graphics hardware available on most machines. On most desktop and laptop machines the full OpenGL library is available, but on handheld devices only a simplified version called OpenGL ES is available. The BCPL interface is designed to work with whichever version of OpenGL is available. This library essentially provides a subset of the OpenGL ES features. Note that the GL interface on the Raspberry Pi uses OpenGL ES.

To include these features in `cintsys` it is necessary to install the OpenGL libraries on you machine and then build `cintsys` using a Makefile such as `MakefileGL`, `MakefileRaspiGL` or `MakefileVCGL`.

The GL library uses the `sys(Sys_gl,...)` functions. There is a header file (`g/gl.h`) declaring the various constants and globals available in the GL library, and `g/gl.b` contains the definitions of several functions providing the interface to OpenGL. The constant `g_glbase` is set in `libhdr` to be the first global used in the GL library. It can be overridden by re-defining `g_glbase` after GETting `libhdr`.

A program wishing to use the OpenGL library should start with the following lines.

```
GET "libhdr"
MANIFEST { g_glbase=nnn  }  // Only used if the default setting of 450 in
                            // libhdr is not suitable.
GET "gl.h"
GET "gl.b"                  // Insert the library source code
.
GET "libhdr"
MANIFEST { g_glbase=nnn  }  // Only used if the default setting of 450 in
                            // libhdr is not suitable.
GET "gl.h"
```

This library will be described in Chapter 5 of `bcpl4raspi.pdf` available from my home page.

3.11 The Sound Library

This library is under development

The sound library uses the `sys(Sys_sound,...)` functions to provide facilities for reading, writing and analysing sound data. There is a sound header file (`g/sound.h`) declaring various constants and globals available in the sound library. The sound library itself is in `g/sound.b` and can be inserted into a program by the following statements.

```
GET "libhdr"
MANIFEST { g_sndbase=nnn  } // Only used if the default setting of 400 in
```

```
                            // libhdr is not suitable.
GET "sound.h"
GET "sound.b"               // Insert the library source code
```

The manifest constant `g_sndbase` specifies the position of the first global variable to be used by the sound library.

3.11.1 The Sound Constants

The sound library is not yet available.

3.11.2 The Sound Global Variables

The sound library is not yet available.

3.11.3 The Sound Functions

The sound library is not yet available.

3.12 The EXT Library

This library is designed to allow users to construct their own extension library involving code in C and assembly language. Its structure is similar to that of the SDL and GL libraries.

It uses the `sys(Sys_ext,...)` functions to interface with C code defined in `sysc/extfn.c`, and has two header files `ext.h` and `ext.b` providing the BCPL interface. Programs using the EXT library should start with the following statements.

```
GET "libhdr"
MANIFEST { g_extbase=nnn  } // Only used if the default setting of 900 in
                            // libhdr is not suitable.
GET "ext.h"
GET "ext.b"                 // Insert the library source code
```

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 one of the directories specified by the shell environment variable (typically `BCPLPATH` or `POSPATH`) whose name is in `rootnode!rtn_path`. These commands are described in Section 4.3 and their source code can be found in the `com` directory. The command language is a combination of the features provided by the CLI and the collection of commands that can be invoked. Under Cintpos, a similar CLI program provides command language interpreters in several contexts such as those created by the commands: `run`, `newcli`, `tcpcli` and `mbxcli`. Details of the implementation of both CLIs are given at the end of this chapter from page 157.

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 141 below.

4.1 Bootstrapping Cintsys

When Cintsys 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 function `start` defined in `sysb/boot.b`. Since

no return link has been stored into the stack, this call of `start` must not attempt to return in the normal way; however, its execution can still be terminated using `sys(Sys_quit,0)`.

The global vector and stack shown in figure 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 the `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.

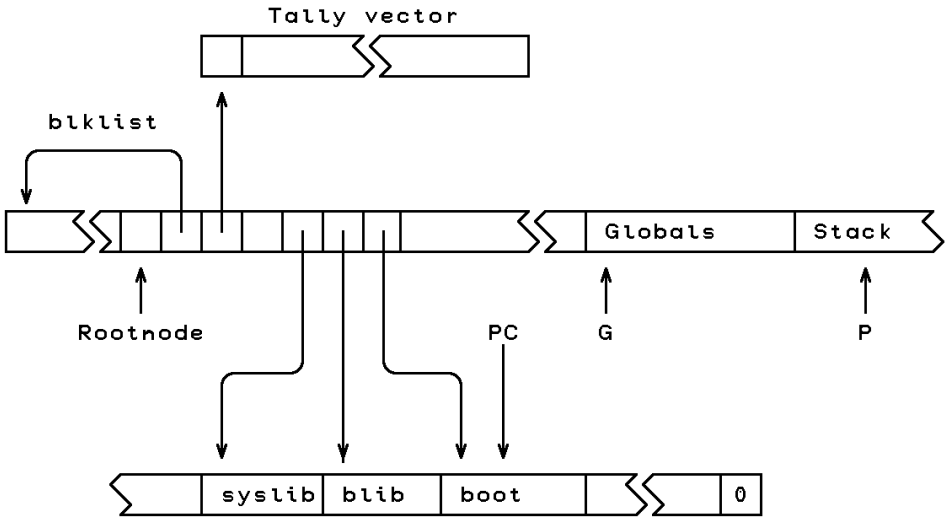


Figure 4.1: The initial state

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

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

At the moment `sys(Sys_interpret,regs)` is first called, only `globsize`, `sys` and `rootnode` have been set in CLI's global vector and so the body of `startroot` 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 `BLIB`, `SYSLIB` and `DLIB`, the following code is used:

```
sys(Sys_globin, rootnode!rtn_blib)
```

If a fault occurs during the execution of `CLI` or a command that it is running, the call of `sys(Sys_interpret,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 Cintsys 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 Chapter 7.

On entry to `startroot`, 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("syscin/cli"))
```

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

4.1.1 Quiet mode execution

Normal execution expects standard input and output to be from the keyboard and to the screen. This allows the user to interact with the system by typing CLI commands. When started in this mode the system outputs an initial message before entering the CLI to read and execute user commands from the keyboard. It generates CLI prompts and echoes keyboard input to the screen.

It is sometimes useful run BCPL programs non interactively in what is called quiet mode. This is done by entering cintsys (or cintpos) with the `-q` option. This enters the BCPL system without generating the initial message, it disables CLI prompts and does not echo keyboard input. In this mode all standard output is explicitly written by the program, except for possible error messages and debugging output. As an example, the program `com/add2.b` which outputs the sum of two integers read from standard input, can be run from a Linux shell by either of the following two commands:

```
echo "111 222" | cintsys -q -- add2
cintsys -q -c add2 111 222
```

Boh commands just output the result 333.

4.2 Bootstrapping Cintpos

Bootstrapping Cintpos is somewhat more complicated than bootstrapping Cintsys 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`. The 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 console 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 initialises the variables used by the standalone debugger. These include the vectors `bpt_addr`, `bpt_instr` and `bpt_dbgvars` which respectively hold breakpoint addresses, 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 `startroot` 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 Cintpos. 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 `dumpmem` mechanism to write the file DUMP.mem representing the current state of the entire Cintcode memory. All other completion codes causes BOOT to invoke the standalone debugger.

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 `startroot` which is described in the next section.

4.2.2 `startroot`

This function creates the Cintpos running environment and loads all the resident system tasks. Finally it enters the Cintpos scheduler which, in turn, gives control to the Idle task which sends a packet to the root CLI task. After some initialisation, this issues the first CLI prompt inviting the user to type in a command. Knowledge of the underlying structures used by Cintpos is key to understanding how Cintpos works. They are described in this section in the order in which `startroot` creates them.

`startroot` is entered by the recursive call of `interpret` from BOOT with a new stack and a different global vector from that used by BOOT. If the interpreter subsequently detects a fault it returns to BOOT's running environment giving control to the interactive debugger allowing the user to inspect the stack and global vector that were current at the time the fault.

Althought `startroot` has three formal parameters `fn`, `size` and `c`, it was entered in a non standard way and these have not been given values. However, the base of `startroot`'s stack is at @fn-3. This points to the zeroth element holding the stack size with all other elements are already set by BOOT to `stackword` (`#xABCD1234`). This stack is turned into a coroutine stack by updating its bottom six elements appropriately. Care is taken to ensure that the code that performs this initialisation is not itself using the stack locations that it is updating. This is one of the reasons why `startroot` was given three parameters.

The function `rootcode` is now called to create the Cintpos resident structures. At this moment the base of the global vector is at `@globsize` (=Global 0), all its elements are filled with words of the form `globword+`*n* (=`#8F8F0000+`*n*), except for `globsize` which holds the upper bound of the global vector, `sys` which holds the entry point

of the sys function, rootnode which points to the rootnode, and currco and colist which both point to the newly created coroutine stack. The other globals are now initialised by two calls of sys(Sys_globin,...).

Cintpos has two vectors tasktab and devtab that provide access to all Cintpos tasks and devices. These are allocated and cleared, and pointers to them are placed in the rootnode.

The resident Cintpos devices are now created. These have device identifiers -1, -2 and -3 corresponding to the clock, the keyboard and the screen. Most Cintsys devices are implemented using separate threads of the underlying operating system. Such devices have device control blocks (DCBs) held their entries in devtab. A DCB has fields used for communication between its device thread and the interpreter. One of these is the work queue of packets sent by client tasks but not yet processed by the device. It has been found that interaction with some device threads is too slow to be satisfactory and so have been replaced by an implementation based on polling by the interpreter. This currently applies to the clock and screen devices. As far as the user is concerned, these devices still have the same indentifiers and still work as before but are faster. An entry in devtab points to a DCB. Devices not using the polling mechanism use threads of the host operating system, other devices are handled entirely by the interpreter thread. The only resident devices currently using a separate threads are the keyboard and TCP devices. Device threads are created using the kernel function createdev defined in sysb/klib.b, and the C code for the resident device threads can be found in sysc/devices.c.

The Cintcode abstract machine can receive interrupts. The mechanism is as follows. If a device wishes to interrupt the interpreter it sets the variable irq to TRUE, and just before the interpreter starts to execute an instruction, if the Cintpos ST register is zero (indicating that interrupts are enabled), it saves the current Cintpos registers and enters the interrupt service routine using the register set in isrregs. The interrupt service routine has its own stack but shares the same global vector a the Cintpos kernel. It always starts execution at the start of the function irqrtn with Cintcode register ST set to 3 to indicate that an interrupt is being serviced. The interrupt sevice routine may return control to the interrupted task or it may enter the scheduler if another task deserves to gain control.

Before creating the resident tasks, startroot initialises a few more rootnode fields. These are rtn_tcblist and rtn_crntask both set to zero since there are currently no Cintpos tasks, rtn_blklist set to the start of the memory block list used by getvec and freevec, rtn_clkintson set to FALSE to globally disable interrupts, rtn_clwkq set to zero representing an empty list of packets for the clock device, and rtn_info set to a cleared table of 50 elements.

The resident tasks are now created using suitable calls of createtask. Each time createtask is called it allocates a task control block (TCB) giving it the next available task identifier and updating the appropriate entry in tasktab to point to it. Such tasks are initially given a state of #b1100 indicating that they are DEAD, not HELD and have no packets in the work queue. The first task to be created is a special one called Idle whose body is in cin/syscin/idle and although createtask will have chosen identifier one for it, this must be replaced by zero and it entry in tasktab removed.

It is given a startup packet and an initial state of `#b1101` indicating it is DEAD, not HELD but has a packet and so can be given control by the scheduler when it is run.

Six more resident tasks are now created, all have state `#b1100`. They are the root command language interpreter that initially waits for commands from the keyboard, and interactive debugging task, the console handler providing communication between the keyboard and tasks, the file handler providing access to disk files, the mailbox handler that provides a mechanism that lets tasks send and receive short messages via named mailboxes and the TCP handler providing TCP/IP communication.

Just after Cintpos starts up the `status` command will output the following.

```
Task  1: Root_Cli          running CLI         Loaded command: status
Task  2: Debug_Task        waiting DEBUG
Task  3: Console_Handler   waiting COHAND
Task  4: File_Handler      waiting FH0
Task  5: MBX_Handler       waiting MBXHAND
Task  6: TCP_Handler       waiting TCPHAND
```

Once the kernel structure and all the resident tasks have been set up, the system can be started by entering the scheduler which is a function called `srchwk` defined in `sysb/klib.b`. It take one argument which is a pointer to the highest priority TCB that could possibly run. It searches through the chain of TCBs that are linked in decreasing priority order looking at only the status field of each. This field is sufficient to tell whether the corresponding task can run or not. It has 4 bits `IWHP`. The `I` bit is a 1 if the task has been interrupted in which case its Cintcode registers will be packed elsewhere in the TCB. The `W` bit is a 1 if the task is suspended in `taskwait` waiting for a packet to arrive from another task or a device. The `H` bit is 1 if the task is in HOLD state indicating that it cannot run even if it otherwise would be ready to do so, and the `P` bit is a 1 if the tasks's work queue is not empty. A task cannot be both interrupted and waiting for a packet and the setting of both the `I` and `W` bits have a special meaning, namely that the task is in DEAD state having no runtime stack or global vector. There are thus 16 posible states a task can have of which only six indicate that it is runnable, they are as follows.

`#b0000`
: This task is runnable but has no packet on its work queue. It is either the current task or it gave up control voluntarily by for instance sending a packet to a higher priority task. When it next gains control it will immediately return from the function that caused it to give up control.

`#b0001`
: This is just like the case above except there is a packet on its work queue.

`#b0101`
: This indicates that the task is waiting for a packet and that one has arrived. It is thus runnable and when given control the first packet on its work queue will be dequeued and returned as the result of the `taskwait` call that caused its suspension.

`#b1000`
: This indicates the task is in interrupted state with an empty work queue. It is thus runnable and when given control it will resume execution using the Cintcode register values saved in the TCB when it was interrupted.

`#b1001`
: This indicates the task is in interrupted state with a non empty work queue. It is thus runnable and when given control it will resume execution using the Cintcode register values save in the TCB when it was interrupted.

`#b1101`
: This is a task in DEAD state (with no stack or global vector) but it now has a startup packet on its work queue. It is thus runnable and when given control will be initialised with a new stack and global vector and its main function `start` in global variable 1 will be called with the startup packet as its first argument. This packet will have been dequeued.

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 7.

adjclock `OFFSET` CIN:y, POS:y, NAT:y

The syntax of the `OFFSET` argument is [-][h][:m], that is: an optional minus sign, followed by an optional number of hours, possibly followed 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 not given an 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 "Your time is up!"
```

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

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

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

This command appends the FROM file on to the end of the TO file. If the TO file does not initially exist, an empty one is created.

bbcbcpl FROM/A,TO/K,REPORT/K,NONAMES/S,MAX/S,SECTLEN/S

CIN:y, POS:y, NAT:y

This invokes a reconstruction of the BCPL compiler for the BBC Microcomputer marketed by my brother's company RCP in the early 1970s. The FROM argument specifies the BCPL source file. The TO argument specifies the desination file for the compiled 16-bit Cintcode. The REPORT argument specifies a file to hold error messages. The NONAMES argument causes the compiler not to embed function names in the compiled code. The MAX argument has no effect in this version of the compiler and SECTLEN controls whether the length of a section of compiled code includes it length in its first word.

This reconstruction was created to possibly help reconstruct the BBC Domesday Project that ran on the BBC Microcomputer using a Philips 12" laser disc. Related commands are mapcode and prbbcocode. For more information, see bcplprogs/bbcmicro/ in the standard BCPL distribution.

bbcbcpl32 FROM/A,TO/K,REPORT/K,NONAMES/S,MAX/S,SECTLEN/S

CIN:y, POS:y, NAT:y

This command is similar to bbcbcpl but generates 32-bit Cintcode suitable for the current BCPL Cintcode system. The BBC BCPL programs may need minor changes needed because the BCPL word length has increased from 16 to 32 bits.

This command is now obsolete since using bbc2bcpl is a more satisfactory way to run BBC BCPL on the modern BCPL Cintcode system.

bbcbcpl32 will soon be deleted.

bbc2bcpl FROM/A,TO/K,HARD/S,EQCASES/S,T64/S,-h/S CIN:y, POS:y, NAT:y

This command convert the BBC BCPL program given by FROM to a destination file given by TO. The result can be compiled and run under the modern BCPL Cintcode system. HARD causes the program to abort on every detected error, EQCASES toggles whether the case of letters in reserved word and identifiers are ignored. By default the case of letters is ignored. Several minor replacements are made, such as LOGAND, LOGOR, LSHIFT, EQ and LE are replaved by &, |, <<. = and <=. Missinf close sections brackets are automatically inserted with the correct indentation when multiple sections are closed by a tagged closing bracket. All dots in identediters are replaced by underscores. The cases of letters used in identifiers is modified to agree with the way each identifier was written when first encountered. To do this the program reads GET files but leave the GET directives unchanged.

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,HARD/S,T32/S,T64/S,
OPT/K,TREE2/S,NOSELST/S 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 using 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|col|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|col|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. If both VER and XREF are specified the xref data is appended to the verification stream. This allows the xref data generated by several separate compilations to be concatenated. The resulting file can be filtered and sorted by the sortxref command. Typical usage is as follows:

```
delete -f rawxref
c compall "ver rawxref xref"
sort rawxref to xrefdata
delete rawxref
```

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 18.

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.

The `HARD` options causes both syntax and translation phase errors to call `abort(100)`. This is useful in commands such as: `c compall hard` allowing each error in a long sequence of compilations to be inspected separately.

The arguments `T32` and `T64` specify whether the target architecture is for 32 or 64 bit BCPL. Note that when using a compiler with a 32 bit word length, manifest constants are calculated using 32 bit integer and floating point arithmetic. If compiling for a 64 bit target integers will be limited to a 32 bt bit range and floating point constants will only have a precision of about 6 of 7 decimal digits. But note that many such constants will still be represented precisely. To obtain the full range and precision on a 64 bit target, a compiler with a 64 bit word length must be used.

The argument `OPT` gives a list of conditional compilation option names consisting of letters, digits, underline and dot, separated by plus signs or any other characters not allowed in option names. These options are declared at the start of compilation of every BCPL section.

The debugging options `TREE` and `TREE2` cause the parse tree to be output before and after the the conversions caused by the FLT feature.

The option `NOSELST` causes the compiler to avoid using the Ocode instructions `SELLD` and `SELST` when compiling the `OF` operator. Less efficient code is compiler using shifts and logical instructions. This option causes the bcpl2sial command not to use the

new SIAL function codes `selld`, `selst` and `xselst`, enabling older Sial codegenerators to continue to work.

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,HARD/S,T32/S,T64/S,`
`OPT/K,TREE2/S,NOSELST/S` 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 11.1. The command `sial-sasm`, described below, can be used to convert Sial into a human readable form and various commands, such as `sial-386`, `sial-alpha` and `sial-arm` will convert Sial to assembly language for corresponding architectures. The `bcpl2sial` command uses the same front end as `bcpl` and so takes the same command 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. This command has largely been superceded by the `xref` option in the `bcpl` command and the related `sortxref` command.

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

This is a simple benchmark program used to test the efficiency of systems implementation languages.

bgpm `FROM,TO/K,UPB/K` CIN:y, POS:y, NAT:y

This is an implementation of Christopher Strachey's GPM macrogenerator. It takes input from the `FROM` file if specified, otherwise it reads from the standard input stream. The `TO` argument specifies the file to receive the macrogenerated result, otherwise this is sent to the standard output stream. The `UPB` argument specified the amount of memory that `bgpm` may use.

A macro call is enclosed in square brackets (`[` and `]`) and contains arguments separated by backslash characters (\). The arguments are macro expanded as they are read in. To avoid macro expansion text can be enclosed within nested quotation marks (`{` and `}`). On reaching the close square bracket at the end of a macro call, the zeroth argument is looked up in the environment of defined macros and macrogeneration continues from the beginning of its value. When the end of this value is reached the expansion of the call is complete and macrogeneration continues from just after the closing square bracket. While a macro call is being expanded, a parameter of the form `^n` is replaced by a copy of the n^{th} argument of the current call. The number n is given as a sequence of decimal digits. The character `''` introduces a comment consisting of all remaining character of the current line followed by all white space characters including newlines up to but not including the next non white space character. The following macros are predefined.

`[def\`*name*`\value]`
This causes a macro with the given name and value to be declared.

`[set\`*name*`\value]`
This updates a named macro with a new value which may be truncated if necessary.

`[eval\`*expression*`]`
This evaluate the given integer expression consisting of numbers and the numeric operators `*`, `/`, `%`, `+` and `-`. Parentheses may be used for grouping and spaces may appear anywhere except within numbers.

`[lquote]`
`[rquote]`
These macros expand to the quotation marks `{` and `}` respectively.

`[eof]`
This macro generates the end of file symbol and can be used to terminate input from the standard input stream.

A simple definition and call is the following.

```
[def\xxx\{arg0 is ^0, arg1 is ^1 and arg2 is ^2}]
[xxx\yyy\zzz]
```

This would generate:

```
arg0 is xxx, arg1 is yyy and arg2 is zzz
```

For an extremely obscure example see: `BCPL/cintcode/perm.bgpm`.

bin-hex `FROM/A,TO/K` CIN:y, POS:y, NAT:y
This outputs the bytes of the `FROM` in hex. For instance, if the file `xxx` was

```
ABCDEFGH
12345678
```

Then the command `bin-hex xxx` would generate

```
 41 42 43 44 45 46 47 48 0A 31 32 33 34 35 36 37
 38 0A
```

Unless `TO` is specified output is sent to the terminal.

bin-x8 `FROM/A,TO/K` CIN:y, POS:y, NAT:y

This outputs the words of the `FROM` in hex. For instance, if the file `xxx` was

```
ABCDEFGH
12345678
```

Then the command `bin-x8 xxx` would generate

```
44434241 48474645 3332310A 37363534 00000A38
```

The default `TO` file name is `JUNK`.

bits2wav `FROM,TO/K,B/N,S/N,A/N,D/N,T/S` CIN:y, POS:y, NAT:y

This commands converts a bit stream file generated by the `raster` command to a `.wav` sound file. `FROM` and `TO` specify the source and destination files. `B` specified the bit rate in bits per second. `S` specifies the `.wav` sample rate which should be 11025, 22050 or 44100. `A` and `D` control the signal filtering and `T` turns on tracing as a debugging aid. The default `FROM` file is `RASTER.bits`. The default `TO` file is `RASTER.wav` and the default sample rate is 11025 samples per second.

bmake `TARGET,FROM/K,TO/K,-m/S,-l/S,-p/S,-r/S,-s/S,-c/S,-d/S` CIN:y, POS:y, NAT:n

This command provides an approximation the `make` command found in other systems. It uses a makefile (normally `bmakefile`) to generate a CLI sequence of commands to bring a specified target up to date. The makefile is expanded using the BGPM macrogenerator and parsed to form a set of pattern rules and explicit rules. Each rule has a target, an optional set of items on which the target depends and a possibly empty CLI command sequence to execute if the target need to be brought up to date.

Pattern rules generate explicit rules when needed. They contain parameters of the form `<tag>`. Within a pattern all tags must be the same and must be declared in the target of the rule.

The optional first argument (`TARGET`) is normally a file name and specifies the target to make. If no target is specified, the target of the first rule is used. The optional `FROM` argument specified the makefile name. The default makefile is `bmakefile`. The optional `TO` argument specifies where the output is to be sent.

The `-m` argument causes `bmake` to output the makefile file after macrogeneration. The `-l` argument outputs the makefile as a sequence of lexical tokens. The `-p` argument outputs the set of rule patterns. The arguments `-r` and `-s` output the explicit rules before and after the application of the rule patterns, respectively. The `-c` argument outputs the sequence of commands required to bring the target up to date. The `-d` argument generates a debugging trace of the execution of `bmake`.

The BGPM macrogenerator is described elsewhere, but the version use in `bmake` uses the following special characters:

%	Comment - skip all characters until a non white space character on a later input line.
[	Start of a new macro call.
!	Argument separator in macro calls.
#	Argument item prefix.
]	End of macro argument list.
{	Open quote character.
}	Close quote character.

A typical macro definition and call is as follows:

```
[def!xxx!{This output results from the call {[xxx!}#1{]}}]
[xxx!yyy]
```

This would generate:

```
This output results from the call [xxx!yyy]
```

The syntax of `bmake` rules is as follows:

target-item `<=` *item ...* *item* `<<` *command-sequence* `>>`

Every rule must have a target item and a body consisting of a possibly empty command sequence enclosed in `<<` and `>>` brackets. The command-sequence is an arbitrary sequence of characters not containing `>>`. The item list may be empty and, if so, the symbol `<=` may be omitted. White space including newlines are allowed anywhere between items.
Pattern rules contain parameter of the form `<tag>` as in:

```
cin/<f> <= com/<f>.b g/hdr.h << c bc <f> >>
```

Such rules are only used when there is no explicit rule for a given target. When a rule pattern is applied all occurrences of its parameter are replaced by the text that allowed the target item to match the required target. So if cin/echo must be brought up to date and has no explicit rule, the above pattern will automatically add the following explicit rule to the set:

```
cin/echo <= com/echo.b g/hdr.h << c bc echo >>
```

A target is out of date if it does not exist or if any of the items it depends on are out of date or have a modify dates later than that of the target. A target is brought up to date by, first, bringing the items it depends on up to date and then executing the CLI command sequence given by the body.

Items may consist of any sequence of characters not including `%`, `[`, `!`, `]`, `{`, `}`, `=`, or white space, and `<` and `>` may only appear in parameters.

In normal use, `bmake` generates a command-command file to bring the target up to date and then returns to the CLI to cause this file to be executed. The `-c` option allows the command-command file to be inspected without execution.

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 `TASK/A,A/S,B/S,C/S,D/S,E/S,ALL/S` CIN:n, POS:y, NAT:n

This Cintpos command is used to break the normal execution of a specified task. The first argument gives the task number and the remaining arguments specify which flags to set. If no flags are specified flag B is set. If `ALL` is specified all the flags from A to E are set.

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

The `c` command allows a file of commands to be executed as though they had just been typed in. The argument *command-file* gives the name of the file containing the command sequence. It first looks in the current directory then the directories specified by the scripts environment variable whose name is in the `rtn_scriptsvar` field of the rootnode, and finally, if that fails, it looks in the directory specified by the root environment variable whose name is in the `rtn_rootsvar` field of the rootnode.

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, optionally, `=` is allowed between the *key* and *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 69) 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`). The `A` switch causes all letters including those in strings to be converted to upper case.

changepri `TASK/N/A,PRI=PRIORITY/N` CIN:n, POS:y, NAT:n

This Cintpos command changes the priority of the specified task to a specified value. If two arguments are given the first identifies the task and the second the new priority. If only one argument is given it is treated as the new priority of the current task. A Cintpos priority can be any positive integer but there is the restiction that no two tasks can have the same priority.

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

This is a test program that checks for errors in the BCPL compiler and Cintcode interpreter.

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

This is a benchmark program to test the efficiency of coroutines.

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

This is a simple coroutine benchmark that bounces a message between two coroutines. On my 1.66Ghz Pentium laptop it outputs the following:

```
0.000> cobounce
Calling the bounce coroutine 10000000 times
About 7812500 coroutine changes per second
done
2.560>
```

This shows that transferring control between coroutines is about 12 time faster than transferring control between Cintpos tasks as demonstrated by the `send` command described below.

compare `FILE1/A,FILE2/A,TO/K,OPT/K` CIN:y, POS:y, NAT:y

This command compares two files outputting a description of how they differ to the `TO` file if specified, or to standard output if not. The `OPT` string consists of items of the form `W`*n*, `M`*n* and `R`*n*, separated by spaces or commas. Each *n* is a number greater than zero. `W`*n* means truncate all input lines to no more than *n* characters. `M`*n* search for up to *n* mismatching lines. `R`*n* means that *n* lines must match before synchronisation is restored after a mismatch.

cosim `-n/n,-s/n,-p/n,-r/n,-t/s` CIN:y, POS:y, NAT:y

This is a demonstration program showing how to write a discrete event simulator using coroutines, and it is also be used as a benchmark. Its arguments can set the variables *n*, *s*, *p* and *r* that configure the test, and the `-t` switch turns on run time tracing to check that the simulator is behaving correctly. For a full description and listing of this program see Section 3.7.3.

dat `TO/K,MSECS/S` CIN:y, POS:y, NAT:y

This commands output the current date and time to the `TO` file, if specified, otherwise it is sent to the standard output stream. The `MSECS` options causes the time to have higher precision. Typical output is as follows:

```
Monday 23-Apr-2010 14:04:12
Monday 23-Apr-2010 14:04:14.392
```

date `TO/K` CIN:y, POS:y, NAT:y

This commands output the current date to the `TO` file, if specified, otherwise it is sent to the standard output stream. Typical output is as follows:

```
Monday 23-Apr-2010
```

delete `,,,,,,,,,-f/S` CIN:y, POS:y, NAT:y

This command will delete up to ten given files. If the `-f` argument is given, no error message is generated if any file to be deleted does not exist.

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.

dumpmem `ON/S,OFF/S` CIN:y, POS:y, NAT:y

The `ON` switch causes Cintsys or Cintpos to set the `dumpflag` in the rootnode to `TRUE`. `OFF` causes the `dumpflag` to be set to `FALSE`. If the `dumpflag` is `TRUE` when a fault occurs or when a return from the interpreter occurs, the entire Cintcode memory is output in a compacted form. Such memory dumps are sent to the file `DUMP.mem` for later inspection by commands such as `sysdebug`, `dumpsys`, `posdebug` and `dumppos`. Calling `dumpmem` without arguments causes an immediate memory.

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

This outputs a readable form of a Cintpos memory dump specified by the `FROM` argument. If `FROM` is not given it uses the file `DUMP.mem`. The output is sent to the `TO` file if given, otherwise it goes to standard output.

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

This outputs a readable form of a Cintsys memory dump specified by the `FROM` argument. If `FROM` is not given it uses the file `DUMP.mem`. The output is sent to the `TO` file if given, otherwise it goes to standard output.

easter `YEAR/N,CYCLE/S` CIN:y, POS:y, NAT:y

This command outputs the date of Easter Sunday for 10 years from the year given by the `YEAR` argument. If the `YEAR` argument is nott given the output starts from the current year.

After many years the sequence of Easter dates repeats. Giving the `CYCLE` option causes the program to discover the length of this cycle.

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

This command outputs its first argument `TEXT`, if given. The text will be followed by a newline unless the switch `N` is set. If the `TO` argument is given, text is sent to the specified file othewise it goes to the standard output stream. The `APPEND` switch causes the output to be appended to the `TO` stream, after creating an empty file if necessary.

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 yet been modified to run on this version of the system.

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

This Cintpos command causes a CLI task to commit suicide.

enlarge `/A,TO/K` CIN:y, POS:y, NAT:y

This command output a large version of its first argument either to file or to standard output. For instance: `enlarge Hello` will generate the following:

```
##    ##  ########  ##        ##         ######
##    ##  ########  ##        ##        ########
##    ##  ##        ##        ##        ##    ##
########  ######    ##        ##        ##    ##
##    ##  ##        ##        ##        ##    ##
##    ##  ##        ##        ##        ##    ##
##    ##  ########  ########  ########  ########
##    ##  ########  ########  ########   ######
```

fact CIN:y, POS:y, NAT:y

This is a simple example program used in the console session demonstration presented on page 8.

fail `RC/N,REASON/N` CIN:y, POS:y, NAT:y

This command returns to the CLI with the specified return code and second result. The default return code is 10 and the default second result is zero. Unlike the `quit` command described below, it does not cause the current command-command to terminate.

failat `FAILLEVEL/N` CIN:y, POS:y, NAT:y

This sets the CLI fail level to its argument if given, otherwise it outputs the current setting. The CLI only issues a warning message if a command yields a return code greater than or equal to the fail level value.

fast CIN:y, POS:y, NAT:n

This is a program selects the fast interpreter.

getlogname `NAME` CIN:y, POS:y, NAT:y

This command outputs the value of a given logical variable name. If none is given it lists the names and values of all logical variables. The list of logical name value pairs is held in the root node element `rtn_envlist`.

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

This is Cintpos command whose purpose test a system by generating a sequences of timed events specified by a script.

help `,,,,,,,,,,,,,,,#HELPDIR/K,#TO/K,#TRACE/S` CIN:y, POS:y, NAT:y

This command is meant to provide a help facility but has not yet been transferred to Cintsys or Cintpos.

hex-bin `FROM/A,TO/K` CIN:y, POS:y, NAT:y

This is the inverse of the `bin-hex` command. It reads pairs hex digit outputting the corresponding 8-bit bytes.

hexdump `FROM/A,N/N,P/N,RL/K/N,RLB/K/N,TO/K,`
`X1/S,X2/S,X4/S,LIT/S,BIG/S` CIN:y, POS:y, NAT:y

This program dumps a file specified by `FROM` in a combination of hex and character forms. If either `RL` or `RLB` is given the file is treated as a sequence of records. `RL` gives the record length in BCPL words and `RLB` gives it in bytes. The `P` and `N` arguments give the number of the first record to dump and `N` specifies how many to dump. If neither `RL` nor `RLB` is given `P` gives the number of the first byte to dump and `N` gives the number of bytes to dump. `X1` causes the file to be dumped as a sequence of individual bytes. `X2` causes the file to be dumped as a sequence of 16-bit words, and `X4` causes the file to be dumped as a sequence of 32-bit words. `LIT` or `BIG` specify whether to use little-ender or big-ender ordering when dumping words. It neither are specified the enderness of the current computer is used. If the file `bc` is as follows:

```
#!/home/mr/distribution/BCPL/cintcode/cintsys -s
.k file/a,arg
echo "bcpl com/<file>.b to cin/<file> hdrs BCPLHDRS <arg>"
bcpl com/<file>.b to cin/<file> hdrs BCPLHDRS <arg>
```

then the command: `hexdump bc 64` would generate the following:

```
Dump of bc  from 0 to 63 little-ender mode

    0/    0: 682F2123 2F656D6F 642F726D 72747369   #!/h ome/ mr/d istr
   16/    4: 74756269 2F6E6F69 4C504342 6E69632F   ibut ion/ BCPL /cin
   32/    8: 646F6374 69632F65 7973746E 732D2073   tcod e/ci ntsy s -s
   48/   12: 206B2E0A 656C6966 612C612F 650A6772   ..k  file /a,a rg.e
```

hold `TASK/N/A` CIN:n, POS:y, NAT:n

This is only available under Cintpos. It causes the specified task to be put into HOLD state to stop it being available to run. Its inverse is `unhold` described below.

idvec `ADDRESS/A` CIN:n, POS:y, NAT:n

This Cintpos command attempts to identify the vector at a given address. Two example call are given below:

```
0.000 1> idvec 23522
Stack of task 4
0.000 1> idvec 15994
Code section of task 5: MBXHAND
0.000 1>
```

if `,NOT/S,WARN/S,ERROR/S,FAIL/S,EQ/K,VAREQ/K,EXISTS/K:` CIN:y, POS:y, NAT:y

This command normally ends with a semicolon and the remainder of the line is conditionally executed by the CLI depending on whether the `if` condition is satisfied. The return code and second result of the previous CLI command are held in the globals `cli_returncode` and `cli_result2`. If one of `WARN`, `ERROR` or `FAIL` was given, the `if` command tests whether the previous command's return code greater or equal to warn(=5), error(=10) or fail(=20). If the `EQ` argument was given, it tests whether the return code is the same as the first argument. If `VAREQ` is given, it specifies is a logical variable name and the value of this variable is compared with the first argument. The `EXISTS` argument is a file name whose existence is tested. The `NOT` switch complements the condition.

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(Sys_quit,-1)` or `sys(Sys_quit,-2)` as described on page 83.

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/A` CIN:y, POS:y, NAT:y

This command has no effect. Its sole purpos is be the destination of `skip` commands.

library `FROM,OVERRIDE/S,CANCEL/K,LIST/S,-g/S,TO/K` CIN:n, POS:y, NAT:n

This rather dangerous command allows the user to add or delete sections of resident system code. If the `FROM` argument is given the specified file is loaded and its sections added to the end of the chain of BLIB sections pointed to by the root node field `rtn_blib`. If `OVERRIDE` is given the newly loaded sections are allowed to replace previous ones with the same section names, otherwise all newly loaded sections must have names

distinct from those already in the BLIB chain. The `CANCEL` argument specifies the name of a section to remove from the BLIB chain. The `LIST` switch argument causes a list of the section names in the BLIB chain to be output. The argument `-g` causes a list of all the global functions defined in the BLIB chain to be output including the names of the sections they are in. The `TO` argument specifies the name of a file where the output is to be sent. It is often useful to sort this file using `sortlines`. Normally the `library` command is only used during the initialisation of special purpose versions of Cintsys or Cintpos, or when one wishes to see which functions are defined in `BLIB`.

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 `,,,,,,,,,,,TO/A/K,STKSIZE/K,GLOBSIZE/K` CIN:y, POS:y, NAT:y

This command is used by the native code version of BCPL to generate a C program used to initialise a native code compilation of BCPL program. It takes a list of BCPL source files and writes to the `TO` file a C program that will perform the necessary runtime initialisation of them. This program also sets the runtime stack size and global vector size to 50000 and 1000, respectively, unless overridden by the `STKSIZE` and `GLOBSIZE` arguments. The resulting C program should compiled and linked with the native code compilations of the BCPL files and various library modules. For more information look in the directory `BCPL/natbcpl` of the standard BCPL distribution. An example of the use of `makeinit` is given on page 246.

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

This command outputs 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. `BLOCKS` outputs a list of all blocks whether allocated or free in the block chain used by `getvec`. `CODE` outputs a list of all code sections currently in memory. `MAPSTORE` output the code sections and function entry points currently in memory, and `PIC` outputs a picture of what memory is currently allocated.

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

This command outputs 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. `BLOCKS` outputs a list of all blocks whether allocated or free in the block chain used by `getvec`. `CODE` outputs a list of all code sections currently in memory. `MAPSTORE` output the code sections and function entry points currently in memory, and `PIC` outputs a picture of what memory is currently allocated.

mapcode `FILE/A,TO/K,-r/N,-t/S,-f/S` CIN:y, POS:y, NAT:y

This command inspects both 16-bit Cintcode and 6502 machine code used on the BBC Microcomputer displaying such data in a readable form. `FROM` specifies the data file, `TO` specified the output file. The `-r` specifies the address of the first byte of machine code to display. The `-t` argument turns on debugging tracing and `-f` causes extra information to be displayed about each byte of data being inspected.

mbxcli `MBXNAME` CIN:n, POS:y, NAT: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/N,-d/N,-b/K` CIN:n, POS:y, NAT: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 milli-seconds 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/N,-d/N,-b/K` CIN:n, POS:y, NAT: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 milli-seconds 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

This command compiles an MCPL program into Mintcode. See the MCPL distribution for more details.

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

This command compiles an MCPL program into MIAL.

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

This translates the MIAL form of an MCPL program into Pentium assembly language.

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

This translates the MIAL form of an MCPL program into a mnemonic form.

mkdata `NAME,SIZE/N` CIN:y, POS:y, NAT:y

This creates a file with given name and size. The default name is `junk` and the default size is 4096*3+10 bytes. Byte *i* of the created file is *i* `MOD 256` except every 64th character is a newline and the first 6 characters of every line hold a decimal number giving the position of the first character of that line.

mkjunk `NAME,SIZE/N` CIN:y, POS:y, NAT:y

This creates a file as described in the `mkdata` command and then tests random access to this file by overwriting some of its bytes.

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

This Cintpos command creates a new CLI task.

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

This command replaces the specified file with one in which line endings have been replaced by those appropriate 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 sometimes used for benchmarking purposes.

origbcpl2bmp CIN:y, POS:y, NAT:y

This is a program to convert the raster data corresponding to the self compilation of origbcpl.b into a .bmp image showing how memory is used by the origbcpl compiler compiling itself. When the `bash` shell under Linux this image can be built by typing the following commands.

```
cd $(BCPLROOT)
rastsys
slow
raster
origbcpl com/origbcpl.b to junk
c bc origbcpl2bmp
origbcpl2bmp to origbcpl.bmp
gimp origbcpl.bmp
```

This image appears in this manual.

playback `FROM/A,WAIT/S,NOTIME/S` CIN:y, POS:y, NAT:y

This plays back a console session recording made using the `record` command.

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

This copies a specified recording file (created by the `record` command) to the specified output enclosing timing bytes in square brackets.

playtime `FROM/A` CIN:y, POS:y, NAT:y

This outputs how long a specified recording (created by the `record` command) will take to playback.

posdebug `FROM` CIN:y, POS:y, NAT:y

This is an interactive debugger that allows the user to inspect a given Cintpos memory dump file. The default file name is `DUMP.mem`. See `dumpmem` described above.

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

This command converts a 16-bit OCODE file used by the BCPL compiler for the BBC Microcoputer into a more readable form. `FROM` specifies the Ocode file. The `TO` argument specifies the destination file. If it is missing it sends the result to the screen.

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

This command is primarily for systems that do not have the concept of a current working directory. 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(Sys_getprefix)` described on page 85. 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 all preloaded commands and their sizes. Preloading improves the efficiency of command execution and is also useful in conjunction with the `stats` command, see below. Preloaded commands can be removed using the `unpreload` command.

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 sends 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,P0/S,P1/S,P3/S,P4/S,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. The switch parameters `P0` to `P4` select commonly used prompt formats. The CLI generates prompts using a call of the following form.

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

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*, *secs* and *msecs* represent the current time of day. The default prompt format under Cintpos is: `"%+%n> "` and under the other systems is: `"%5.3d> "`. An example of how it might be used is as follows.

```
0>
0> prompt "%+%+%z2:%z2:%z2 %-%-%-%-%-%5.3d> "
15:11:52 0.000>
15:11:55 0.000> bench100

bench mark starting, Count=1000000

starting

finished
qpkt count = 2326410  holdcount = 930563
these results are correct
end of run
15:12:14 10.690>
```

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

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

This causes a CLI command-command to terminate returning a completion code of zero unless overridden by the `RC` argument. If `REASON` is given it is placed in `result2`. This command differs from `fail` since it terminates the execution of a command-command while `fail` allows a command-command to continue run.

rast2ps `FROM,SCALE/N,TO/K,ML/N,MH/N,MG/N,FL/N,FH/N,FG/N,`
`DPI/K/N,INCL/K,A5/S,A4/S,A3/S,A2/S,A1/S,A0/S` CIN:y, POS:y, NAT:y

This command has been superseded by programs such as `com/origbcpl2bmp.b`. This command used to be used to converts a raster data file (written using the `raster` command described below) into a postscript file suitable for printing.

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 default values of `MH` and `FH` are given by the `FROM` file. The default values of `ML` and `FL` are both zero. Unless `MG` and `FG` are given, suitable values are chosen automatically. The units are in bytes. The parameters `FL` and `FH` specify the low and high limits of the instruction count axis to be displayed. `FG` specifies the separation of the grid line on the memory axis. `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. This file allows annotations to be made in the picture. The file `cintcode/origbcplps.h` 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 edited by hand.

rast2wav `FROM,TO/K,n/N,s=secs/N,r/N,d/N,stereo/N,t/N` CIN:y, POS:y, NAT:y

This command converts a raster data file (written using the `raster` command described below) into `.wav` sound file based on the pattern of memory accesses during the CLI command following the call of `raster`.

The `FROM` parameter specifies the name of the raster data file. `RASTER` is the default. The `TO` parameter specifies the name of the `.wav` file to be generated. `RASTER.wav` is the default. By default, the program only generated notes that are equal temperament semitone (12 per octave), but the `n` argument allows the user to specify a different number of notes per octave, susch as 24 or 41. The duration of the generated sound file can be specified using the `s` or `secs` argument. The default `.wav` sample rate is 44100 per second, but 22050 or 11025 can be specified using the `r` argument. By default notes are numbered upwards from 0 to 60 with 12 notes per octave the lowest note id C two octaves below middle C. The `d` option is a debugging aid that causes all notes other that a specified one to be silent. This allows the algorithm to choose when to sound a note to be tested including how it volumes envelope changes. The `t` argument is not yet implemented but willin due course generate trace output during the execution of `rast2wav`. This command was written the sound generated by EDSAC 2's loudspreaker in the 1960s was a remarkably useful debugging aid.

As a demonstration, `origbcpl.wav` or `origbcpl.mp3` is the sound of the early version of the BCPL compiler compiling itself, and the following command sequence from a `bash` prompt will generate an approximation to Bach's Invention no 10, bwv

784. These demonstrations are not yet ready.

```
rastsys
c bc bwv784
raster
bwv784
rast2wav
ctrl-c
audacity RASTER.wav
```

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

This command controls the generation of raster data 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 85. If `raster` is called without the `BITS` options it activates the rastering mechanism for the duration of the next CLI command. Without the `BITS` option the default `TO` file is `RASTER`. The format of this file is outlined on page 85.

The `COUNT` argument specifies the number of Cintcode instructions to obey per raster line. The default is 1000. The `SCALE` argument gives the number of byte addresses per unit on the memory axis. The default being 8.

The raster data file can be processed and converted to Postscript using the `rast2ps` command described above. Typical use of the `raster` command is following script, starting from a linux `bash` prompt:

```
rastsys
raster
origbcpl com/origbcpl.b to junk
rast2ps incl origbcplps.h
ctrl-c
ps2pdf RASTER.ps
okular RASTER.pdf
```

This will create a `.pdf` file for an early version of the BCPL compiler compiling itself, similar to that shown in Figure 4.2. For a more detailed view of the parse tree while `SYN` is being compiled, try:

```
rastsys
raster
origbcpl com/origbcpl.b to junk
rast2ps incl origbcplps.h ml 350000 mh 500000 fh 6000000
ctrl-c
ps2pdf RASTER.ps
okular RASTER.pdf
```

Noth that `rast2ps` is now obsolete having been superseded by programs such as `com/origbcpl2bmp` which can be used to convert the raster data files into a .bmp image files.

If `raster` is called with the `BITS` option, the next CLI command will generate a bit stream file corresponding to the fifth bit of every Cintcode byte address accessed. The default `TO` file name is `RASTER.bits`. This file contains one byte for every 8 memory

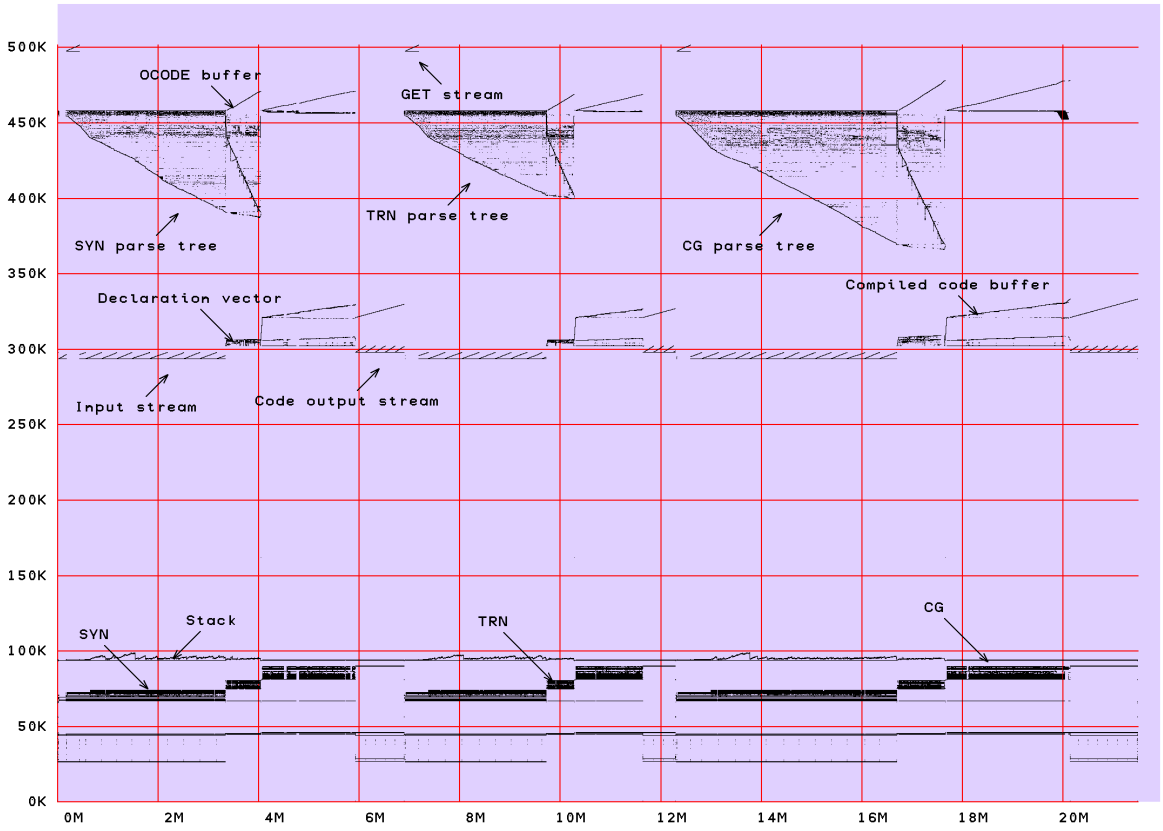


Figure 4.2: Self compilation memory-time graph

references so can become very large. It can be converted to a `.wav` sound file using the `bits2wav` command.

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

This Cintpos command starts sending a recording data including timing information of the current console sessions to the specified file. The recording is stopped by the command `record off`. See the commands `playback`, `playfast`, and `playtime`.

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

This attempt to reposition CLI input to the start of the current command line thereby causing it to be executed again. For example:

```
wait 3; echo hello; repeat
```

will output `hello` to the screen every 3 seconds until interrupted by the D flag (set by `@d`).

run ***command-line*** CIN:n, POS:y, NAT:n

This Cintpos command creates a new CLI task giving it *command-line* to execute. On complete this new CLI task commits suicide.

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

This is part of the Cintpos bounce demonstration. It repeatedly 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. On my 1.66Ghz Pentium laptop, `send` runs for 3.19secs corresponding to about 630000 task changes per second.

setflags `TASK,A/S,B/S,C/S,D/S,E/S,QUIET/S` CIN:n, POS:y, NAT:n

This Cintpos command sets the specified flags in the task control block of the given task. Unless `QUIET` is given it outputs the previous setting of the flags.

setlogname `NAME,VALUE` CIN:y, POS:y, NAT:y

This command sets or possible displays Cintsys or Cintpos logical variables. These must not be confused with shell environments variables described in Section 3.6. Cintsys and Cintpos logical variables are held in a linked list held in the rootnode element `rtn_envlist`. If both `NAME` and `VALUE` are given, the given logical variable name is given the specified value, but if no value is given the specified variable is unset. If `setlogname` is called without arguments, the names and values of all logical variables are output. A running program can lookup and set logical variables using the functions `getlogname` and `setlogname`.

setroot `ROOT,PATH,HDRS,SCRIPTS` CIN:y, POS:y, NAT:y

If no arguments are given it just outputs the current settings of the four environment variable names. Otherwise, the specified variables are given new names.

shellcom `COMMAND/A` CIN:y, POS:y, NAT:y

This command causes its argument to be processed by the command language interpreter shell of the underlying operating system (typically Linux or Windows). It does not return until the shell has completed processing the command.

sial-arm `FROM,TO/K` CIN:y, POS:y, NAT:y

This command converts the Sial intermediate code generated by `bcpl2sial` to the equivalent assembly language for machines using the ARM processor.

sial-386 `FROM,TO/K` CIN:y, POS:y, NAT:y

This command converts the Sial intermediate code generated by `bcpl2sial` to the equivalent assembly language for i386 machines such as Pentiums.

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

This command converts the Sial intermediate code generated by `bcpl2sial` to the equivalent assembly language for DEC Alpha machines.

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

This command converts the Sial intermediate code generated by `bcpl2sial` into a human readable form.

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

This command converts the Sial intermediate code generated by `bcpl2sial` to the equivalent assembly language for VAX machines.

skip `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.

slow CIN:y, POS:y, NAT:n

This is a program selects the slow interpreter.

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

This command sorts the lines specified by the `FROM` file sending the result to the `TO` file, removing duplicate lines. Output is sent to the screen if the `TO` parameter is not given. This can be used to sort the data generated by the command: `library -g to junk`.

sortxref `FROM/A,TO/K,FNS/S` CIN:y, POS:y, NAT:y

This command sorts the lines specified by the `FROM` file sending the result to the `TO` file, removing duplicate lines. Output is sent to the screen if the `TO` parameter is not given. Only lines lines containing `G:`, `M:`, `F:` or `S:` are included, and if `FNS` is specified, only lines also containing `FN` or `RT` are included. This is useful when processing cross reference data generated by the BCPL compiler when the `XREF` parameter is specified. A typical cross reference listing can be found in `cintcode/xrefdata`.

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. Without an argument it 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 on`	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 `TASK,FULL/S,TCB/S,SEGS/S,CLI=ALL/S` CIN:n, POS:y, NAT:n

This Cintpos command outputs information about all currently existing Cintpos tasks.

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

This is a program to demonstrate various synchronisation mechanisms implemented using coroutines and multi-event tasks.

sysdebug `FROM` CIN:y, POS:y, NAT:y

This is an interactive debugger that allows the user to inspect a given Cintsys memory dump file. The default file name is `DUMP.mem`. See `dumpmem` described above.

sysinfo CIN:y, POS:y, NAT:y

This outputs some information about the current BCPL system and the host machine on which it is running. Typical output is as follows:

```
This version of BCPL is running on a little ender machine
The BCPL word length is 32 bits

The host address size = 64 bits
```

system CIN:y, POS:y, NAT:y

This command outputs a message indicating whether the current system is Cintsys, Cintpos or Unknown. It determines which by inspecting the rootnode field `rtn_system`.

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

This command calls `writef` with the given format and the current task number as the second argument. The default format is `"Taskid=%n*n"`.

tcpaddr `HOST,PORT` CIN:n, POS:y, NAT:n

This attempts to output the IP address and port number given the names of the host and port.

tcpbench `-n/K,-k/K,-s/K,-h/K,-t/S,master/s,slave/s` CIN:n, POS:y, NAT:n

This is a benchmark program to test the efficiency of TCP/IP communication. For information about what it does and how to use it, see the comments at the start of the source code.

tcpcli `PORT,NOPROMPT/S` CIN:n, POS:y, NAT:n

This command creates a new CLI task communicating through the given port. The default port number is 8000. If `NOPROMPT` is specified the newly created CLI will not issue prompts.

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

This outputs the list of Cintpos TCP/IP devices that currently exist. The list includes information about sockets, states and associated hosts and port numbers.

tcprx `HOST,PORT` CIN:n, POS:y, NAT:n

This is a TCP/IP demonstration program to be used in conjuction with `tcptx`. It will output data received from a specified host via a specified port. If no host is specified wait for a connection from any host. The default port number is `9000`.

tcptest `-n/K,-k/K,-s/K,-h/K,-t/S` CIN:n, POS:y, NAT:n

This is a TCP/IP test program. See its source code for details.

tcptx `HOST,PORT,N` CIN:n, POS:y, NAT:n

This is a TCP/IP test program to be used in conjunction with `tcprx`. It attempts to send the message `hello world` to a specified host via a specified port. The number of times the message is sent is given by the `N` argument.

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

This command tests the real time clock, outputting a line such as:

```
days=14876 hours=11 mins=59 secs=11 msecs=982
```

time `TO/K,MSECS/S` CIN:y, POS:y, NAT:y

This command outputs the current time of day to the `TO` file, if specified, otherwise it is sent to the standard output stream. The `MSECS` options causes the time to have higher precision. Typical output is as follows:

```
14:12:36.069
```

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.

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. Its output should be compared with that generated by the `hexdump` command.

unhold `TASK/N/A` CIN:n, POS:y, NAT:n

This Cintpos command resets the HOLD status bit of a specified task. That task is then immediately available to run unless suspended of other reasons.

unpreload `,,,,,,,,,,ALL/S` CIN:y, POS:y, NAT:y

This command will remove up to 10 specified preloaded commands from the Cintcode memory. The `ALL` switch will cause all preloaded commands to be removed. Commands can be preloaded into memory using the `preload` which can also be used to list all preloaded commands.

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

This command output information about blocks of Cintcode memory that are currently allocated. Typical output (from Cintpos) is the following:

```
    3:  12     4:   2     6:   1    15:   2    22:   1    23:   7
   27:   4    28:   1    41:   1    80:   1   200:   2   291:   1
  306:   2   316:   1   406:   1   462:   1   500:   1   506:   3
  571:   1   597:   1   757:   1   982:   1  1000:  10  1006:   6
 1025:   2  1901:   1  2422:   1  3303:   1 20000:   1
```

This indicates, for instance, that there are currently 7 blocks of requested size 23 allocated.

wait `N/N,SEC=SECS/S,MIN=MINS/S,UNTIL/K` CIN:y, POS:y, NAT:y

This causes the CLI to wait for a specified number of seconds or minutes, or until a specified time is reached.

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

This command attempts to give the reason why the previous command failed. For fun you can type `why` several times.

x8-bin `FROM/A,TO/K` CIN:y, POS:y, NAT:y

This converts a file of 32-bit words in hex into a file of the corresponding bytes. For instance, it will convert the file:

```
44434241 48474645 4C4B4A49 504F4E4D 54535251 58575655 310A5A59 35343332
39383736 00000A30
```

to

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
```

xcmpltest CIN:y, POS:y, NAT:y

This is a test program that checks for errors in the XBCPL compiler and extended features in the Cintcode interpreter.

xcdecode FROM/A,LIST/S,BIN/S CIN:y, POS:y, NAT:y

This command is the inverse of xcencode. With the LIST option it will inspect the FROM file listing the names of the files it contains. Without the LIST option it will extract and decode these files. If BIN is set, files are written using binwrch so that carriage return characters ('*c') are not ignored. All characters before the first file separator are ignored.

xcencode FILE,LIST/K,TO/K/A,BIN/S CIN:y, POS:y, NAT:y

This command is designed to encode one or more files in such a way that they can be passed as the body of an email message without interferring with the email mechanism. It uses a simple form of run length encoding to reduce the size of the resulting file. Either FILE or LIST or both must be supplied. If given FILE is the first filename to be encoded followed by those given in LIST file, if present. If BIN is set, files are read using binrdch so that carriage return characters ('*c') are not ignored. Each encoded file is preceded by a separator of the form:

```
#####filename#
```

followed by the encoded file in which all characters with ASCII codes in the range 33 to 126 except for '#', '=' and '.' are copied, spaces are replaced by dots ('.') and all other characters (including '#' '=' and '.') are encoded by #hh where hh is the ASCII code in hex. The encoded files are broken into lines of about 50 characters. The last file to be encoded is terminated by ######+#.

Such xencode'd files can be decoded by the xdecode command.

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 which should be a `writef` format string since it used in the CLI as follows:

```
writef(cli_prompt,
       cpumsecs, // msecs used by last command
       taskid,   // The task number, if running under Cintpos
       hours, mins, secs, msecs) // The time of day
```

where `hours`, `mins` and `secs` correspond to the current time of day. On single threaded BCPL systems `taskid` is set to 1.

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 temporary 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 executing command. It is used by the CLI to unload commands after they have been run.

Chapter 5

Console Input and Output

When `cintsys` or `cintpos` is started a stream is opened to receive 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 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 unless running in quiet mode.

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. Unless running i quiet mode its echoed symbol is 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`. It is also possible to read keyboard characters by polling them using the call `sys(Sys_pollsardch)`. This yields the next character if one is available, otherwise it returns `pollingch=-3`, allowing the program to do other work before trying again.

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

The stream control block for the screen 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. Calls of `wrch` places characters in this buffer, and when a newline or newpage character

is written, or when 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`.

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.

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

Chapter 6

Cintpos Devices

Cintpos allows asynchronous communication with peripheral devices using the `qpkt` and `taskwait` functions. If the `pkt_id` field of packet given to `qpkt` is negative, the packet is sent to the identified device. It is returned when the device has completed the requested operation. Most devices have device control blocks (DCBs) that contain device related data. There is a device table pointed to by `rootnode!rtn_devtab` whose upper bound is held in its zeroth element. The n^{th} element of the device table is zero if the device does not exist, otherwise it points to the DCB of device -n. Most devices are implemented using threads of the host operating system, but some devices such as the clock and screen are special and use a polling mechanism implemented entirely within the interpreter thread. The extra overhead for this is small since the interpreter only performs the polling operation about once every 10000 or so Cintcode instructions. This figure is typically adjusted to cause polling to take place about once per millisecond. When Cintpos has no work to do it should enter the Idle task and stop executing Cintcode instructions so that other programs can run. For the polling mechanism to work, such suspensions must be short. This is normally implemented using the `waitirq` `sys` function with a short timeout. Each time `waitirq` returns, a counter in the intepreter is set to zero to cause the polling mechanism to be activated.

The resident Cintpos devices are described below.

6.0.1 The Clock Device

This device has identifier `-1` and is treated specially by both `qpkt` and the interpreter. The `pkt_arg1` field of its packet holds the number of milliseconds that the packet should remain with the clock before being returned. The time stamp of when it should be returned is calculated by `qpkt` and placed in the `pkt_res1` and `pkt_res2` fields of the packet. It is then inserted into the time ordered clock queue held in `rootnode!rtn_clwkq`. Every time the interpreter performs the polling operation it tests the packets at the start of the clock queue returning though that have expired to their task.

6.0.2 The Keyboard Device

This device has identifier `-2` and is currently not treated specially, and so it has a DCB, and a device thread that is continually trying to read character from standard input which is normally the keyboard. Packets for this device are placed on the end of the work queue held in the `dcb_wkq` field of the DCB. When a character becomes available it is placed in the `pkt_res1` field of the first packet in the queue before returning the packet to its task.

It is planned to modify keyboard packets to allow them to handle timeouts. This will be done by setting the `pkt_arg1` field to a timeout value. If it is is negative no timeout is used and the packet will remain with the device until a character is received, otherwise it specifies a timeout in milliseconds. If no character is received within that time, `pollingch` (=-2) is returned in the `res1` field, but if a character becomes available within that time it it returned in the normal way.

6.0.3 The Screen Device

This device has identifier `-3` and is treated specially. The `pkt_arg1` field of the packet holds the next character to send to the screen and when this transfer is complete the packet is returned to the client task. Normally output to the screen causes no real time delay.

6.0.4 TCP/IP Devices

TCP/IP devices provide a mechanism to communicate with other machines over the internet. The `pkt_type` field specified the TCP/IP operation required and the argument field provide additional information about the request. The possible packet type are as follows.

Tcp_name2ipaddr `arg1:` *name*

This looks up the URL *name* and returns its IP address. Names such as `127.0.0.1` are allowed.

Tcp_name2port `arg1:` *name*

This looks up the the given port name and returns its its number.

Tcp_socket

This attempts to create a port for a two way byte stream using the IPv4 protocol. If the result is `-1` there was an error, otherwise it returns the number of the new socket.

Tcp_reuseaddr arg1: ***sock*** `arg2:` *flag*

If *flag*=`1` this modifies the socket *sock* to allow reuse of local addresses, otherwise these are disallowed. A result of zero indicates success.

Tcp_sndbufsz arg1: ***sock*** `arg2:` *size*

This sets the send buffer size of the given socket to *size* bytes. A zero result indicates success.

Tcp_rcvbufsz arg1: *sock* arg2: *sz*

This sets the receive buffer size of the given socket to *size* bytes. A zero result indicates success.

Tcp_bind arg1: *sock* arg2: *ipaddr* arg3: *port*

This assigns local host and port numbers to the specified socket. A zero result indicates success.

Tcp_connect arg1: *sock* arg2: *ipaddr* arg3: *port* arg4: *timeout*

This attempts to establish a connection to a remote host via the given socket within the given timeout. If *timeout* is greater than zero it specifies a timeout time in milli-seconds, if it is zero there is no timeout and if it is `-1` polling will be used but this is not yet implemented. The result is zero if a connection was established, otherwise it is negative and the second result indicates why the connection was not established. A value greater than zero indicates an error, the value `-1` the connection was closed by the remote host, `-2` indicates that the connection was not established within the timeout period, and `-3` indicates that when polling the connection has not yet been established.

Tcp_listen arg1: *sock* arg2: *n*

This causes the specified socket to be willing to accept incoming calls from remote hosts. The queue limit for incoming connections is specified by *n*. A zero result indicates success.

Tcp_accept arg1: *sock* arg2: *tcp,* arg4: *timeout*

BEWARE: the implementation does not yet quite match the following specification. This attempts to accept a connection from a remote host via a listening socket within a specified timeout period. If *timeout* is greater than zero it is the timeout period in milli-seconds, if it is zero there is no timeout and if it is negative the packet is returned immediately having accepted a connection if possible. A positive result indicates success and is the number of a new socket to to be used by the connection. A negative result indicates failure with a reason in the second result. A second result of `-1` indicates the connection was closed by the remote host, `-2` means a connection was not accepted within the timeout period, and `-3` indicates that there is currently no connection to accept when polling.

Tcp_recv arg1: *sock* arg2: *buf* arg3: *len* arg4: *timeout*

This attempts to read up to *len* bytes into the given buffer from the specified socket within a specified timeout period. If *timeout* is greater than zero it is the timeout period in milli-seconds, if it is zero there is no timeout and if it is negative the packet is returned immediately with as many characters as are currently available. A negative result indicates failure with a reason given in the second result, otherwise it is the number of bytes actually read.

Tcp_send arg1: *sock* arg2: *buf* arg3: *len* arg4: *timeout*

This attempts to send *len* bytes from the given buffer via the specified socket within

a specified timeout period. If *timeout* is greater than zero it is the timeout period in milli-seconds, if it is zero there is no timeout and if it is negative the packet is returned immediately having written as many bytes as are currently possible. A negative result indicates failure with a reason given in the second result, otherwise it is the number of bytes actually sent.

Tcp_close `arg1:`*`sock`*

This closes the specified socket. A zero result indicates success.

Chapter 7

The Debugger

Both Cintsys and Cintpos have interactive debuggers but these are slightly different and so will be described separately.

7.1 The Cintsys Debugger

When the Cintsys 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 ^e    Dyadic operators
!e                         Subscription
< >                        Shift left/right one place
$b $c $d $e $f $o $s $u $x Set the print style
SGn SPn SRn SVn            Store in variable
=          Print current value
TRn        Trace the next n instructions
Tn         Print n consecutive locations
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/parent/first/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.

7.2 The Cintpos Debugger

Under Cintpos, the interactive debugger can be entered by connecting the console to task 2 (using `@s02`). This allows debugging to take place while other tasks are running. Alternatively, the debugger is automatically entered in standalone mode when a fault is encountered or by an explicit call of `abort`. Most of its facilities are the same as for the Cintsys version, however a few more operations are available to access Cintpos features. The `?` command prints the following.

```
?          Print list of debug commands
Gn Pn Rn Vn Wn An          Variables
G  P  R  V  W  A           Pointers
123 #o377 #FF03 'c         Constants
*e /e %e +e -e |e &e ^e    Dyadic operators
!e                         Subscription
< >                        Shift left/right one place
$b $c $d $f $o $s $u $x    Set the print style
SGn SPn SRn SVn SWn SAn    Store current value
Sn                         Select task n
S.                         Select current task
H          Hold/Release selected task
K          Disable/Enable clock interrupts
=          Print current value
TRn        Trace the next n instructions
Tn         Print n consecutive locations
I          Print current instruction
N          Print next instruction
D          Dump Cintcode memory to DUMP.mem
Q          Quit -- leave the cintpos system
M          Set/Reset memory watch address
B 0Bn eBn  List, Unset or Set breakpoints
X  (G4B9C) Set breakpoint 9 at start of clihook
Z  (P1B9C) Set breakpoint 9 at return of current function
C          Continue normal execution
\          Single step execute one Cintcode instruction
. ; [ ]    Move to current/parent/first/next coroutine
,          Move down one stack frame
a1#
```

The main additions as `Sn` to select a task, `S.` to select the current task and `H` to hold or unhold the currently selected task. Since interrupts (particularly from the clock device) interfere with single stepping of Cintcode instructions, the `K` command is provided to turn clock interrupts on and off. The address of the task control block of the currently selected task is given by `W`. Thus the first locations of the control block can be printed by the command `Wt10`.

The debugger prompt contains a letter indicating whether the next instruction is to be executed in user mode (`a`), in kernel mode (`k`) or within the interrupt service routine (`i`). It also contains a number indicating which user task was running.

7.3 Debugging Techniques

This section explores techniques that can be used to find and eliminate errors in programs. To ensure this process is realistic a program called `com/rast2wav.b` of about 1000 lines has been chosen as a case study. This program contains various pairs of lines one correct and the other containing a a bug. Normally the line with the bug is commented out. By changing which line is commented, it is possible to see the effect of a bug and demonstrate how it can be found.

The program is intended to create a `.wav` sound file based on raster data created by the rastering version of the BCPL system called `rastsys` with the aid of the command `raster`. Raster data in the file `RASTER` can be created by the following sequence of commands.

```
rastsys
c b testrast          -- Compile testraster.b.
raster                -- Cause the next command to
                      -- generate raster data.
testraster            -- Actually generate the data.
```

This creates the raster data file `RASTER` representing the accesses of memory locations during the execution of the program `testraster.b` whose source is:

```
GET "libhdr"

LET start() = VALOF
{ FOR p = 1 TO 250000 DO IF !p LOOP
  RESULTIS 0
}
```

As can be seen this is a simple test program that that reads every Cintcode memory word from 0 to 250000. Under 32-bit BCPL these correspond to byte addresses in the range 0 to 1000000. The resulting file `RASTER` starts as follows:

```
F1750051 M1000000   K1000 S8
W0B71W3073B1W858B1W23B3W562B1W2B1W396B1W43B4W1B1
W213B1W5B1W2135B2W67B3W7B6W1B1N
W70B72W7325B2W14B1N
W142B71W7254B2W14B1N
W213B72W7182B2W14B1N
W284B72W7111B2W14B1N
W356B72W7039B2W14B1N
```

The first line specifies the rastering parameters. `F1750051` states that the program executed 1750051 Cintcode instructions. `M1000000` specifies that the highest byte address referenced was 1000000. `K1000` indicates that 1000 Cintcode instructions were executed

per raster line and S8 says that one unit in the raster lines correspond to 8 address bytes. What then follows are raster lines with each indentifying which addresses have been referenced by the previous 1000 instructions. They use run length encoding with Wn indicating that none the next n units of address space have been referenced and Bn states that all the next n have been referenced. Each raster line is terminated by an N. This file and others, some hand written, are used as test data for rast2wav.

The output generated by the program is a .wav file and so it is necessary to fully understand the format of such a file. The structure is quite simple with a small header block that describes such things as whether mono or stereo is being used and what the sample rate is. This block is followed by 16-bit samples. Luckily it is easy to check that the .wav file structure is correct using the freely available audacity program. This allows the user inspect, edit and play .wav files.

Probably the most important advice on debugging is to spend sufficient time proof reading the source code with great care. This is likely to save time in the long run. It is, of course, essential to thoroughly understand the meaning of every construct in the code. Misunderstanding the meaning of a statement can lead to bugs that are hard to find. Luckily BCPL is simple and is easy to learn. Additionally, there are compile options that help the user to check the meaning of any construct. The precedence of expression operators such as +, -, * and / are fairly intuitive, and can be checked by console sessions such as the following.

```
0.000>
0.000> type t24.b
LET f(x,y,z) = x * y / z
0.012> c b t24 tree
bcpl t24.b to t24 tree

BCPL (3 Sep 2019) 32 bit with the FLT feature
bcpl compiling to file: t24
Parse Tree
LET  t24.b[1]
*-FNDEF  t24.b[1]
! *-NAME: f
! *-COMMA
! ! *-NAME: x
! ! *-COMMA
! !   *-NAME: y
! !   *-NAME: z
! *-DIV
!   *-MUL
!   ! *-NAME: x
!   ! *-NAME: y
!   *-NAME: z
*-Nil
Code size =    36 bytes of 32-bit little ender Cintcode
0.044>
```

This shows that `x*y` is computed before dividing by `z`. Using integer arithmetic the result would often be different if the division was done first. This difference is significant in the meaning of `q := memvupb * (n+1) / (C7-C2+1)` taken from `rast2wav.b`. The `D1` option is also sometimes helpful, as in:

```
00.000> c b t24 d1
bcpl t24.b to t24 d1

BCPL (3 Sep 2019) 32 bit with the FLT feature
bcpl compiling to file: t24
   0:  DATAW #x00000000
   4:  DATAW #x0000DFDF
   8:  DATAW #x2020660B
  12:  DATAW #x20202020
  16:  DATAW #x20202020
// Entry to:   f
  20: L1:
  20:    LP4
  21:    MUL
  22:    LP5
  23:    DIV
  24:    RTN
  25: L2:
  28:  DATAW #x00000000
  32:  DATAW #x00000000
Code size =    36 bytes of 32-bit little ender Cintcode
0.045>
```

Note that the three arguments of `f` are held in positions 3, 4 and 5 relative to the `P` pointer.

The precedence of non arithmetic operators are not so intuitive and, indeed, tend to be different in different languages. A typical BCPL error is the belief that `IF a&7 = b&7 DO` means `IF (a&7)=(b&7) DO`. BCPL uses operators such as `&` and `|` for both Boolean and bit pattern operations and gives them the precedence normally given to Boolean operators.

The transformations performed by the FLT feature and not always understood but can be checked using the `TREE2` compiler option that outputs the parse tree after these transformations have been done by the translation phase. As an example study the following console session.

```
0.000> type t25.b
LET f(x, FLT y, z) BE
  x, y, z +:= 1, FLOAT x + y * 2, FIX y / z
0.012> c b t25 tree2
bcpl t25.b to t25 tree2
```

```
BCPL (3 Sep 2019) 32 bit with the FLT feature
bcpl compiling to file: t25

Parse Tree after calling translate
LET  t25.b[1]
*-RTDEF  t25.b[1]
! *-NAME: f
! *-COMMA
! ! *-NAME: x
! ! *-COMMA
! !   *-FLT
! !   ! *-NAME: y
! !   *-NAME: z
! *-SEQ
!   *-ASSADD  t25.b[2]
!   ! *-NAME: x
!   ! *-NUMBER: 1
!   *-SEQ
!     *-ASSFADD  t25.b[2]
!     ! *-NAME: y
!     ! *-FADD
!     !   *-FLOAT
!     !   ! *-NAME: x
!     !   *-FMUL
!     !     *-NAME: y
!     !     *-FNUM:       2.000000
!     *-ASSADD  t25.b[2]
!       *-NAME: z
!       *-DIV
!         *-FIX
!         ! *-NAME: y
!         *-NAME: z
*-Nil
Code size =    68 bytes of 32-bit little ender Cintcode
0.047>
```

This shows that the so called simultaneneous assignment is, in fact, a sequence of three assignments with the second one promoted to floating point. It shows that `FLOAT` and `FIX` are monadic operators more binding that multiplication and division. It also shows that `FLOAT x+y*2` is transformed to `FLOAT x#+y#*2.0`.

Another useful debugging aid is BCPL's cross referencing facility. A cross reference file `xrast2wav` can be created by the command `make xrast2wav`. This uses the following commands in `Makefile`.

```
xrast2wav:      allcompiled com/rast2wav.b
        cintsys -c c bc rast2wav xref >rawxref
```

```
cintsys -c sortxref rawxref to xrast2wav
rm rawxref
```

A few lines from `xrast2wav` are as follows:

```
fcount G:226 DEF com/rast2wav.b[97] fcount=
fcount G:226 LG com/rast2wav.b[643]
    line_fsecs#:=ftotalsecs#*FLOAT fcount#/fmaxfcount
fcount G:226 LG com/rast2wav.b[705] fcount:=fcount+kval
fcount G:226 LG com/rast2wav.b[706] line_fsecs#:=pos2secs(fcount)
fcount G:226 SG com/rast2wav.b[360] fcount:=0
fcount G:226 SG com/rast2wav.b[440] fcount:=maxfcount
fcount G:226 SG com/rast2wav.b[705] fcount:=fcount+kval
filter G:209 DEF com/rast2wav.b[71] filter=
filter G:209 LG com/rast2wav.b[725] filter(notev,C7)
filter G:209 RT com/rast2wav.b[923] LET filter(v,upb)BE..
```

This shows that `fcount` was declared as global variable 226 on line 97 of rast2wav.b. This variable is used to hold the number of Cintcode instructions obeyed to reach the current raster line. On line 643 it is used to compute the time in seconds as a floating point number corressponding to the time of the current raster line. The actual statement in `rast2wav.b` is:

```
line_fsecs := ftotalsecs * FLOAT fcount / fmaxfcount
```

The last three lines show that the function `filter` was declared to be global 209 and was defined on line 923 and used just once on line 725. We can see that the arguments in the call matches the parameters in its definition. Careful reading of the cross reference listing can sometime find errors in the program. This is worth doing occasionally as the program is being developed.

Another vital tool to assist debugging is the interactive debugger. This is entered automatically when a fault is detected but can also be entered explicitly by the user. At an early stage of debugging the following sequence of commands are useful.

```
0.000> abort

!! ABORT 99: User requested
* x
Breakpoint 9 at start of clihook

0.011> rast2wav

!! BPT 9:        clihook
    A=            0  B=            0   25100:    K4G  1
*
```

This sets breakpoint 9 to be in `clihook` which is in the resident system. It causes a breakpoint just as the `rast2wav` command is about to be entered after it has been loaded into memory and initialised. The instruction `K4G 1` is about to call the function `start` in rast2wav.b. At this point we can inspect the global vector, as in:

```
* g+200t15

G  200:  smoot'mples     wrsample        mark0        mark1          rdn
G  205:  read_'arams  read_'lines      testmem     notecofn       filter
G  210:      addnote     note2str      #G0212#      #G0213#      #G0214#
*
```

This shows that the 13 global functions in `rast2wav.b` have been correctly initionalised. These are useful since they allow the user to set breakpoints at the the the first instruction of any of these functions, as in:

```
* g209=       filter
* b1
* b
1:         filter
9:        clihook
* c
Converting file RASTER to RASTER.wav
sample_rate = 44100
mono 16-bit samples
Total time with the extra second: 11 seconds

maxaddress =    460876
maxfcount  =  58862328
kval=1000 sval=8
c2=0 C3=0 C4=24 C5=36 C6=48 C7=60
Data bytes = 970184
Total number of samples: 440992

debugnote=-1


!! BPT 1:         filter
    A=       145563  B=           60   63920:    LM1
*
```

Another good way to enter the debugger is to insert calls of `abort` in the code usually preceded by a call of `writef` or `sawritef` to output the values of some relevant variables. In the early stages of debugging it is useful to call `abort` after the command arguments have been decoded. For example:

```
0.000> c bc rast2wav
bcpl com/rast2wav.b to cin/rast2wav

BCPL (3 Sep 2019) 32 bit with the FLT feature
bcpl compiling to file: cin/rast2wav
Code size =  5516 bytes of 32-bit little ender Cintcode
0.151> rast2wav
Converting file RASTER to RASTER.wav
sample_rate = 44100
mono 16-bit samples
Total time with the extra second: 11 seconds

maxaddress =    460876
maxfcount  =  58862328
kval=1000 sval=8

!! ABORT 8889: Unknown fault
*
```

This method means we do not need to set the breakpoint in `clihook`. Giving the call of abort an essentially random argument makes it easier to find the call in the source code later.

More to follows.

7.4 Finding a bug during the development of `playmus.b`

This is a case study of how I tracked down a bug in the program `com/playmus.b` in the early stages of developing that program. `playmus` is a program ultimately intended to accompany a soloist playing a musical composition, using realtime data from a microphone to allow it to synchronise with the soloist. This program is currently over 9000 lines long.

This program starts by reading a specification of the complete score in the MUS language which gives all the notes to be played by the accompanist and the soloist including fine detail of how they should be played. These annotations include the information about how the tempo, volume, legatoness and many other aspects of the performance should change as it is played. Details of the MUS language can be found in `musman.pdf` and the `Musprogs` distribution both available from my homepage.

A bug was detected when applying `playmus` to the following MUS file.

```
$get!mushdr;
\score "Opus 1" [
\conductor ( s1        || )
\part      ( 4c4 d e f || )
```

```
]
}
```

Chapter 8

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 8.1 so that the codegenerator (CG), which is inherently machine dependent, is separated from the frontend of the compiler. The front end 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.

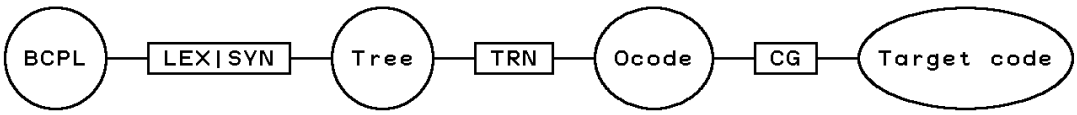


Figure 8.1: The structure of the compiler

8.1 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 if the compiler is not given the `TO` argument it does not invoke the codegenerator but, instead, outputs the OCODE data to the file `ocode` in text form as a sequence of signed decimal numbers. This numerical representation of OCODE can be transformed to a more readable mnemonic form using the `procode` command, described on page 147. 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` would write the following text to the file `ocode`.:

```
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 `procode` command:

```
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 ADD LP 5 ADD LG 74 RTAP 6 RTRN STACK 3
ENDPROC STACK 3 LAB L2 STORE GLOBAL 1 1 L1
```

8.2 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 8.2.

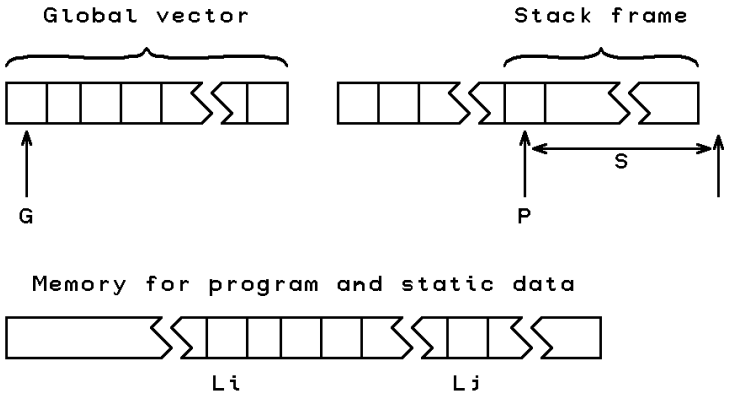


Figure 8.2: The BCPL abstract machine

The OCODE machine has four registers `G`, `P`, `PC` and `A`. `G` points to the global vector and `P` points to the stack frame of the currently executing function. `PC` points to the

next instruction to execute and `A` is a register used hold the results of function calls and in the compilation of `VALOF` expressions. The symbol `S` holds the current size of the stack frame. It is not a register since its value is known at every point in the program by both the frontend of the compiler and the codegenerator. They both know the effect on `S` of every OCODE statement. Program labels are of the form `L`*n* where n is an integer. These point to positions in the program such as the entry points of function, the desinations of jumps and the location of static data. As with `S` labels do not need registers since their values are known.

Static variables, tables and string constants are allocated space embedded in the compiled code. All global, local and static variables are of the same size which is commonly 32 or 64 bits. Some old versions of BCPL have other word length, such as 16 bits for the Intel 6502 or Zilog Z80.

OCODE is normally encoded as a sequence of integers since it is generated by the frontend of the compiler and read by the codegenerator. A more readable form can be created using the `procode` command described on page 147. An OCODE statement consists of a function code or directive followed by operands that are either optionally signed integers, quoted characters or labels such as `L13` or Ł97). 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 the implementation of functions and routines, and for controlling the flow of execution. There are also directives for the allocation of static storage.

8.3 Loading and Storing values

BCPL variables may be local, global or static, and may be accessed in various ways depending on its context. The Ocode 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 L`n	`P!S := L`n`; S := S+1`
`LLP` n	`P!S := @P!`n`; S := S+1`
`LLG` n	`P!S := @G!`n`; S := S+1`
`LLL L`n	`P!S := @L`n`; S := S+1`
`SP` n	`S := S-1; P!`n `:= P!S`
`SG` n	`S := S-1; G!`n `:= P!S`
`SL L`n	`S := S-1; L`n `:= P!S`
`RSTACK` n	`P!`n `:= A; S :=` n`+1`

The RSTACK statement is used in conjunction with the RES statement in the compilation of VALOF expressions. See page 190 for details.

The following tables shows the statements for loading constants.

Statement	Meaning
LF Ln	P!S := 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 n $C_1 \ldots C_n$	P!S := "$C_1 \ldots C_n$"; S := S+1

LF L*n* loads the entry address of a non global function onto the stack. LN *n* loads the signed integer constant n onto the stack. If LN is loading a floating point value n will be a 32 or 64 bit integer has a bit pattern corresponding to a single or double length floating point number. The statements TRUE and FALSE are present to improve portability between machines that may use ones complement representation for integers. On such machines TRUE is not equivalent to LN -1. QUERY loads an undefined value onto the stack, and the LSTR statement allocates a string in static memory and loads a pointer to it onto the stack.

Indirect assignments and assignments to elements of word and byte arrays normally 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  ADD    STIND
LN 65  LG 200  LN 3  PUTBYTE
```

8.4 Field Selection Operators

Accessing and updating fields as required by the OF operator are implemented using the OCODE operators SELLD and SELST.

SELLD takes two argments *len* and *sh*. It effect is equivalent to

P!(S-1) := !(P!(S-1)) >> *sh* & *mask*

where *mask* is a bit pattern containing *len* right justified ones. If em len is zero no masking is done.

SELST takes three argments *op*, *len* and *sh*. If *op* is zero, its effect is equivalent to

```
SLCT len:sh:0 OF (P!(S-1)) := P!(S-2); S := S-2
```

but if *op* is non zero it represents and assignment operator (*assop*) and the statement is equivalent to:

```
SLCT len:sh:0 OF (P!(S-1)) assop:= P!(S-2); S := S-2
```

The mapping between *op* and *assop* is given by the following table.

op	*assop*	*op*	*assop*	*op*	*assop*
0	none	1	!	2	#*
3	#/	4	#MOD	5	#+
6	#-	7	*	8	/
9	MOD	10	+	11	-
12	<<	13	>>	14	&
15	|	16	EQV	17	XOR

The floating-point assignment operators are only allowed when the specified field is a full word, typically with *len* and *sh* both zero. The SELST operator with *len* and *sh* both zero is used in the compilation *assop*:= assignments where the left hand side is a simple variable or a subscripted expression. For instance, the assigment v!3+:=1 might generate the following OCODE.

```
LN 1
LN 3 LG 200 ADD
SELST ADD 0 0
```

8.5 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)
FABS	P!(S-1) := FABS P!(S-1)
FLOAT	P!(S-1) := FLOAT P!(S-1)
FIX	P!(S-1) := FIX P!(S-1)
NEG	P!(S-1) := - P!(S-1)
FNEG	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	S := S-1; P!(S-1) := P!(S-1) % P!S
MUL	S := S-1; P!(S-1) := P!(S-1) * P!S
FMUL	S := S-1; P!(S-1) := P!(S-1) #* P!S
DIV	S := S-1; P!(S-1) := P!(S-1) / P!S
FDIV	S := S-1; P!(S-1) := P!(S-1) #/ P!S
MOD	S := S-1; P!(S-1) := P!(S-1) MOD P!S
ADD	S := S-1; P!(S-1) := P!(S-1) + P!S
FADD	S := S-1; P!(S-1) := P!(S-1) #+ P!S
SUB	S := S-1; P!(S-1) := P!(S-1) - P!S
FSUB	S := S-1; P!(S-1) := P!(S-1) #- P!S
EQ	S := S-1; P!(S-1) := P!(S-1) = P!S
FEQ	S := S-1; P!(S-1) := P!(S-1) #= P!S
NE	S := S-1; P!(S-1) := P!(S-1) ~= P!S
FNE	S := S-1; P!(S-1) := P!(S-1) #~= P!S
LS	S := S-1; P!(S-1) := P!(S-1) < P!S
FLS	S := S-1; P!(S-1) := P!(S-1) #< P!S
GR	S := S-1; P!(S-1) := P!(S-1) > P!S
FGR	S := S-1; P!(S-1) := P!(S-1) #> P!S
LE	S := S-1; P!(S-1) := P!(S-1) <= P!S
FLE	S := S-1; P!(S-1) := P!(S-1) #<= P!S
GE	S := S-1; P!(S-1) := P!(S-1) >= P!S
FGE	S := S-1; P!(S-1) := P!(S-1) #>= P!S
LSHIFT	S := S-1; P!(S-1) := P!(S-1) << P!S
RSHIFT	S := S-1; P!(S-1) := P!(S-1) >> P!S
LOGAND	S := S-1; P!(S-1) := P!(S-1) & P!S
LOGOR	S := S-1; P!(S-1) := P!(S-1) | P!S
EQV	S := S-1; P!(S-1) := P!(S-1) EQV P!S
XOR	S := S-1; P!(S-1) := P!(S-1) XOR P!S

Vector subscription (E_1`!`E_2 is implemented using `PLUS` and `RV`. The value of `x MOD y` is either zero or it has the same sign as `x` and its magnitude is less than `ABS y`. Shifts by a negative amounts yield undefined results, but on some versions of BCPL the shift direction is reversed. Shifts greater than the word length yield zero.

8.6 Functions and Routines

The design of the OCODE statements for the implementation of function and routine calls 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 worthwhile, and the distribution of work between the code for a call and the code at the entry point is up to the implementer. In a typical program there are about five calls for each function or routine 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 function or routine definition generates an OCODE sequence of the following form:

```
ENTRY Li n C1 ... Cn
SAVE s
body of function or routine
ENDPROC
```

`L`i is the label allocated for the entry point. As a debugging aid, the length of the function or routine name is given by n and its characters by the $\texttt{C}_1 \ldots \texttt{C}_n$. The `SAVE` statement specifies the initial setting of `S`, which is just the save space size (typically 3) plus the number of formal parameters. For functions defined using pattern matching the number of formal parameters is determined by the patterns in the match list. The state of the stack just after entry is shown in figure 8.3.

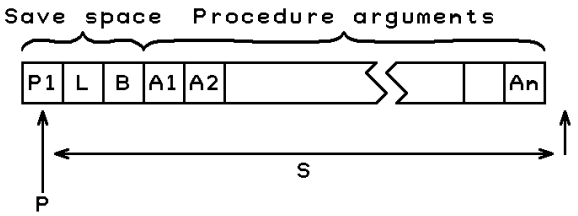


Figure 8.3: The stack frame on function or routine entry

The save space is used to hold `P1` the previous value of `P`, `L` the return address and `B` the function entry address. Although in some versions of BCPL the save space is reduced to two word by omitting the function entry address. This saves a little stack space but makes certain debugging aids impossible. Thus, the first argument of a function is normally at position 3 relative to the `P` pointer.

The end of the body is marked by an `ENDPROC` statement which is non executable but allows the code generator to keep track of nested function definitions.

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 arguments of a call in the locations they will occupy when the function or routine 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 to make a subroutine jump to E. The state is then as shown in figure 8.4. The subroutine jump is made using `FNAP` k or `RTAP` k, depending on whether a function or routine call is being compiled. Notice that k is the distance between the old and new stack frames.

The return from a routine is performed by `RTRN` which restores the previous value of `P` and resumes execution at 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.

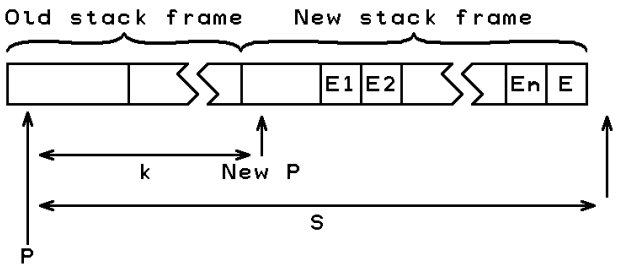


Figure 8.4: The moment of calling `E(E1,E2,...En)`

8.7 Control

The `PC` register holds the address of the next instructions to be executed. For most OCODE statements it is incremented by the size of the instruction, but for control statements `PC` is set specifically as needed by, for instance, conditional jumps or `SWITCHON` commands. The OCODE statements concerned with function and routine calls have already been described above. The remaining control statements are described here.

`LAB` Ln Ln `:= PC`

This directive sets the value of label Ln to the current position in the compiled code.

`JUMP` Ln `PC :=` Ln

`JT` Ln `s := S-1; IF P!S DO PC :=` Ln

`JF` Ln `S := S-1; UNLESS P!S DO PC :=` Ln

`JUMP` causes an unconditional jump to the instruction labelled by Ln. Both `JT` and `JF` pop the top item from the stack then conditionally jump to label Ln depending on its value.

`GOTO` `S := S-1; PC := P!S`

This is used in the translation of the BCPL GOTO command.

`RES` Ln `S := S-1; A := P!S; PC :=` Ln

This is used in the compilation of `RESULTIS` commands. The result is evaluated and placed on the top of the stack, followed by a `RES` statement which pops the result from the stack and places the it in register `A` before jumping to Ln. This label points to the first instruction after the code for the `VALOF` block where there is an `RSTACK` statement will push `A` onto the top of the stack after setting `S` appropriately for this point in the program.

If the `VALOF` block is the body of a function, the compiled code for its `RESULTIS` commands is optimised using `FNRN` rather than `RES`.

`SWITCHON` n L$dK_1L_1 \ldots K_nL_n$

This 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 (Ld) is the default label. K_1 to K_n are the case constants and L_1 to L_n are the corresponding labels. This is normally compiled as a squence of

tests, a label vector switch or a mechanism involving binary chopping, depending on the number and range of the case constants.

`FINISH` Ln `S := S-1; A := P!S; PC :=` Ln

This statement is the compilation of the BCPL `FINISH` command. It should be converted by the codegenerator into code equivalent to `stop(0)` by the code generator. Users are strongly discourage from using `FINISH`

8.8 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` Ln.

The `SECTION` and `NEEDS` directives in a BCPL program translate into `SECTION` and `NEEDS` statements of the form:

$$\texttt{SECTION}\ n\ C_1 \ldots C_n$$
$$\texttt{NEEDS}\ n\ C_1 \ldots C_n$$

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 functions, routines and labels. The form of the `GLOBAL` statement is as follows:

$$\texttt{GLOBAL}\ n\ K_1 L_1 \ldots K_n L_n$$

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.

8.9 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 extensions 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 functions, routines and labels 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 9

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 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`, and `AW`, sign extend the four byte immediate operand before adding the `MW` register into the senior half of the 64-bit result before resetting the `MW` to zero. In this version static variables are allocated in 64-bit 8 byte aligned locations.

The Cintcode machine has nine registers as shown in figure 9.1.

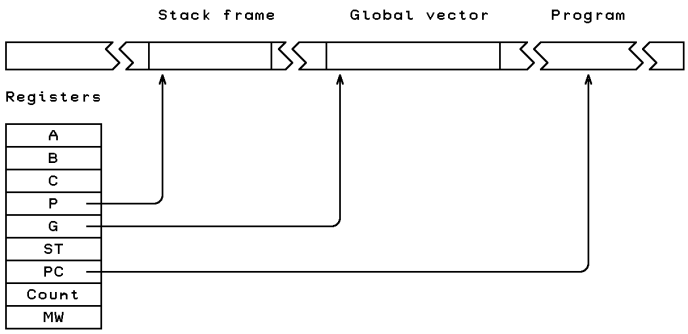


Figure 9.1: The Cintcode machine

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` is used as a status register in the Cintpos version of Cintcode, 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 immediately follow some function bytes. The two instructions used to implement switches have inline data following the function byte. Cintcode modules also contains static data for stings, integers, tables and global initialisation data.

9.1 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 9.1. This data was produced using the tallying feature controlled by the `stats` command, described on page 153.

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 9.1 that accessing variables and constants requires special care, and that conditional jumps, addition, 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 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.

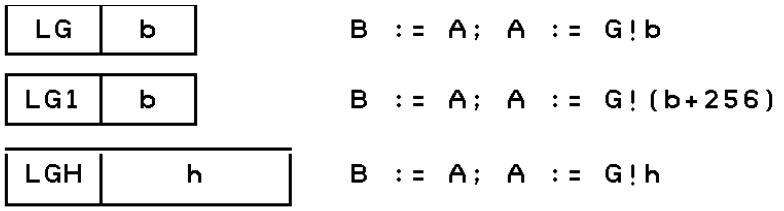
Operation	Executions	Static count
Loading a local variable	3777408	1479
Updating a local variable	1965885	1098
Loading a global variable	5041968	1759
Updating a global variable	796761	363
Using a positive constant	4083433	1603
Using a negative constant	160224	93
Conditional jumps (all)	2013013	488
Conditional 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 9.1: Counts from the BCPL self compilation test

Although not shown here, other statistics, such as the distribution of relative addressing offsets and operand values, influenced the design of Cintcode.

9.1.1 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:



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

9.1.2 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`.

9.1.3 Relative Addressing

A relative addressing mechanism is used in conditional and unconditional jumps and the instructions: `LL`, `LLL`, `SL` and `LF`. All these instructions refer to locations within the code and are optimised for small relative distances. To simplify the 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.

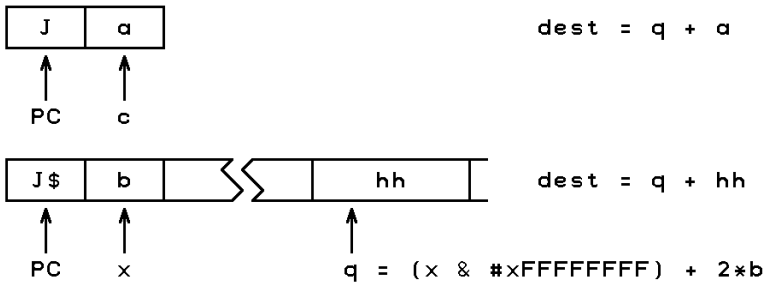


Figure 9.2: The relative addressing mechanism

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 9.2, 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.

9.2 The Cintcode Instruction Set

The resulting selection of function codes is shown in Table 9.2 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.
`MW`	The Cintcode `MW` register used in 64-bit Cintcode.

9.2.1 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 8.3 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

	0	32	64	96	128	160	192	224
0	-	K	LLP	L	LP	SP	AP	A
1	FLTOP	KH	LLPH	LH	LPH	SPH	APH	AH
2	BRK	KW	LLPW	LW	LPW	SPW	APW	AW
3	K3	K3G	K3G1	K3GH	LP3	SP3	AP3	L0P3
4	K4	K4G	K4G1	K4GH	LP4	SP4	AP4	L0P4
5	K5	K5G	K5G1	K5GH	LP5	SP5	AP5	L0P5
6	K6	K6G	K6G1	K6GH	LP6	SP6	AP6	L0P6
7	K7	K7G	K7G1	K7GH	LP7	SP7	AP7	L0P7
8	K8	K8G	K8G1	K8GH	LP8	SP8	AP8	L0P8
9	K9	K9G	K9G1	K9GH	LP9	SP9	AP9	L0P9
10	K10	K10G	K10G1	K10GH	LP10	SP10	AP10	L0P10
11	K11	K11G	K11G1	K11GH	LP11	SP11	AP11	L0P11
12	LF	S0G	S0G1	S0GH	LP12	SP12	AP12	L0P12
13	LF$	L0G	L0G1	L0GH	LP13	SP13	XPBYT	S
14	LM	L1G	L1G1	L1GH	LP14	SP14	LMH	SH
15	LM1	L2G	L2G1	L2GH	LP15	SP15	BTC	MDIV
16	L0	LG	LG1	LGH	LP16	SP16	NOP	CHGC0
17	L1	SG	SG1	SGH	SYS	S1	A1	NEG
18	L2	LLG	LLG1	LLGH	SWB	S2	A2	NOT
19	L3	AG	AG1	AGH	SWL	S3	A3	L1P3
20	L4	MUL	ADD	RV	ST	S4	A4	L1P4
21	L5	DIV	SUB	RV1	ST1	XCH	A5	L1P5
22	L6	MOD	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	SELLD
31	JEQ0$	JNE0$	JLS0$	JGR0$	JLE0$	JGE0$	MW	SELST

Table 9.2: The Cintcode function codes

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 131.

9.2.2 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
`LM` b		`B := A; A :=` $-b$
`LH` h		`B := A; A :=` h
`LMH` h		`B := A; A :=` $-h$
`LW` w		`B := A; A :=` w
`MW` w		`MW :=` 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. The `MW` instruction is only used in 64-bit Cintcode. See page 193 for more details.

`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 pointers to local, global and static variables.

9.2.3 Indirect Load

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

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

9.2.4 Expression Operators

NEG	A := -A
NOT	A := ~A

These instructions implement the three monadic expression operators.

MUL	A := B * A
DIV	A := B / A
MOD	A := B MOD 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 XOR A

These instructions provide for all the normal arithmetic and bit pattern dyadic operators. The instructions DIV and MOD 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 204. Addition is the most frequently used arithmetic operation and so there are various special instructions improve its efficiency.

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

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

APn	$3 \leq n \leq 12$	A := A + P!n
AP b		A := A + P!b
APH h		A := A + P!h
APW w		A := A + P!w
AG b		A := A + G!b
AG1 b		A := A + G!(b+256)
AGH h		A := A + G!h

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.

9.2.5 Simple Assignment

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

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

9.2.6 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.

9.2.7 Function and Routine Calls

At the moment a function or routine is called the state of the stack is as shown in figure 9.3. At the entry point of a function or routine the first argument, if any, will be in register A and in memory P!3.

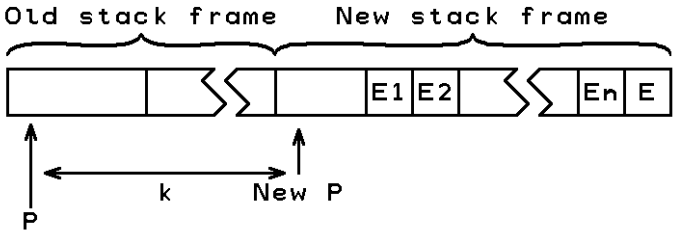


Figure 9.3: The moment of calling E(E1,E2,...En)

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

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

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

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

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

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

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

RTN

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

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

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

9.2.8 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 9.1.3. 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`

9.2.9 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 optional 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 9.1.3 for details on how resolving half words are interpreted.

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

This instruction is used when the range of case constants is too large for `SWL` to be economical. It performs the jump using a binary chop strategy. The quantities n, *dlab*, K_1 to K_n and L_1 to L_n are 16 bit half words aligned on 16 bit boundaries by the option filler byte. This instruction successively tests `A` with the case constants in the balanced binary tree given in the instruction. The tree is structured in a way similar to that used in heapsort with the children of the node at position i at positions $2i$ and $2i+1$. References to nodes beyond n are treated as null pointers. Within this tree, K_i is greater than all case constants in the tree rooted at position $2i$, and less than those in the tree at $2i+1$. The search starts at position 1 and continues until a matching case constant is found or a null pointer is reached. If `A` is equal to some K_i then `PC` is set using the resolving half word L_i, otherwise it uses the resolving half word *dlab* to jump to the default label. See Section 9.1.3 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 %eax,%eax         ;   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 preceding this loop with two copies of:

```
       cmpw (%esi,%eax,4),%bx ;   compare Ki with A
       je swb3                ;   Jump if match found
       adcl %eax,%eax         ;   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.

9.2.10 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, the interpreter returns with exception code P!4.

If A is -1 it sets 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 4.3.

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 changeco(val,cptr) where val is passed in Cintcode register A and cptr is in P!4. Its effect, which is rather subtle, is shown below. For more information see page 56.

```
G!Gn_currco!0 := P!0   // !currco := !P  -- changeco's old P pointer
PC            := P!1   // PC      := P!1 -- changeco's return address
G!Gn_currco   := P!4   // currco  := cptr
P             := P!4!0 // P       := !cptr
```

BRK

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

9.2.11 Floating-point Instructions

Floating-point operations other than those performed by `SELST` are provided by the `FLTOP` instruction. They are as follows.

```
FLTOP 1 b       A := floating point(A × 10^b)
FLTOP 3         A := FLOAT A
FLTOP 4         A := FIX A
FLTOP 5         A := #ABS A
FLTOP 6         A := A #* B
FLTOP 7         A := A #/ B
FLTOP 8         A := A #+ B
FLTOP 9         A := A #- B
FLTOP 10        A := #+A
FLTOP 11        A := #-A
FLTOP 12        A := A #= B
FLTOP 13        A := A #~= B
FLTOP 14        A := A #< B
FLTOP 15        A := A #> B
FLTOP 16        A := A #<= B
FLTOP 17        A := A #>= B
```

In the above table, b is a signed byte representing a decimal exponent in the range -128 to +127. Floating point numbers with exponents outside this range can be generated using `sys(Sys_flt, fl_mk, x, e)` as described on page 3.3.

9.2.12 Select Instructions

Access to fields and some *op*:= assignment are performed using the following instructions.

```
SELLD len sh          A := SLCT len:sh:0 OF A
SELST 0 len sh        SLCT len:sh:0 OF A := B
SELST op len sh       SLCT len:sh:0 OF A op:= B
```

The mapping between *op* and its corresponding expression operator is given by the table on page 187.

9.2.13 Undefined Instructions

There is now only one undefined instruction and it code is 0. It will cause the interpreter to return with exception code of 1.

9.2.14 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.

9.2.15 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 MOD the program counter is left pointing to the offending instruction. For more details see the description of `sys(Sys_interpret,...)` on page 80.

9.3 Example translation of code fragments

This section contains fragments of BCPL code and their translation into Cintcode. The purpose of these examples is to consolidate the reader's understanding of BCPL and show the simplicity of its translation into Cincode. It also shows the level of optimisation performed by the compiler. It is easy to see how a fragment of code is compiled. For instance, consider the program in `z.b`.

```
GLOBAL { w:200; f }

LET f(x) BE WHILE x<10 DO
{ w := x
  IF x=5 BREAK
  x := x+2
}
```

The parse tree for this program can be printed using the following command.

```
0.000> bcpl z.b tree

32 bit BCPL (30 Oct 2021) with pattern matching, 32 bit target

Parse Tree
GLOBAL  z.b[1]
*-CONSTDEF  z.b[1]
! *-CONSTDEF  z.b[1]
! ! *-Nil
! ! *-NAME: f
! ! *-Nil
! *-NAME: w
! *-NUMBER: 200
*-LET  z.b[3]
  *-RTDEF  z.b[3]
  ! *-NAME: f
  ! *-NAME: x
  ! *-WHILE  z.b[3]
  !   *-LS
  !   ! *-NAME: x
  !   ! *-NUMBER: 10
  !   *-SEQ
  !     *-ASS  z.b[4]
  !     ! *-NAME: w
  !     ! *-NAME: x
  !     *-SEQ
  !       *-IF  z.b[5]
  !       ! *-EQ
  !       ! ! *-NAME: x
  !       ! ! *-NUMBER: 5
  !       ! *-BREAK  z.b[5]
  !       *-ASS  z.b[6]
  !         *-NAME: x
  !         *-ADD
  !           *-NAME: x
  !           *-NUMBER: 2
  *-Nil

OCODE size:    52/400000
```

This shows that the parse tree for the `WHILE` command on line 3 of `z.b` has a first argument representing `x<10` and a second argument representing a sequence of three commands, the first being the assigment to `w`. the second being the `IF`statement and the third being the assignment to `x`.

In addition to outputing the pase tree this command also creates a file `ocode` of the corresponding Ocode of the program. This can be printed using the `procode` commnd.

```
0.000> procode
converting ocode to *
ENTRY L10 1  'f'
SAVE 4
LP 3
LN 10
LS
JF L12
LAB L11
LP 3
SG 200
LN 5
LP 3
EQ
JT L12
LN 2
LP 3
ADD
SP 3
LP 3
LN 10
LS
JT L11
LAB L12
RTRN
RTRN
ENDPROC
STACK 3
STORE
GLOBAL 1
     201   L10
```

The corresponding Cintcode translation can be seen using the compiler's `d1` option.

```
0.001> c b z d1
bcpl z.b to z d1

32 bit BCPL (30 Oct 2021) with pattern matching, 32 bit target
   0:  DATAW #x00000000
   4:  DATAW #x0000DFDF
   8:  DATAW #x2020660B
  12:  DATAW #x20202020
  16:  DATAW #x20202020
// Entry to:   f
  20: L10:
  20:    L10
  21:    JGE  L12
  23: L11:
  23:    LP3
  24:     SG  200
  26:     L5
  27:    JEQ  L12
  29:     L2
  30:    AP3
  31:    SP3
  32:    L10
  33:    JLS  L11
  35: L12:
  35:    RTN
  36:  DATAW #x00000000
  40:  DATAW #x000000C9
  44:  DATAW #x00000014
  48:  DATAW #x000000C9
Code size =    52 bytes of 32-bit little ender Cintcode
0.040>
```

You can see from this output that most of the compiled Cintcode instructions occupy one byte, the only exceptions are the conditional jumps and the `SG` instruction. Note that the `WHILE` loop conditions `x<10` is evaluated before the body is executed for the first time and also at the end of the body. This strategy is used since the code at both places can be compiled more efficiently, and if the result of the initial test can be determined at compile time, a conditional jump is not required.

The actual compiled code was placed in the file `z` which is a text file of hexdecimal words.

```
0.001> type z
000003E8 0000000D
0000000D 0000DFDF 2020660B 20202020 20202020 830DBC1A 1C15C831 A3C31207
7BF55C1A 00000000 000000C9 00000014 000000C9
```

Note that this contains the bytes of the compiled code prefixed by the two words `000003E8 0000000D` saying that the compiled code is a hunk consisting of 13 (0000000D) 32 bit words. The word at location zero has been updated with this size now that it is known.

Most of the BCPL code fragments in this section are take from programs in the directories `cintcode/com` and `cintcode/sysb`. The Cintcode translation of the BCPL compiler was placed in `bcpl.cin` by the command:

```
bcpl com/bcpl.b to junk d1 ver bcpl.cin.
```

9.3.1 Translation of `mk1`

The definition of `mk1` is:

```
AND mk1(x) = VALOF
{ LET p = newvec(0)
  p!0 := x
  RESULTIS p
}
```

Its Cintcode translation is:

```
// Entry to:   mk1
9200: L576:
9200:     L0
9201:   K4G1   67
9203:    SP4
9204:    LP3
9205:  ST0P4
9206:    LP4
9207:    RTN
```

The call `newvec(0)` is compiled as `L0 K4G1 67` because `newvec` is in global 323 (=67+256). The variable p is in stack location P4 so the assignment to p!0 can be performed by `ST0P4`.

9.3.2 Translation of `mk2`

The definition of `mk2` is:

```
AND mk2(x, y) = VALOF
{ LET p = newvec(1)
  p!0, p!1 := x, y
  RESULTIS p
}
```

Its Cintcode translation is:

```
// Entry to:   mk2
9224: L577:
9224:     L1
9225:   K5G1   67
9227:    SP5
9228:    LP3
9229:    XCH
9230:     ST
9231:    LP4
9232:    LP5
9233:    ST1
9234:    LP5
9235:    RTN
```

Here the assignment to `p!0` is compiled by `LP3 XCH ST` since Cintcode does not have the instruction `ST0P5`. Note that even though `p` is in the `A` register just before the instruction `ST1` it must be reloaded after the indirect assignment since the compiler cannot assume that `ST1` will not change the value of `p`.

9.3.3 Translation of `rnamelist`

The definition of `rnamelist` is:

```
AND rnamelist() = VALOF
{ // Read a list of names each possibly prefixed by FLT
  LET a = rname()
  UNLESS token=s_comma RESULTIS a
  lex()
  RESULTIS mk3(s_comma, a, rnamelist())
}
```

Its Cintcode translation is:

```
// Entry to:   rnamelist
11152: L641:
11152:   K3G1   47
11154:    SP3
11155:      L   37
11157:    LG1   18
11159:    JEQ  L678
11161:    LP3
11162:    RTN
11163: L678:
11163:   K4G1   23
11165:   K9G1   46
11167:    SP9
11168:    LP3
11169:    SP8
11170:      L   37
11172:   K4G1   60
11174:    RTN
```

It starts by calling `rname` (G303=47+256) and saving the result in stack location P3 for variable `a`. If `token` is not equal to `s_comma` (=37), it returns from `rnamelist` with `a` as the result. If the token was `s_comma` it calls `lex` (G279) then makes a recursive call of `rnamelist` storing the result in stack location P9, the position of `mk3`'s third argument. The second argument at P8 is given the value `a` (P3), and the first argument `s_comma` (=37) is loaded into register `A`. The call of `mk3` (G316) is made by `K4G1 60` and its result immediately becomes the result of `rnamelist`.

9.3.4 Translation of `trnext`

The definition of `trnext` is:

```
LET trnext(next) BE
{ // Compile code to follow a command
  // next is >0, =0 or =-1

  IF next=0 RETURN // No code to compile.

  IF next>0 DO { out2(s_jump, next); RETURN }

  // next must be =-1
  TEST proccontext=s_fnrn
  THEN { out2(s_ln, 0); out1(s_fnrn) }
  ELSE { out1(s_rtrn) }
}
```

Its Cintcode translation is:

```
// Entry to:   trnext
 664: L37:
 664:   JNE0  L38
 666:    RTN
 667: L38:
 667:    LP3
 668:   JLE0  L39
 670:    SP8
 671:      L  146
 673:   K4G1  148
 675:    RTN
 676: L39:
 676:      L  156
 678:    LG1  135
 680:    JNE  L40
 682:     L0
 683:    SP8
 684:      L  136
 686:   K4G1  148
 688:      L  156
 690:   K4G1  147
 692:    RTN
 693: L40:
 693:      L  157
 695:   K4G1  147
 697:    RTN
```

If the argument `next` is zero it returns from `trnext` immediately. At label L39 we know that the `A` register still holds `next` but since the compiler assumes that there may be other instructions jumping to this label it has to reload `next` using `LP3` before testing whether it is greater than zero. If it is `A` still holds `next` and can be placed in stack location P8, the location of the second argument of `out2`. The first argument `s_jump` (=146) is then loaded into `A` ready for the call `H4G1 148` of `out2` (G404=148+256). The remaining code for this routine is straightforward.

9.3.5 Translation of `tst` in `patcmpltest.b`

The definition of `trt` is:

```
// p -> [101, 102, 103, [201, [301,302,303], 203], 105, 106]

LET tst : [a1, a2, a3, [a41, [a421,a422,a423], a43], a5, a6] BE
{ t(a1,   101)
  t(a2,   102)
  t(a3,   103)
  t(a41,  201)
  t(a421, 301)
  t(a422, 302)
  t(a423, 303)
  t(a43,  203)
  t(a5,   105)
  t(a6,   106)
}
```

As can be seen in the Cintcode translation below pattern variables can be accessed with reasonable efficiency. The table below shows the code sequence used to access each of the pattern variables used in this function.

Variable	Code to load the value	Equivalent to
a1	L0P3	p!0
a2	L1P3	p!1
a3	L2P3	p!2
a41	L3P3 RV	p!3!0
a421	L3P3 RV1 RV	p!3!1!0
a422	L3P3 RV1 RV1	p!3!1!1
a423	L3P3 RV1 RV2	p!3!1!2
a43	L3P3 RV2	p!3!2
a5	L4P3	p!4
a6	LP3 RV5	p!5

If we call the argument of `tst` `p`, we see that the pattern variable `a1` is equivalent to `p!0` so if the argument is updated the location referenced by `a1` will change. Similarly, `a41` depends on both `p` and `p!3`, so if either of these change the location referenced by `a41` may change. This effect means great care is needed when defining functions which update pattern variables during their evaluation. The function `splay` is a prime example of this kind of function and should be studied with care. The source code can be found in `BCPL/bcplprogs/patdemos/splay.b`.

```
// Entry to:   tst
 768: L52:
 768:      L  101
 770:    SP8
 771:   L0P3
 772:    K4G  207
 774:      L  102
 776:    SP8
 777:   L1P3
 778:    K4G  207
 780:      L  103
 782:    SP8
 783:   L2P3
 784:    K4G  207
 786:   L3P3
 787:     RV
 788:      L  201
 790:    SP8
 791:    XCH
 792:    K4G  207
 794:   L3P3
 795:    RV1
 796:     RV
 797:     LH  301
 800:    SP8
 801:    XCH
 802:    K4G  207
 804:   L3P3
 805:    RV1
 806:    RV1
 807:     LH  302
 810:    SP8
 811:    XCH
 812:    K4G  207
 814:   L3P3
 815:    RV1
 816:    RV2
 817:     LH  303
 820:    SP8
 821:    XCH
 822:    K4G  207
 824:   L3P3
 825:    RV2
 826:      L  203
 828:    SP8
```

```
 829:    XCH
 830:    K4G  207
 832:      L  105
 834:    SP8
 835:   L4P3
 836:    K4G  207
 838:    LP3
 839:    RV5
 840:      L  106
 842:    SP8
 843:    XCH
 844:    K4G  207
 846:    RTN
 848:  DATAH L21-$
 850: L54:
 850:     L0
 851:    RTN
```

The `DATAH` statement near the end is a 16 bit relative address resolving word for label L21 which has nothing to do with the compilation of `tst`.

9.3.6 Translation of `coins` and `c` in `patdemos/coins.b`

The following function definitions are taken from the coins program `coins.b`.

```
LET coins
: sum => c(sum,
           TABLE 200, 100, 50, 20, 10, 5, 2, 1)
AND c
: <0          => 0
: 0 | (?,[1]) => 1
: sum, t[d]   => c(sum, t+1) + c(sum-d, t)
```

The translation of `coins` is as follows.

```
// Entry to:   coins
  36: L10:
  36:    LLL  L13
  38:    SP8
  39:    XCH
  40:     LF  L11
  42:     K4
  43:    RTN
  44: L12:
  44:     L0
```

```
 45:    RTN
 48: L13:
 48:  DATAW #x000000C8
 52:  DATAW #x00000064
 56:  DATAW #x00000032
 60:  DATAW #x00000014
 64:  DATAW #x0000000A
 68:  DATAW #x00000005
 72:  DATAW #x00000002
 76:  DATAW #x00000001
```

The only oddity here is the code labelled L12 can never be executed since the match item pattern will always be successful. The translation of the function `c` is as follows and is more interesting.

```
// Entry to:   c        LET c
  96: L11:
  96:   JGE0  L14       : <0
  98:     L0                 => 0
  99:    RTN
 100: L14:

 100:    LP3            : 0 |
 101:   JEQ0  L16
 103:   L0P4                  (?,[1])
 104:     L1
 105:    JNE  L15
 107: L16:
 107:     L1                          => 1
 108:    RTN
 109: L15:              : sum, t[d]
 109:     L1                       => c(sum, t+1) +
 110:    AP4
 111:    SP9
 112:    LP3
 113:     LF  L11
 115:     K5
 116:    SP5

 117:    LP3                         c(sum-d, t)
 118:   L0P4
 119:    SUB
 120:    LP4
 121:   SP10
 122:    XCH
```

```
 123:     LF  L11
 125:     K6

 126:    AP5
 127:    RTN
 128: L19:              Unnecessary code
 128:     L0
 129:    RTN
```

In the second call of `c` it would have been better to place `t` in P10 before evaluating `sum-d`.

9.3.7 Translation of `rotleft` from `patdemos/splay.b`

The definition of `rotleft` is:

```
AND rotleft    // Promote right child      p          p
: n[key, val,                           //   |          |
    np[?,?,?,npl,npr],                  //   n    =>    r
    nx,                                 //  / \        / \
    nr[?,?,nrp,nry[?,?,nryp,?,?],nrz] // x   r      n   z
   ] BE                                 //    / \    / \
                                        //   y   z  x   y
{ LET y = nry
  // The order of the assigments was chosen with great care.

  TEST np              // Test if n has a parent.
  THEN TEST n=npl
       THEN npl := nr  // Update the parent's left branch.
       ELSE npr := nr  // Update the parent's right branch.
  ELSE root := nr      // n has no parent, so r is the new root.
  IF nry DO nryp := n  // If y exists, its parent should be n.

  nrp := np
  nry := n
  np  := nr
  nr  := y
}
```

This function makes a simple rearrangement of the nodes close to a given node `n` in a splay tree. A splay tree of a binary tree of key-value pairs with each node being of the form: `[key val parent left right]`. The fields `parent`, `left` and `right` are pointers to other nodes but may be null.

The function has just one match item which contains a pattern that only contains pattern variable declarations so always matches its argument. The variable names are

chosen to make it easy to tell which variables depend on other pattern variables. For instance `nryp` depends on `nrp`, `nr` and `n`. We therefore know that an assignment to `nryp` must typically be made before updating `nr`.

The pattern and its declared variables are valid even when some of the pointers are null. For instance, the variables `npl` and `npr` should only be accessed when `np` is known to be non null.

The Cintcode translation of `rotleft` starts as follows:

```
// Entry to:   rotleft
 392: L18:
 392:   L4P3               LET y = nry
 393:    RV3
 394:    SP4
```

Notice that the pattern variable `nry` is accessed efficiently by two single byte Cintcode instructions. As will be seen even deeply nested pattern variables are accessed with reasonable efficiency.

```
 395:   L2P3               TEST np
 396:   JEQ0  L45
 398:    RV3               THEN TEST n=npl
 399:    LP3
 400:    JNE  L47
 402:   L4P3                    THEN npl := nr
 403:   L2P3
 404:    ST3
 405:      J  L46
 407: L47:
 407:   L2P3                    ELSE npr := nr
 408:     A4
 409:   L4P3
 410:    XCH
 411:     ST
 412:      J  L46
 414: L45:
 414:   L4P3               ELSE root := nr
 415:     SG  204
 417: L46:
```

Notice that the assignment to `npr` is slightly less efficient than the assignment to `npl`. This is because Cintcode has the instruction `ST3` but not `ST4`.

```
 417:   L4P3         IF nry
 418:    RV3
 419:   JEQ0  L48
```

```
421:   L4P3                DO nryp := n
422:    RV3
423:    LP3
424:    XCH
425:    ST2
426: L48:
```

Even though the pattern variable `nryp` is deeply nested the assignment is still reasonably efficient.

```
426:   L2P3          nrp := np
427:   L4P3
428:    ST2

429:    LP3          nry := nr
430:   L4P3
431:    ST3

432:   L4P3          np := nr
433:    LP3
434:    ST2

435:    LP4          nr := y
436:     L4
437:   STP3

438:    RTN
```

Note that the four assignments above are each implemented by three single byte Cintcode instructions.

Chapter 10

The BCPL Compiler

The previous chapters have given the definition of BCPL, Ocode and Cintcode. This chapter gives a brief outline of design of the BCPL compiler. The reason for this chapter is the it provides a example of how BCPL can be used to implement a reasonably significant program, and it may help to consolidate the readers understanding of the language.

The compiler is quite small and easy to understand partly because BCPL is so simple and its translation into Cintcode needs little optimisation. It is just an ordinary command with its source code in the directory `cintcode/com` which contains the source of all the other standard commands.

If the system has been installed in the standard way, all its files will be in the directory `distribution/BCPL` in the user's home directory. The directory `BCPL` contains various subdirectories. The directory `g` holds header files such as `libhdr.h` which contains declarations of all the standard library functions. It also declares library variables and constants needed by most programs. Other header files such as `sdl.h` and `gl.h` provide optional declarations of less frequently used packages, in this case the SDL graphics library and Open GL. Some files in `g` such as `sdl.b` and `gl.b` contain contain actual definitions of functions needed by these packages. One header file of particular relevence to the BCPL compiler is `bcplfecg.h`. This is used by programs closely related to the compiler such as `com/bcplsyn.b` and `bcplcgcin.b`. Files in directory `g` are normally included in programs using `GET` directives.

The main source of the BCPL compiler is the file `com/bcpl.b` but this essentially just contains `GET` directives to include the three components of the compiler, namely `bcplsyn.b`, `bcpltrn.b` and `bcplcgcin.b`. If run under the BCPL Cintcode system when the current working directory is `~/distribution/BCPL/cintcode`, the compiler can be recompiled using the command: `c bc bcpl`. This places the compiled code in directory `cintcode/cin` which is the normal place for all compiled standard commands. On a Raspberry Pi 5 compiling the compiler takes less a second. The components of the compiler are briefly described in turn.

10.1 Lexical Analyser

The lexical analyser and syntax analyser are combined in the file `bcplsyn.b`. When the syntax analyser requires another lexical token it calls the lexical analyser function `lex()` which updates the variable `token` with a value representing the next token. Sometimes `lex` places additional information in other variables such as `wordnode` `decval`, `fltval`, It also sets `lineno` to hold word containing the packed file and line number of the latest token. It sets `nlpending` to `TRUE` if the latest token is the first token on an input line.

Much of the implementation of `lex` is trivially simple switching on the next character of input, normally held in the variable `ch` to decide what to do. Some characters such as spaces and tabs are just skipped over and several others such as `';'`, `','` and `'@'` repesent tokens directly, while others such as `'<'` may require a single character lookahead to determine whether the token is `<=`, `<<` or just `<`. The lexical analysis of names is more involved since some names, such as `WHILE` or `ABS`, are reserved words. When a name in not a reserved word, `token` is set to `s_name` with `wordnode` pointing to its parse tree node. Multiple occurrences of the same name share the same name node. This allows the equality of pointers to be used an efficient test of whether two names are indeed the same. To implement this, a hash table is used to hold lists of name nodes. When a name is encountered its hash value is determined and only name nodes with the same hash value need to be inspected. To implement this name nodes have a link field holding a pointer to the next node in its hash chain. In the translation phase when name nodes are no longer being created this link field is used for another purpose.

The first word of every node of the parse tree identifies what what it represents. For name nodes this field is given the constant value `s_name`. The use of this hash table mechanism allows the table to be preset with nodes for all the reserved words, placing the appropriate token values in their first words.

The hash table is also used to hold section bracket tags and the tags used by the conditional compilation mechanism. These nodes have their name strings starting with dollar signs to avoid confusion with ordinary variable names and reserved words.

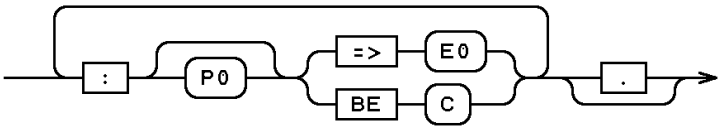
Parse tree nodes could be allocated using `getvec` but it is more convenient and efficient to use an alternative space allocator called `newvec`. This obtains fairly large blocks of memory as needed using `getvec` allocating the typically small parse tree nodes from such blocks. When the parse tree is no longer needed it can be returned efficiently to free store without having to return every parse tree node individually.

Integer constants have their values placed in `decval` and floating point constants use `fltval` to hold the floating point value. String constants do not use the hash table but do use `wordnode` to pass newly created string constant nodes to the syntax analyser.

There is a vector `charv` that holds a circular buffer of recent characters of input. This is used when generating error messages detected during lexical and syntax analysis. The vector `getv` is used in the implementation of `GET` directives. It holds a stack of items containg the current selected input stream, the current file and line number.

10.2 Syntax analyser

The syntax analyser takes a stream of lexical tokens obtained by successive calls of `lex()`. It recognises the syntactic constructs and creates a tree representing the parsed program in a form that is convenient for the next phase of the compilation. The program is easy to understand since it is a direct implementaions of the recursive descent parser specified by the flow graphs in Appendix A. To illustrate the way these flow graphs are implemented in BCPL we will look at the definition of `rdmatchlist` which reads sequences of match items used in `MATCH` expressions and some function definitions. This function is given the argument `s_yields` or `s_be` indicating whether the match list is selecting an expression or a command. This argument is zero if the kind of match list is not yet known. The flow graph for match lists is as follows:

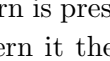


The defintion of `rdmatchlist` is as follows:

```
AND rdmatchlist(sort) = VALOF
{ // Return the parse tree for the match list.
  // Return in result2 {\tt s_yields} or {\tt s_be}
  // indicating which kind of match list was found.
  LET res = rdmatchitem(sort) // Read the first match item
  LET lastitem = res
  sort := result2

  WHILE token=s_colon DO
  { LET item = rdmatchitem(sort)
    h4!lastitem := item
    lastitem := item
  }
  IF token = s_dot DO lex() // The final dot is optional
  result2 := sort
  RESULTIS res
}
```

This function calls `rdmatchitem` to read match items forming them into a list. Match item nodes have five elements the first is the operator `s_matchiteme` or `s_matchitemc` indicating the kind of match item. The `h2` and `h3` fields hold the item's pattern list and expresion or command. The `h4` field holds a link to the next match item, if any, and the final field holds the packed file and line number. The deinition of `rdmatchitem` is as follows:

```
AND rdmatchitem(sort) = VALOF
{ // sort is either s_yields or s_be or zero if not yet known.

  // It returns a pointer to a match item with a null link, ie
  //       [ matchiteme, Plist, E, 0, ln ]
  // or    [ matchitemc, Plist, C, 0, ln ]

  // result2 is set to s_yields or s_be, as appropriate

  LET res = 0
  LET patlist = 0
  LET ln = lineno

  UNLESS token = s_colon DO
    synerr("A match item must start with a ':'")

  lex() // Skip over the colon

  UNLESS token=s_yields | token=s_be DO
  { // There must be a pattern if token is not => or BE
    patlist := rpat(0)
    UNLESS token=s_yields | token=s_be DO
      synerr("token is %s when => or BE expected", opname(token))
  }
  UNLESS sort DO sort := token
  ln := lineno  // The line number of => or BE

  // Check that then defining operator in all match item are the same.
  UNLESS sort=token
    TEST sort=s_yields
    THEN paterr("*nThe defining operator in this match item should be '=>'")
    ELSE paterr("*nThe defining operator in this match item should be 'BE'")

  TEST sort=s_yields
  THEN res := mk5(s_matchiteme, patlist, rnexp(0), 0, ln)
  ELSE res := mk5(s_matchitemc, patlist, rncom(),  0, ln)

  result2 := sort
  RESULTIS res
}
```

Notice that the pattern between `:` and `=>` or `BE` is optional as specified by the flow graph. If a pattern is present it is read by the call of `rpat(0)` correponding to 

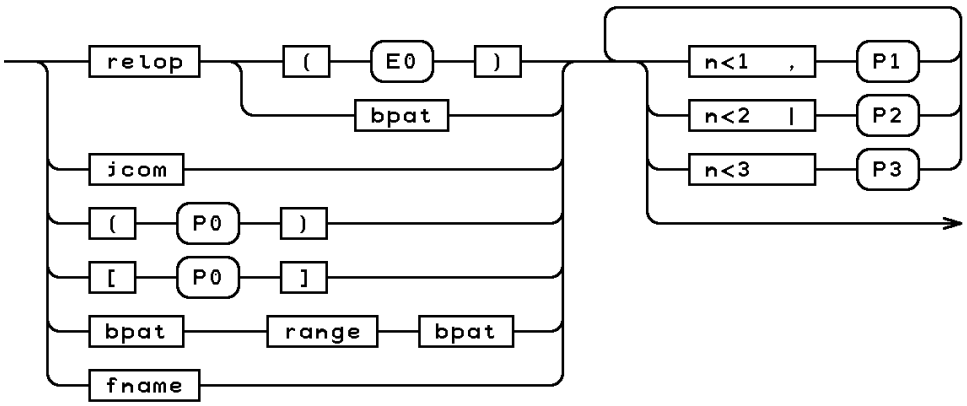
. If after the pattern it then encounters `=>` it calls `rnexp(0)` which calls `lex()` before returning the result of `rexp(0)`, but if it encounters `BE` it calls `rncom()` to read a

command.

As the syntax analyser runs it creates a parse tree of typically small nodes. They are allocated using `newvec` as used by `lex` when allocating name nodes. For convenience most tree nodes are allocated using functions such `mk5` that allocates a node of specified size and sets its elements. Notice that the packed file and line number of the `:` at the start of the current match item was saved in `ln` and placed in the fifth element of the match item node.

Every recursive descent function, such as `rdmatchitem` and `rexp`, follow the same convention that on entry `token` will be the first token of the construct to be parsed, and on exit `token` will be the first token following the construct just read. For many such functions it is useful to have auxiliary functions that call `lex()` before calling the recursive descent function itself. We have already seen two examples `rnpat` and `rnexp`.

The function `rpat` parses a pattern based on the folllowing flow graph.



It definition is as follows:

```
AND rpat(n) = VALOF
{ // Returns zero if token cannot start a pattern.
  // otherwise return it parse tree.

  LET pat = rspat() // Read a simple pattern not involving
                    // comma, vertical bar or juxtapositions.
  UNLESS pat RESULTIS 0

  { // Repeatedly combine pat with other simple patterns
    // separated by commas, vertical bars and juxtapositions
    // depending on the precedence n.

    SWITCHON token INTO
    { DEFAULT:
        // token is not s_comma or s_logor but
        // juxtaposition is possible.
        IF n<3 DO
        { // Juxtaposition is allowable
```

```
          LET b = rpat(3)
          IF b DO
          { pat := mk3(s_patand, pat, b)
            LOOP
          }
        }
        // Juxtaposition was not possible
        RESULTIS pat

      CASE s_comma:
        UNLESS n<1 RESULTIS pat // Comma is not allowed
        lex()
        pat := mk3(s_comma, pat, rpat(1))
        LOOP

      CASE s_logor:
        UNLESS n<2 RESULTIS pat // Vertical bar not allowed
        lex()
        pat := mk3(s_pator, pat, rpat(2))
        LOOP
    }
  } REPEAT
}
```

This function starts by calling `rspat` to read a simple pattern not involving commas, vertical bars or juxtapositions. This corresponds to the left side of the flowgraph. It then enters a loop that combines this pattern with other simple patterns forming `s_comma`, `s_pator` or `s_patand` nodes, as appropriate provided the precedence value `n` is suitable.

The function that reads a simple pattern is straightforward and is as follows:

```
AND rspat() = VALOF
{ // Attempt to read a simple basic, ie one that does not
  // include comma. vertical bar or juxtaposition at the
  // outermost level.

  // It returns zero if token cannot start a pattern.

  LET pat = 0
  LET op = token
  SWITCHON op INTO
  { DEFAULT:
    { pat := rbpat()
      UNLESS pat RESULTIS 0
      IF token=s_range | token=s_frange DO
```

```
  { LET op = token
    LET b = rnbpat()
    UNLESS b DO
      synerr("Problem with the right hand operand of a range")
    RESULTIS mk3(op, pat, b)
  }
  RESULTIS pat
}

// All the tokens in relop
CASE s_eq:    CASE s_feq:
CASE s_ne:    CASE s_fne:
CASE s_le:    CASE s_fle:
CASE s_ge:    CASE s_fge:
CASE s_ls:    CASE s_fls:
CASE s_gr:    CASE s_fgr:
{ LET patrelop = rel2patrel(token)
  lex()
  IF token=s_lparen DO
  { pat := mk2(patrelop, rnexp(0))              // patrelop ( E )
    UNLESS token=s_rparen DO
      synerr("*n   There is a problem with the expression enclosed*n*
             *   in parentheses following a relational operator.")
    lex() // Skip over the close parenthesis
    RESULTIS pat
  }

  { // The operand must be a bpat
    LET b = rbpat()
    UNLESS b DO
      synerr("Bad relational expression in a pattern")
    RESULTIS mk2(relop, b)
  }
}

// All the tokens belonging to jcom.
CASE s_break:
CASE s_loop:
CASE s_endcase:
CASE s_next:
CASE s_exit:
CASE s_return:
  RESULTIS rbcom()

CASE s_lparen:
```

```
      pat := rnpat(0)                // ( P0 )
      UNLESS pat & token=s_rparen DO
        synerr("There is a problem with a pattern enclosed in parentheses")
      lex() // Skip over the close parenthesis
      RESULTIS mk2(s_patseq, pat)

    CASE s_sbra:
      pat := rnpat(0)                // [ P0 ]
      UNLESS pat & token=s_sket DO
        synerr("There is a problem with a pattern enclosed in square brackets")
      lex() // Skip over the close square bracket
      RESULTIS mk2(s_patptr, pat)

    CASE s_flt:
      // Note a name not preceeded by FLT will have been read
      // rbpat() above.
      lex()
      UNLESS token=s_name DO synerr("A name must follow FLT")
      RESULTIS mk2(s_flt, wordnode)
  }
}
```

As can be seen it follows precisely the parsing algorithm specified by the flow graph. It uses `rbpat` whenever it needs to read a basic pattern. The definition of `rbpat` is as follows:

```
AND rbpat() = VALOF
{ // Attempt to read a basic pattern,
  // ie a possibly signed integer or floating point constant.
  // a character constant, TRUE, FALSE, BITSPERBCPLWORD, ?,
  // or a name not preceeded by FLT.

  SWITCHON token INTO
  { DEFAULT:
      RESULTIS 0

    CASE s_number:
    CASE s_fnum:
    CASE s_true:
    CASE s_false:
    CASE s_query:
    CASE s_name:
      RESULTIS rbexp()

    CASE s_add: CASE s_fadd:
```

```
    CASE s_sub: CASE s_fsub:
    CASE s_abs: CASE s_fabs:
    { LET op = token
      lex()
      UNLESS token=s_number | token=s_fnum DO
         synerr("A number must follow a monadic sign operator in a pattern")
      RESULTIS mk2(op=s_add  -> s_pos,   // Use the monadic version
                   op=s_fadd -> s_fpos,  // of + and -.
                   op=s_sub  -> s_neg,
                   op=s_fsub -> s_fneg,
                   op,                   // op is s_abs or s_fabs
                   rbexp())
    }
  }
}
```

Notice that dyadic operators are converted to their monadic versions as necessary, and that the operator `ABS` is treated as a sign since it corresponds to monadic plus or minus depending on the sign of its operand.

This completes the summary of how patterns are parsed. The parsing of expressions, command and definitions are done using the functions `rexp`, `rcom` and `rdef`. Their implementation is as easy to understand as those that parsed patterns and so will not be described here.

There are however some slight subtleties. One is in the treatment of so called simultaneous assignments. In BCPL these are not simultaneous but are executed as a sequence of simple assignments from left to right. For instance the asignment:

```
a, b, c := x, y, z
```

is precisely equivalent to the following:

```
{ a := x; b := y; c := z }
```

This tranformation is done during syntax analysis using the function `cvassign`. This became necessary when the `FLT` feature was added to the language since some of the individual assignments may be given the `FLT` tag. There is a similar transformation required in simultaneous definitions.

One function `plist` is defined in `bcplsyn.b` but is not strictly part of the syntax analyser since it is only used as a debugging aid to output the parse tree is a readable form. But its definition is useful since it can be regarded as a description of the structure of every kind of node in the parse tree. If you compile the following program with the `tree` option it will output the following representation of the parse tree.

LET f : a, [-2, y] =¿ a + y
OCODE size: 40/400000

10.3 The translation phase

10.4 The Codegenerator

Chapter 11

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. The current version of Sial has not yet been updated to deal with the extended BCPL features such as floating point and *op*`:=` assignments.

As as example of the use of Sial, consider the program `com/hello.b` which is as follows:

```
GET "libhdr"

LET start() = VALOF
{ writef("Hello*n")
  RESULTIS 0
}
```

This can be translated into Sial using `bcpl2sial com/hello.b to hello.sial`. The resulting file is:

```
F104
F113 K5 C115 C116 C97 C114 C116
F111 L1
F112 M9001
F32 P3 G94
F11 K0
F77
F107 M9001 K6 C72 C101 C108 C108 C111 C10
F106 K1 G1 L1 G94
F105
```

This can be converted into something slightly more readable using the command: `sial-sasm hello.sial to *` giving: This can be translated into Sial using the `bcpl2sial` command as follows.

```
0.010> sial-sasm hello.sial to *
Converting hello.sial to *
MODSTART

//Entry to: start
ENTRY   K5 C115 C116 C97 C114 C116
LAB     L1
LSTR    M9001
KPG     P3 G94
L       K0
RTN
STRING  M9001 K6 C72 C101 C108 C108 C111 C10
GLOBAL K1
G1 L1
G94
MODEND
Conversion complete
0.000>
```

Alternatively, the Sial can be translated, statement by statement, into the assembly language of a machine such as the Pentium as follows.

```
 0.000> sial-386 hello.sial to hello.s
Converting hello.sial to hello.s
Conversion complete
0.010> type hello.s
# Code generated by sial-386

.text
.align 16
# MODSTART

# 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)
# LSTR    M9001
 leal MA9001,%ebx
 shrl $2,%ebx
# KPG     P3 G94
 movl 376(%esi),%eax
 leal 12(%ebp),%edx
```

```
 call *%eax
# L       K0
 xorl %ebx,%ebx
# RTN
 movl 4(%ebp),%eax
 movl 0(%ebp),%ebp
 jmp *%eax
# STRING  M9001 K6 C72 C101 C108 C108 C111 C10
.data
 .align 4
MA9001:
 .byte 6
 .byte 72
 .byte 101
 .byte 108
 .byte 108
 .byte 111
 .byte 10
 .text
# GLOBAL K1

.globl prog

.globl _prog
prog:
_prog:
 movl 4(%esp),%eax
# G1 L1
 movl $LA1,4(%eax)
# G94
 ret

# MODEND
0.020>
```

Sial was designed as an experiment in the compact representation of algorithms that can be just-in-time compiled easily 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 ten fold speedup over the Cintcode interpretive version. An experienced programmer can normally modify an existing Sial translator to generate reasonable code for a new target in one or two days.

The following sections give a specification of Sial and an outline of how the translator `sial-686` works.

11.1 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 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 the following 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 instrunctions

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 := 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 := ?
xmod		a := a MOD b; c := ?
xsub		a := a - b; c := ?
mul		a := b * a; c := ?
div		a := b / a; c := ?
mod		a := b MOD 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                                 Function or routine return
goto                                PC := a
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 following Sial operators were added in August 2014 to allow native code compilation of the floating point code. All floating point operators may corrupt global 11 (`tempval`).

`float`	`a := FLOAT a; b := ?`
`fix`	`a := FIX a; b := ?`
`fabs`	`a := #ABS a; b := ?`
`fneg`	`a := #- a; b := ?`
`fmul`	`a := b #* a; b := ?`
`fdiv`	`a := b #/ a; b := ?`
`fmod`	`a := b #MOD a; b := ?`
`fadd`	`a := b #+ a; b := ?`
`fsub`	`a := b #- a; b := ?`
`feq`	`a := b #= a; b := ?`
`fne`	`a := b #~= a; b := ?`
`fls`	`a := b #< a; b := ?`
`fgr`	`a := b #> a; b := ?`
`fle`	`a := b #<= a; b := ?`
`fge`	`a := b #>= a; b := ?`
`feq0`	`a := a #= 0; b := ?`
`fne0`	`a := a #~= 0; b := ?`
`fls0`	`a := a #< 0; b := ?`
`fgr0`	`a := a #> 0; b := ?`
`fle0`	`a := a #<= 0; b := ?`
`fge0`	`a := a #>= 0; b := ?`

The floating point conditional jump instructions are as follows.

```
    jfeq       Ln                    Jump to Ln if b #= a; b := ?
    jfne       Ln                    Jump to Ln if b #~= a; b := ?
    jfls       Ln                    Jump to Ln if b #< a; b := ?
    jfgr       Ln                    Jump to Ln if b #> a; b := ?
    jfle       Ln                    Jump to Ln if b #<= a; b := ?
    jfge       Ln                    Jump to Ln if b #>= a; b := ?
    jfeq0      Ln                    Jump to Ln if a #= 0; b := ?
    jfne0      Ln                    Jump to Ln if a #~= 0; b := ?
    jfls0      Ln                    Jump to Ln if a #< 0; b := ?
    jfgr0      Ln                    Jump to Ln if a #> 0; b := ?
    jfle0      Ln                    Jump to Ln if a #<= 0; b := ?
    jfge0      Ln                    Jump to Ln if a #>= 0; b := ?
```

Notice that all floating point instructions currently leave register `b` undefined, but this may be changed later. They may also may corrupt global 11 (`tempval`).

A second example of the use of Sial is the following program (`com/fact.b`):

```
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)
```

It translation in Sial code is as follows:

```
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
```

Using the `sial-sasm` command we obtain the following more readable version:

```
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
JNEO    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
```

This can be translated into assembly language using the program `com/sial-686.b` which is a simple program based on `sial-sasm.b`. This version can now compile the floating point instructions recently added to Sial. It generates the readable version of the Sial source as comments interspersed with the corresponding Pentium assembly code. For the example program given above, it outputs the following assembly language.

```
# Code generated by sial-686

.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-686` 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
%st(0)	The X register used in the compilation of floating point operations
%sp(0)	The S register used when transferring floating point numbers between X and A or B

The chosen BCPL calling sequence is as follows:

```
                                  # Entry address must be in %eax
                                  # The first argument must be in %ebx
leal <stack increment>(%ebp),%edx # Set %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 %ebx(=A)
                    # The new P pointer is in %edx(=C)
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 %ebx(=A)
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-686 is simple. It mainly consists of a large switch within the function scan that has a case for each Sial function code and directive. For example, the case for the function code kpg is approximately 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.

To improve the efficiency of the floating point code, instructions that normally load a value into `A` or `B` are delay until it is known how the value is to be used. If a floating point operation is about to be performed it is better to load the value into `X`. Where possible, `sial-686.b` remembers what value (such as which local or global) is currently held in `A`, `B`, `X` and `S`.

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 within this module. To complete the 686 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`.

Every section must contain the definition of a function to initialise the global vector with the entry points of functions defined in the section. For a section defined in BCPL, the name of the initialisation function is the section name which must have been specified in the BCPL source. A C program such as `initprog.c` must be provided to with a definition of a function called `initsections` that calls the initialisation function of every section of the program. The command `makeinit`, described on page 144, can be used to create the initialiation program. For the program `prog.b` given above, the following command:

```
makeinit prog.b to initprog.c
```

will create the file `initprog.c` which is as follows:

```
// Initialisation file written by makeinit version 2.0

#include "bcpl.h"

WORD stackupb=50000;
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;
}
```

If needed, the runtime stack size and the size of the global vector can be specified by arguments to `makeinit`. The compilation of `initprog` must be linked in with the object code of all the other sections needed by `prog.b` when building its executable. Assuming the executable is placed in `bin/prog`, it can be executed by the `bash` shell command `./bin/prog` or possibly just `prog` if the `PATH` environment variable is suitably set.

11.2 Compaction of Sial

In order to transmit program to a device such as a mobile phone or space probe over a slow connection it is useful to have a compact representation of the code. Sial is both target machine independent and can be compacted with ease. This section gives a brief overview of an experimental compaction technique that seems to performs well.

Since the types of operands and their number depend only on the Sial operator, an Sial stream can be split into several streams of which the main one is the stream of Sial operators. Others are streams holding global variable numbers, local variable offsets, program label numbers, data labels, integer constants, character codes and a some others. These streams can be separately compressed taking advantage of the special properties of each. Some ideas are given below.

Local variable offsets have a very skew distribution and so are susceptible to Huffman (or possibly arithmetic) coding after some preprocessing to deal with large values and the implementation of a mechanism to take advantage of the observation that, if an offset is used once, the same offset is likely to be used again in the near future. This might suggest the use of move-to-front buffering.

Program labels have the property that, in any section, they are each only set once using a `LAB` or `ENTRY` statement. If they are systematically renumbered so that successive label setting statements take successive label numbers, there is no need for these statements to take a label argument. The remaining labels in the stream are typically nearly monotonic the compaction algorithm can take advantage of this.

The operation code stream often contains repeated patterns that are susceptible to the conventional techniques used to compress text, and the same applies to the stream of characters. It might be worth separating out the integers representing the character string lengths from other integers and place them either in a stream of their own or insert them into the stream of characters.

Some preliminary experiments on Sial compression can be found in the directory `bcplprogs/sial` in the standard BCPL distribution.

Chapter 12

The MC Package

This chapter describes the MC package which provides a machine independent way to generate and execute 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. Currently the MC package does not have any floating point operations. This will be rectified in due course.

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.

12.1 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 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

12.2 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.

12.3 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 insufficent 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 preceded 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.

12.4 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.

12.5 The n-queens Demonstration

This section shows how the algorithm to solve the n-queens problem as described in Section 14.3 on page 278 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
 col   = mc_b
 rd    = mc_c
 poss  = mc_d
 p     = mc_e
 count = mc_f
}

LET start() = VALOF
{ // Load the dynamic code generation package
  LET argv = VEC 50
  LET lo, hi, dlevel = 1, 16, #x0000
  LET mcname = "mci386" // Default setting
  LET mcseg = 0
  LET mcb = 0
```

```
  UNLESS rdargs("mc,lo/n,hi/n,-c/s,-m/s,-a/s,-b/s", argv, 50) DO
  { writef("Bad arguments for mcqueens*n")
    RESULTIS 0
  }

  IF argv!0 DO mcname := argv!0           // mc
  IF argv!1 DO lo := !argv!1              // lo/n
  IF argv!2 DO hi := !argv!2              // hi/n
  IF argv!3 DO dlevel := dlevel | #b0001  // -c/s   comments
  IF argv!4 DO dlevel := dlevel | #b0010  // -m/s   mc instructions
  IF argv!5 DO dlevel := dlevel | #b0100  // -a/s   assembler
  IF argv!6 DO dlevel := dlevel | #b1000  // -b/s   binary

  mcseg := globin(loadseg(mcname))

  UNLESS mcseg DO
  { writef("Trouble with MC package: mci386*n")
    GOTO fin
  }

  // Create an MC instance for hi functions with a data space
  // of 10 words and code space of 4000
  mcb := mcInit(hi, 10, 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 = lo TO hi 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 = lo TO hi DO
  { LET k = 0
    writef("Calling mcCall(%n)*n", n)
    k := mcCall(n)
    writef("Number of solutions to %i2-queens is %i9*n", n, k)
  }

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, 3, 0)

  mcRK(mc_mv, ld,    0)
  mcRK(mc_mv, col,   0)
  mcRK(mc_mv, rd,    0)
  mcRK(mc_mv, count, 0)

  cmpltry(1, n, all)        // Compile the outermost call of try

  mcRR(mc_mv, mc_a, count)  // return count
  mcF(mc_rtn)
  mcF(mc_endfn)
}

AND cmpltry(i, n, all) BE
{ LET L = mcNextlab()

  mcComment("*n// Start of code from try(%n, %n, %n)*n", i, n, all)

  mcRR(mc_mv,  poss, ld)         // LET poss = (~(ld | col | rd)) & all
  mcRR(mc_or,  poss, col)
  mcRR(mc_or,  poss, rd)
  mcR (mc_not, poss)
  mcRK(mc_and, poss, all)

  mcRK(mc_cmp, poss, 0)          // IF poss DO
  TEST n-i<=2
  THEN mcJS(mc_jeq, L)           // (use a short jump if near the last row)
  ELSE mcJL(mc_jeq, L)

  TEST i=n
  THEN { // We can place a queen in the final row.
         mcR(mc_inc,  count)     //   count := count+1
       }
  ELSE { // We can place queen(s) in a non 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)  //   // p is a valid queens position
         mcRR(mc_sub,  poss, p)  //   poss := poss - p


         mcR (mc_push, ld)       //   Save current state
         mcR (mc_push, col)
         mcR (mc_push, rd)
         mcR (mc_push, poss)
                                 //   Call try((ld+p)<<1, col+p, (rd+p)>>1)
         mcRR(mc_add,  ld,  p)
         mcRK(mc_lsh,  ld,  1)   //   ld  := (ld+p)<<1
```

```
        mcRR(mc_add,  col, p)   //   col := col+p
        mcRR(mc_add,  rd,  p)
        mcRK(mc_rsh,  rd,  1)   //   rd  := (rd+p)>>1

        cmpltry(i+1, n, all)    //   Compile code for row i+1

        mcR (mc_pop,  poss)     //   Restore the state
        mcR (mc_pop,  rd)
        mcR (mc_pop,  col)
        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 13

Installation

The implementation of BCPL described in this report is freely available via my Home Page [3] to individuals for private use and to academic institutions. If you install the system, please send me an email (to `mr10@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), although these are now little used and are no longer maintained. For Windows XP and Windows 10 there are precompiled `.exe` files such as `wincintsys.exe` and `winrastsys.exe`. These were constructed under Windows XP using Visual Studio an have not been updated since I moved to Windos 10 and so may no longer work. To try them, 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, but this is reasonably easy to do. The simplest installation is for 32 and 64-bit Linux machines which will be covered in detail here. Both the single threaded BCPL Cintcode System called `cintsys` and the Cintcode version of the Tripos Portable Operating System called `cintpos` can be constructed providing a BCPL word length of 32 or 64 bits. BCPL continues to change including the addition of floating points operations, the FLT feature and more recently the MCPL style pattern matching features. I have recently updated the syntax specification of BCPL using the new transition diagrams given in the Appendix of this manual, and there is now a program (`checksyn.b` to test whether BCPL programs conform to this new syntax specification. In due course this program will be modified to attempt to find minimum cost syntactic corrections to erroneous programs. Such corrections are unlikely to produce semantically correct programs but should provide better syntactic error messages.

I repeatedly test the `cintsys` and `cintpos` systems on the machines I currently own, and maintain a log of these tests in the files `CintsysTestLog.txt` in the BCPL distribution and `CintposTestLog.txt` in the Cintpos distribution.

13.1 Linux Installation

This section describes how to install the BCPL Cintcode System on a Linux machine using an Intel 386 or later Intel processors. It can be installed on both 32 and 64 bit architectures, and the size of the BCPL word can be either 32 or 64 bits. To rebuild the BCPL Cintcode system perform the followwing steps.

First create a directory typically named `distribution` in your home directory (`$HOME`) and extract the BCPL distribution files in bcpl.tgz by, typically, typing the commands:

```
ch $HOME/distribution
tar zxvf ../Downloads/bcpl.tgz
```

This creates the directory `BCPL` containing all the files needed to rebuild the system. Next enter the `cintcode` directory by typing the following command.

```
cd BCPL/cintcode
```

You are now ready to rebuild the system, but first you must ensure that the environment variables `BCPLROOT`, `BCPLHDRS`, `BCPLPATH`, `BCPLSCRIPTS` are properly defined. For convenience, there is a `bash` shell script in `os/Linux/setbcplenv`. This script also adds the directory `cintcode/bin` to the `PATH` variable. To set all the variables run the following command.

```
. $(HOME)/distribution/BCPL/cintcode/os/linux/setbcplenv
```

It is probably even better to place this line near the end of `~/.bashrc` so that the environment variables are setup every time a new shell window is created.

You are now ready to rebuild the BCPL system by typing the following commands.

```
cd $(HOME)/distribution/BCPL/cintcode
make clean
make
```

This should recompile all the C and BCPL code required by the BCPL system and leave it waiting for the user to type a BCPL Cmmand Language (`CLI`) command. To test it type the following commands.

```
type com/hello.b
c bc hello
hello
bcpl com/bcpl.b to junk
junk com/bcpl.b to junk
c bc bcpl
bench100
c bc cmpltest
cmpltest
```

What the `make` command did perhaps needs some explanation. Without arguments `make` reads the file `Makefile` from the current directory and performs the first action it finds in this file. This first causes `bin/cintsys` to be created by compiling and linking all the source files needed to build `cintsys`. But before doing this it creates the `#include` file `sysc/INT.h` by compiling and running `sysc/mkint-h.c`. `INT.h` contains several `#define` macros that that allow the C programs to determine important properties of the host machine, such as the C types for signed and unsigned characters. The C source files for `cintsys` are all in the directory `sysc/` and are: `cintsys.c`, `cinterp.c`, `kblib.c`, `cfuncs.c`, `joyfn.c`, `sdlfn.c`, `glfn.c` and `extfn.c`.

Although `cintsys` can now be called, it will only work if precompiled Cintcode compilations of `sysb/boot.b`, `sysb/blib.b`, `sysb/dlib.b`, `sysb/cli.b`, are placed in the directorie `cin/syscin/`. The hand written Cintcode file `syslib` must also be placed there for the (trivial) definitions of the functions `sys`, `changeco` and `muldiv`. Compiled versions of the commands `abort`, `c`, `echo` and `bcpl` are then placed in `cin/`. Finally several scripts such as `b`, `bc` and `bs` are placed in `cintcode/`. Most of these files have different versions depending on whether the host is a big or little ender machine.

Finally, `make` causes the command `c compall` to be executed on the newly created system. This compiles all the resident system components contained in `sysb` and the standard commands in `com/`. The system is then ready for use.

You will notice that directory `BCPL` contains `BCPL/cintcode`, `BCPL/bcplprogs` and `BCPL/natbcpl`. The directory `BCPL/cintcode` contains all the source files of the BCPL Cintcode System, `BCPL/bcplprogs` contains a collection of directories holding demonstration programs, and `BCPL/natbcpl` contains a version of BCPL that compiles into native code (for Intel and ALPHA machines) using a mechanism based on the Sial abstract machine code.

Once the system hase been built it is normally entered using the command `cintsys` which can be called when in any directory. If anything has gone wrong various debugging aids can be turned on using either

```
cintsys -f -v
```

or

```
cintsys -f -vv
```

The output should be studied in conjunction with `sysc/cintsys.c` and `sysb/boot.b`. Hopefully, there will be enough information there to diagnose and correct the problem. It includes, in particular, a trace of all uses of the shell environment variables which are a common source of trouble.

Read the documentation in `cintcode/doc/` and any `README` files you can find. A log of recent changes can be found in `cintcode/doc/changes`. A log of recent tests under different machines and operating systems can be found in `cintcode/CintsysTestsLog.txt`. The current version of this BCPL manual is available from my home page as a `.pdf` file. There is an extensive demonstration script of commands in `cintcode/doc/notes`.

To create the 64-bit version of Cintcode BCPL, type the following.

```
make clean64
make sys64
cintsys64
```

The resulting system is almost identical to the standard 32-bit Cintcode BCPL system but uses a BCPL word length of 64 bits rather that the normal 32.

Other versions of the system that can be created using other make files, for instance:

```
make -f MakefileSDL clean
make -f MakefileSDL
```

This will provide a version with an interface to the SDL graphics library. An interface to the OpenGL graphics library is provided if `MakefileGL` is used. The GL version can be demonstrated by the follow sequence of commands.

```
cd $(HOME)
cd ../bcplprogs/raspi
cintsys
sysinfo
c b engine
engine
c b dragon
dragon
c b bucket
bucket
c b gltst
gltst
```

When you enter `cintsys` you can choose one of two Cintcode interpreters. These can be selected by the commands `fast` and `slow`. The slow interpreter performs more runtime checks and has mode debugging aids than the fast interpreter and is thus somewhat slower. Both interpreters are compilations of the same source file `sysc/cinterp.c` with the differences controlled by conditional compilation statements such as `#ifdef FASTERPyes`.

The `make` command actually creates `rastsys` in addition to `cintsys`. This is a verion of the system that allows the user to generate raster data that can be used to make graphs such as the one in Figure 4.2 on page 151 showing memory references during the compilation of a BCPL compiler. This version is built from the same the source programs in C using conditional compilation statements such as `#ifdef RASTERPyes`.

There is a different but related system called `cintpos` that is closely related to `cintsys`. It is a Cintcode based implementation of the Tripos Portable Operating System originally implemented at Cambridge in the late 1970s. This system allows the user to create tasks which in modern terminology would be called threads since they all use the same address space. Information can be sent from one task to another using the call `qpkt(pkt)`. This appends the packet on the end of a work queue belonging to the destination task. A task can extract the first packet on its work queue using a call of `taskwait()`. If the work queue is empty the task becomes suspended. Every task

has a distinct integer priority and there is a scheduler that ensures the highest priority task that can run is given control. As with the BCPL distribution, Cintpos has its own directory `~/distribution/Cintpos` and all its files are contained in `cintpos.tgz`. Two of the main programs of Cintpos are called `cintpos.c` and `cinterp.c`. These have much in common with `cintsys.c` and `cinterp.c` of the BCPL distribution and the plan is make the C programs in Cintpos identical to the corresponding ones in the BCPL distribution with the the differences controlled by conditional compilation statements such as `#ifdef CINTSYSyes` and `#ifdef CINTPOSyes`. This change is still under development.

13.2 Command Line Arguments

The commands `cintsys`, `cintsys64` 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.
`-q`	Set quiet mode. This stops the resident system from outputting text other than error or debugging messages. It also stops the CLI from outputting prompts or echoing standard input (normally the keyboard).
`-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
`-vv`	As -v, but also include some Cincode level tracing
`-h`	Output some help information.

The rastering versions of the interpreter `rastsys`, `rastsys64` can receive the same arguments.

13.3 Installation on Other Machines

Carry out steps 1 to 4 above. In the directory `BCPL/cintcode/sysasm` 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 `sysasm/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.

13.4 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 os\windows\VC8env.bat
```

and

```
Start in: E:\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 `VC8env.bat`. The BCPL system was then rebuilt by the commands:

```
nmake /f os/windows/MakefileVC clean
nmake /f os/windows/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
```

13.5 Installation using Cygwin

I recommend using the GNU development tools and utilities for Windows that are available from `http://sourceware.cygnus.com/cygwin/`.

Edit the `cintcode/Makefile` to comment out the LINUX version

```
CC = gcc -O9 -DforLINUX -DSOUND -DCALLC -lm
SYSM = ../cintcode/sysasm/linux
```

and enable the CYGWIN32 version

```
CC = gcc -O9 -DforCYGWIN32 -DSOUND -DCALLC -lm
SYSM = ../cintcode/sysasm/CYGWIN32
```

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 cintsys can be run in any directory.

Careful inspection of the Makefile and directories in `cintcode/sysasm` 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.

13.6 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 README file in `sysasm/shwince`. This system provides a scrollable 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.

13.7 The Native Code Version

A BCPL native mode system for 686/Pentium based machines is in directory `BCPL/natbcpl`. It can be re-built and tested by changing to the directory `BCPL/natbcpl` and running `make`. If you have the SDL libraries installed (see `bcpl4raspi.pdf`), you could try

```
make -f MakefileSDL clean
make -f MakefileSDL bucket
./bucket
```

A version (64 bit) for the DEC Alpha is also available but is now out of date and has not been tested recently. To re-build it, it is necessary to comment out the lines for Linux and uncomment the lines for the ALPHA in `Makefile`, before running `make`.

Recently, a version for the ARM processor has been added, particularly for the Raspberry Pi machine. In directory `BCPL/natbcpl` on the Raspberry Pi, try typing

```
make -f MakefileRaspi clean
make -f MakefileRaspi
```

If you have the SDL libraries installed (see `bcpl4raspi.pdf`), you could try

```
make -f MakefileRaspiSDL clean
make -f MakefileRaspiSDL bucket
./bucket
```

It is useful to know how the make commands such as those above work. Here is a brief explanation.

The command `make clean` just deletes all previously built executables together with all files in the directories `obj`, `sial`, `temps` and `tempc` since these can easily be recreated.

The call `make prog` causes the required BCPL programs to be compiled, if necessary, into Pentium assembly language by executing the following CLI commands. This also ensures the C program `tempc/initprog.c` is up to date.

```
bcpl2sial ./prog.b to sial/prog.sial noselst
sial-686 -t  sial/prog.sial to temps/prog.s

bcpl2sial sysb/blib.b to sial/blib.sial noselst
sial-686 -t  sial/blib.sial to temps/blib.s

bcpl2sial ../cintcode/sysb/dlib.b to sial/dlib.sial noselst
sial-686 -t  sial/dlib.sial to temps/dlib.s

makeinit prog.b to tempc/initprog.c
```

If necessary `make prog` also updates the header file `tempc/INT.h` needed by `clib.c` using the following `bash` commands.

```
gcc -o mkint-h sysc/mkint-h.c
./mkint-h >sysc/INT.h
rm -f mkint-h
cp sysc/INT.h tempc
cp sysc/bcpl.h tempc
```

Finally it updates the executable `prog`, if necessary, by compiling and linking all the required C and assembly language programs.

```
gcc -O9 -DforLINUX -o obj/initprog.o -c tempc/initprog.c
gcc -O9 -DforLINUX -o obj/clib.o     -c sysc/clib.c
gcc -O9 -DforLINUX -o obj/kblib.o    -c sysc/kblib.c
gcc -O9 -DforLINUX -o obj/sdlfn.o    -c sysc/sdlfn.c

gcc -O9 -DforLINUX -o obj/prog.o     -c temps/prog.s
gcc -O9 -DforLINUX -o obj/blib.o     -c temps/blib.s
gcc -O9 -DforLINUX -o obj/dlib.o     -c temps/dlib.s
gcc -O9 -DforLINUX -o obj/mlib.o     -c i386/mlib.s

gcc -O9 -DforLINUX -o prog
          obj/initprog.o obj/prog.o
          obj/mlib.o     obj/clib.o  obj/blib.o
          obj/dlib.o     obj/kblib.o obj/sdlfn.o -lm
```

The native code program can now be executed in a `bash` shell using the command `prog` or possibly `./prog`.

Chapter 14

Example Programs

14.1 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))
```

14.2 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 MOD 10 = 0 DO newline()
   writef(" %i3", n)
   count := count + 1
}
```

14.3 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, col, rd) BE TEST col=all

                        THEN count := count + 1

                        ELSE { LET poss = all & ~(ld | col | rd)
                               UNTIL poss=0 DO
                               { LET p = poss & -poss
                                 poss := poss - p
                                 try(ld+p << 1, col+p, rd+p >> 1)
                               }
                             }
```

```
LET start() = VALOF
{ all := 1

  FOR i = 1 TO 16 DO
  { count := 0
    try(0, 0, 0)
    writef("Number of solutions to %i2-queens is %i9*n", i, count)
    all := 2*all + 1
  }

  RESULTIS 0
}
```

14.4 Fridays

The following program prints a table of how often the 13^{th} 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 13^{th} 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) MOD 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 MOD 400 = 0 -> 29,
                    year MOD 100 = 0 -> 28,
                    year MOD 4   = 0 -> 29,
                    28

AND select(n, a0, a1, a2, a3, a4, a5, a6) = n!@a0
```

14.5 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
}
```

14.6 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 N^{th} 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 MOD 8 = 7 DO newline()
                      }
  newline()
}

AND dv(a, m, b, n) = a=1 -> m,
                     a=0 -> m-n,
                     a<b -> dv(a, m, b MOD a, m*(b/a)+n),
                     dv(a MOD 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

This appendix gives the precise syntax of BCPL as it is now, at least in February 2022. It includes the floating point operators, the `FLT` feature and the newly added pattern matching constructs. It also contains some constructs from older versions of BCPL to make compilation of older BCPL programs easier.

The syntax of programming languages is often specified using Backus Naur Form or BNF. Mathematicians like BNF notation because of its simplicity, power and interesting properties, while language designers like it because the rules just confirm their understanding of the language grammar they are designing. For users, understanding a grammar from its BNF specification is harder. There are typically a hundreds of syntactic categories, many with artificial names, and a greater number of rules. Understanding the rules is hard because they mostly depend on each other. There is also sometimes a problem noticing whether a BNF grammar is ambiguous. Indeed it is not possible, in general, to write a program that can determine whether a BNF grammar is ambiguous, and it is also not always easy to write a parser that precisely agrees with the BNF specification.

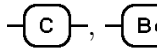
The BCPL syntax is given using the diagrams shown in figures A.1, A.2, A.3, A.4, A.5, A.6 and A.7 for the syntactic categories `Prog`, `D`, `Mlist`, `Pn`, `C`, `Bexp` and `En`. In the diagrams these categories are represented by the rounded boxes:

, 

, 

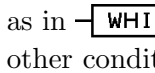
, 

, 

, 

, and 

.

A rectangular boxes are called a test boxes and may contain a terminal symbols as in 

or

, or a label representing a set of terminal symbols or some other condition. These test box labels are specified in the following table.

Label	Possible symbols or condition
`name`	A name not preceded by `FLT`
`fname`	A name possibly preceded by `FLT`
`number`	Integer or floating point constant
`bpat`	Possibly signed integer or floating point constant, character constant, `TRUE`, `FALSE`, `?`, or a name not preceded by `FLT`
`string`	A string constant
`mulop`	`* / MOD #* #/ #MOD`
`posop`	`+ - ABS #+ #- #ABS`
`addop`	`+ - #+ #-`
`relop`	`= ~= < <= > >=` `#= #~= #< #<= #> #>=`
`fcond`	`-> #->`
`range`	`.. #..`
`jcom`	`NEXT EXIT BREAK LOOP ENDCASE RETURN`
`assop`	`:= *:= /:= MOD:= +:= -:=` `#:= #*:= #/:= #MOD:= #+:= #-:=` `<<:= >>:= &:= |:= EQV:= XOR:=`
`iscall`	This is only satisfied if the most recent construct was a function, routine or method call
`isname`	This is only satisfied if the most recent construct was a name not enclosed in parentheses
`nonl`	This is only satisfied if the previous and current tokens are on the same line
`defop`	This is satisfied when reading a `GLOBAL` declaration if the current token is `:` This is also satisfied when reading a `MANIFEST` or `STATIC` declaration if the current token is `=`
`eof`	This is only satisfied if the program file is exhausted

For compatibility with older versions of BCPL some terminal symbols have synonyms as follow.

Symbol	Possible synonyms
`{`	`$(`, possibly tagged
`}`	`$)`, possibly tagged
`DO`	`THEN`
`THEN`	`DO`
`MOD`	`REM`
`NOT`	`~`
`OF`	`::`
`= ~=`	`EQ NE`
`< <=`	`LS LE`
`> >=`	`GR GE`
`<< >>`	`LSHIFT RSHIFT`
`& |`	`LOGAND LOGOR`
`XOR`	`NEQV`

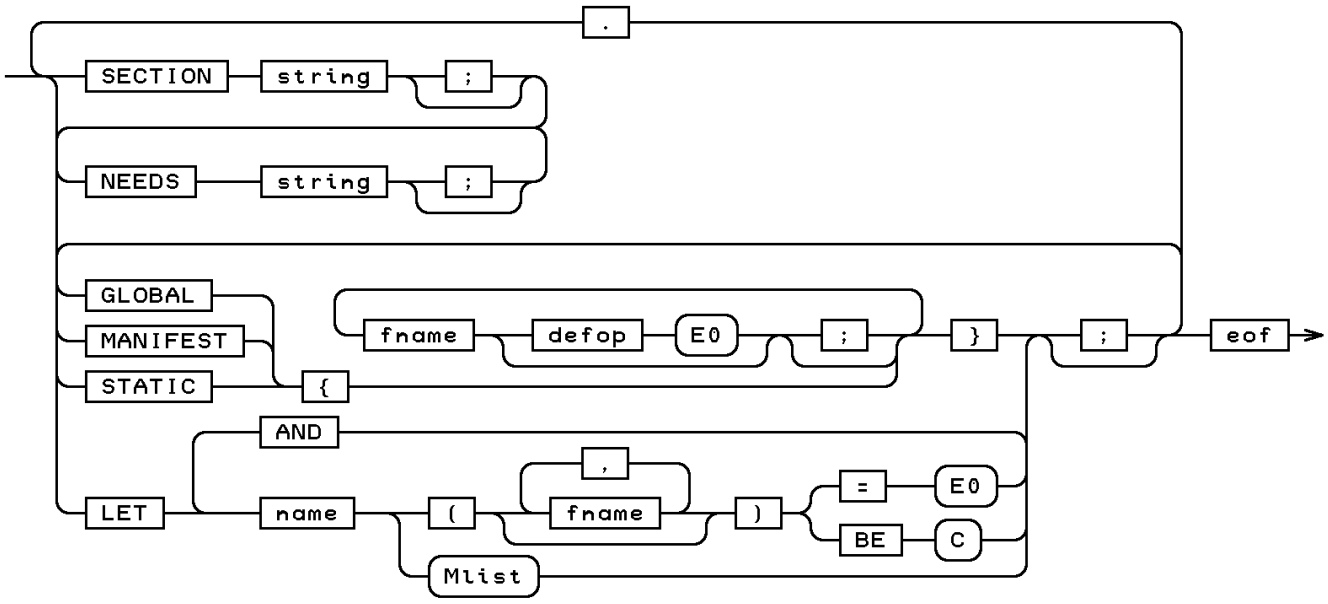


Figure A.1: The definition of Prog

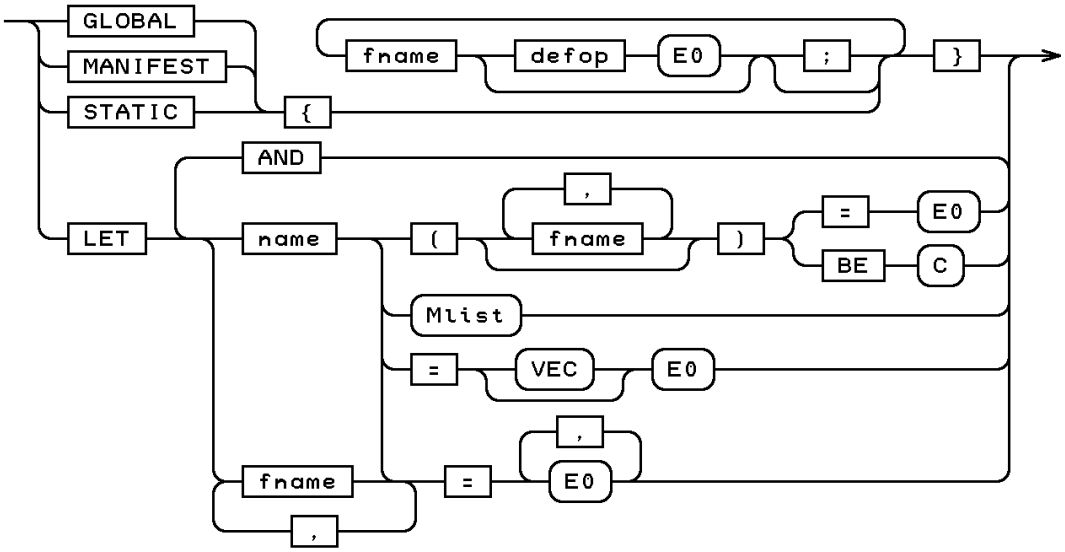


Figure A.2: The definition of D

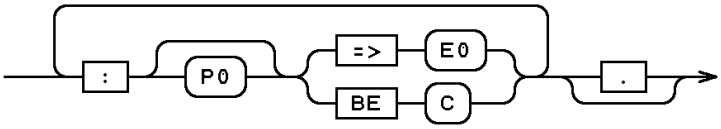


Figure A.3: The definition of Mlist

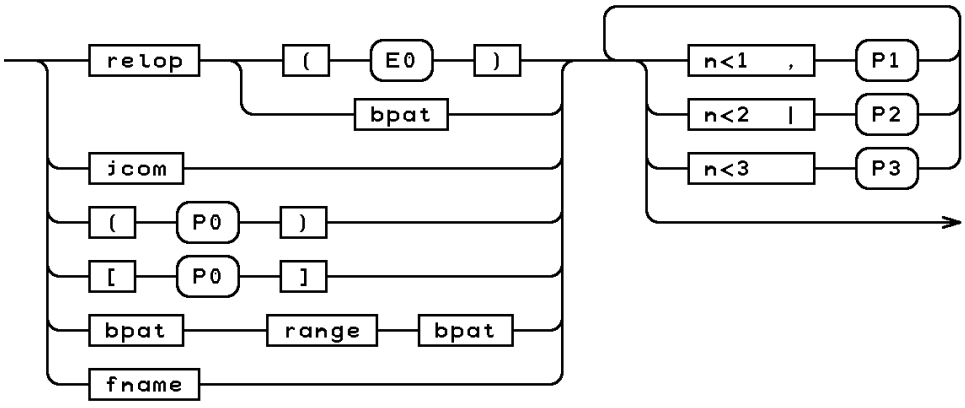


Figure A.4: The definition of Pn

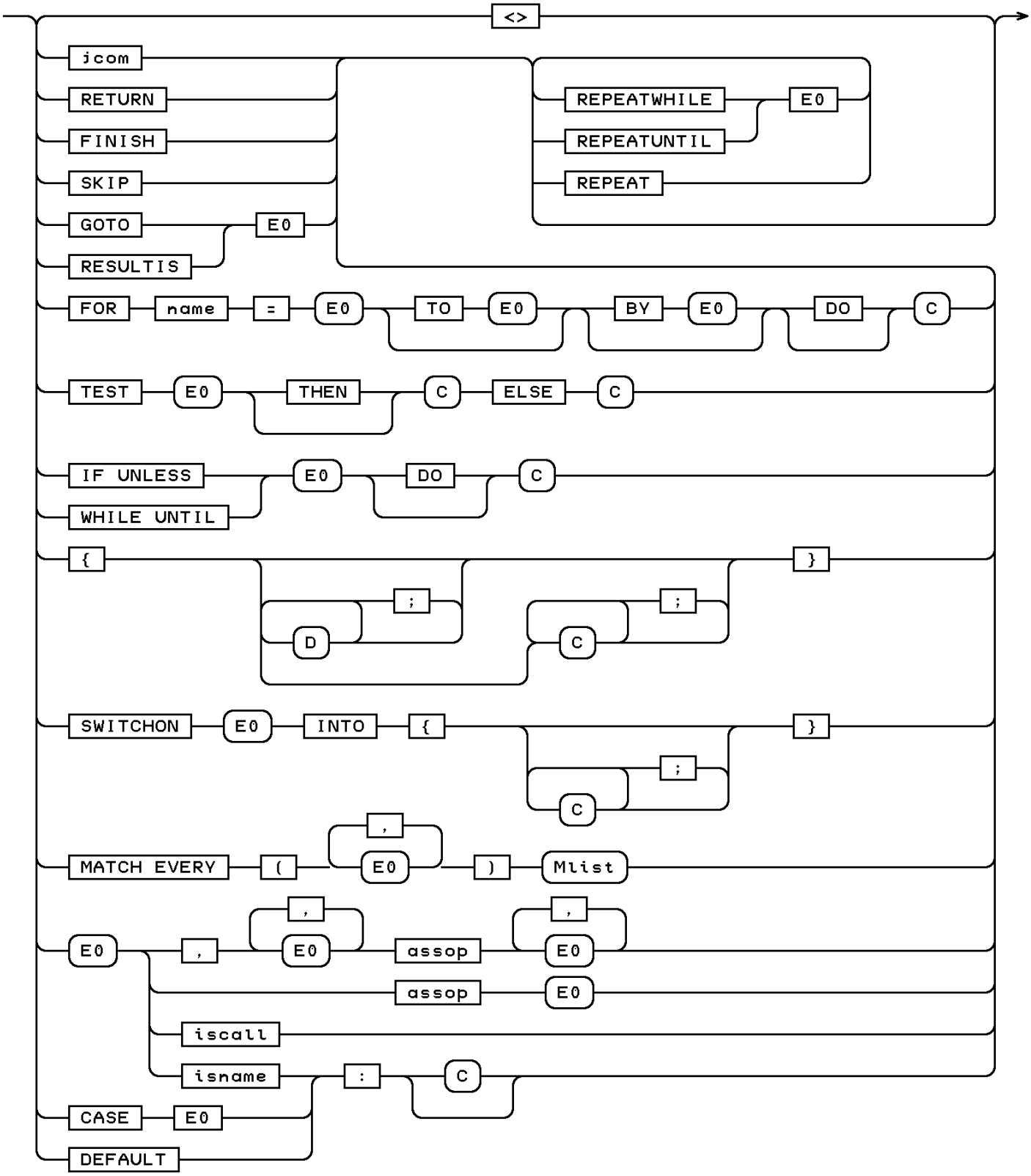


Figure A.5: The definition of C

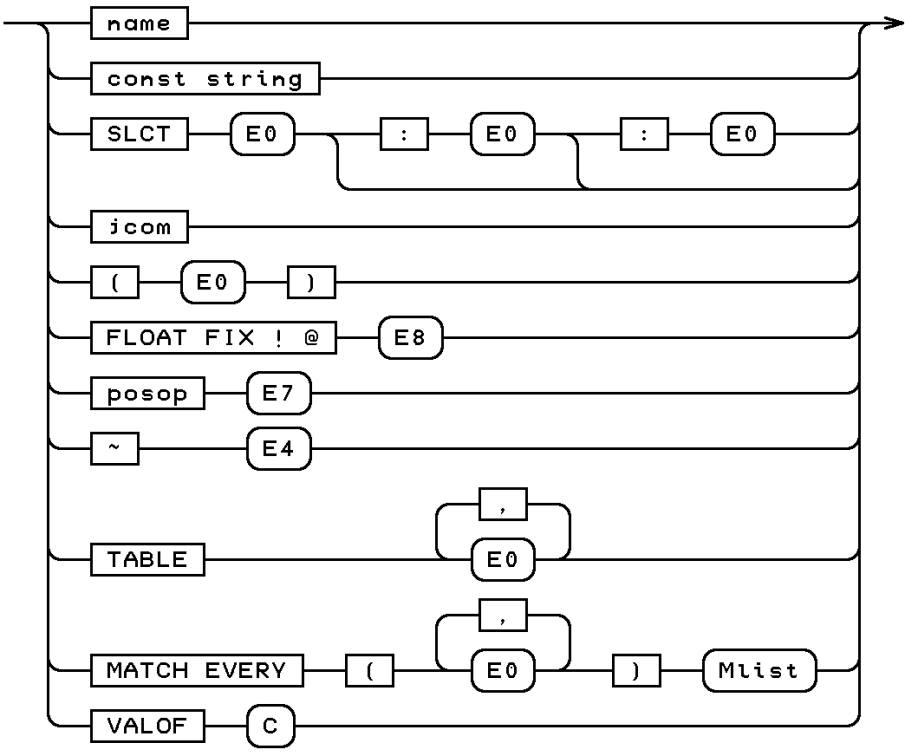


Figure A.6: The definition of Bexp

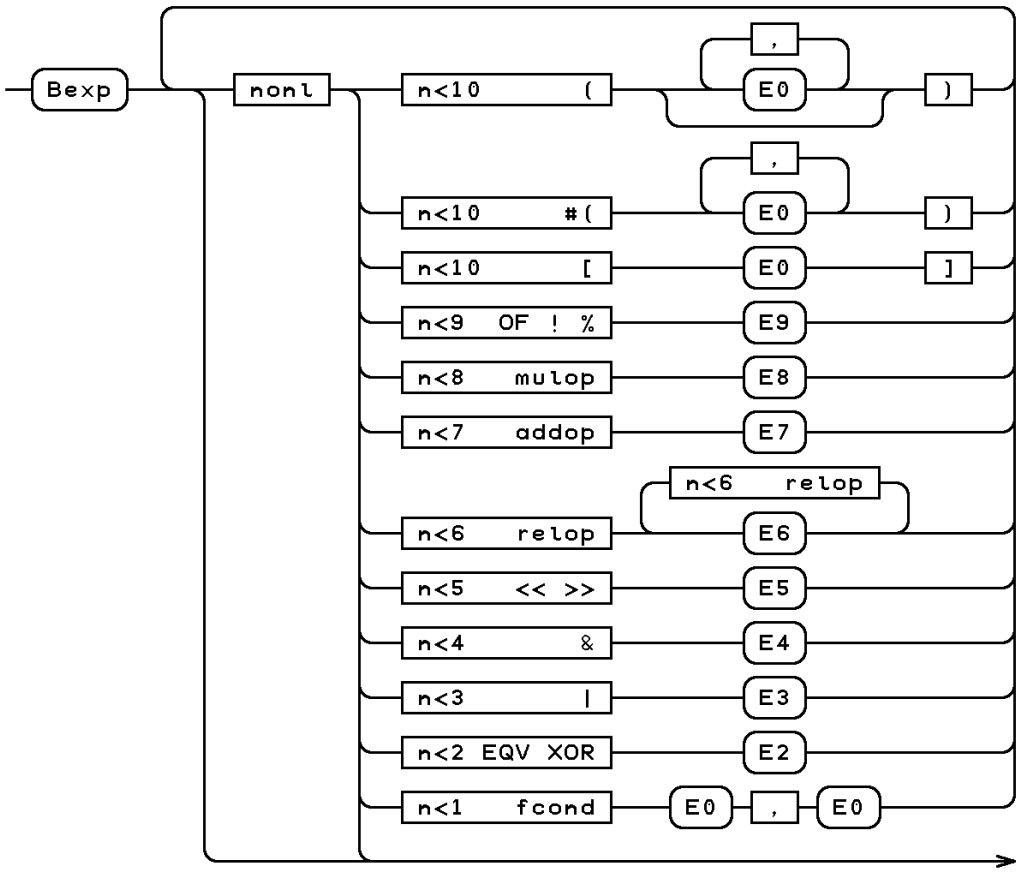
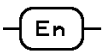


Figure A.7: The definition of 

Appendix B

The Syntax of Lexical Tokens

This appendix is currently under construction

The previous appendix specifies the syntax of BCPL based on a stream of lexical tokens. This appendix uses a similar technique to specify how a stream of characters corresponds to a stream of lexical tokens.

This appendix gives a precise description of how the function `lex` reads the character stream source of a BCPL program creating a stream of lexical tokens.

What follows will be a summary of the diagrams that will be drawn.