Implementing Dependable Process Control Applications using BCPL Coroutines and Cintpos (draft)

Martin Richards*
University of Cambridge
mr@cl.cam.ac.uk

February 2, 2004

Abstract

This paper outlines a method of implementing large and complex process control applications where extreme reliability is required and where the application is expected to have a long life of perhaps between 20 and 50 years. The approach relies on the use of a interpretive abstract machine that supports a multi-tasking operating system. The entire system including the operating system and the compiler for the systems implementation language are simple enough to be maintained as part of the application and so will continue to exist as long as needed. The abstract machine is simple enough to be easily re-implemented on new hardware as the need arises. This paper suggests using Cintcode as the abstract machine supporting an interpretive implementation of the Tripos portable operating system called Cintpos using BCPL as the systems implementation language. This paper pays particular attention to how synchronization primitives can be implemented using BCPL coroutines and the multi-tasking features of the Cintpos portable operating system and why such code will remain reliable throughout the long lifetime of the process control application.

Keywords: process control, dependability, systems programming, coroutines, BCPL, Cintpos

1 Introduction

There are many process control applications that can be expected to have a long life. Examples include the control of chemical plants and nuclear power stations, the control of vehicles and aircraft, the control of manufacturing plants and even distributed booking systems. In all these applications reliability is of great importance since the cost of system failure is likely to be measured in tens or hundreds of thousands of Pounds per minute. The complete system must be constructed with great care so that component failures will not cause the entire system to fail. Often such systems run on multiple computers so that if one fails another can automatically replace it without interrupting the process being controlled.

The proposal is to implement the application using BCPL as the implementation language supported by the simple multi-tasking operating system called Cintpos which is itself supported by an interpretive abstract machine that is easy to re-implement on any new hardware that may become available during the lifetime of the application. The abstract machine, the BCPL to

*Computer Laboratory, William Gates Building, JJ Thomson Ave, Cambridge CB3 0FD, UK

Cintcode compiler and Cintpos are treated as part of the application and are easily maintained by those who maintain the application code.

2 Cintpos

The Tripos Portable Operating System [?] was first implemented in 1978 and ran on a variety of mini-computers of the day, such as the PDP-11, the Data General Nova and machines based on the Motorola 68000 processor. It was implemented in the portable programming language BCPL[?, ?] which still has a freely available implementation available via my home page[1]. Tripos was used at Cambridge and elsewhere for several years providing a framework for operating system design and networking research. When the hardware on which Tripos ran became obsolete much of the original Tripos source code was archived. It was later revived, mainly for historical reasons, when the 32 bit BCPL Cintcode system[?] was developed. The Cintcode virtual machine was enhanced to include interrupts, and the Tripos kernel re-implemented in BCPL rather than the original assembly language. This demonstration version of Tripos was called Cintpos and was made freely available via my home page. More recently, as machines became larger, faster and cheaper, it became clear that this interpretive system could form the basis of dependable process control applications, and so the original "toy" version of Cintpos has undergone substantial development to improve its reliability and to include new features such as TCP/IP connectivity. It is still freely available via the web[?], even in its current experimental form.

The reasons that the BCPL Cintpos system provides a good environment for writing high quality process control applications are as follow:

1. The Cintpos operating system is simple and clearly defined. It has few kernel primitives but these are sufficient to provide the multi-tasking and synchronization facilities needed for real time process control.

2. The whole system including the kernel primitives are implemented in BCPL that is itself supported by a clean simple interpretive virtual machine (Cintcode) that can easily be re-implemented to run on any hardware. Currently this virtual machine is implemented in ANSI C but in years to come it could be re-implemented in whatever language is appropriate at the time. The BCPL to Cintcode compiler is simple and easy to maintain being little more than 5000 lines of source code and able to compile itself in a fraction of a second.

3. BCPL coroutines[2] can be combined with the Cintpos multi-tasking facilities to provide an excellent way to control the scheduling and synchronization needed in process control. This paper shows how easily features such as mutexes, condition variables, semaphores, and signaling and waiting primitives can be implemented in BCPL within the Cintpos system. By this means these very important and sometime subtle primitives will be guaranteed to behave identically throughout the lifetime of the application and not be subject to the inevitable differences arising from changes in standards, re-implementation of the primitives and operating system upgrades that are bound to happen in the next 50 years. It is hard to predict what Windows, Java, Linux or Pthreads will be like that far in the future.

The entire system is implemented in BCPL including the Cintpos Kernel, all the Cintpos resident tasks, the libraries and the application code. The examples given in this paper are in BCPL but these should be easy to understand for anyone familiar with C. The main differences are that BCPL is typeless with every value, variable and vector element being 32 bits long. The BCPL operator `!` is used for indirection like monadic `*` in C. It is also used for vector subscription. The operator `@` will form a pointer to a variable just as monadic `&` does in C. `TEST` is used instead of `IF` in the BCPL version of the `if-then-else` construct found in C. A complete description of BCPL can be found in Richards[**?**].

2.1 Cintpos Kernel Primitives

Cintpos is a simple multi-tasking operating system. Its tasks in modern parlance would be called threads since they all share the same address space. This allows particularly efficient inter-task communication.

Tasks communicate with each other and with devices by sending and receiving packets. These are small vectors (or arrays) of 32-bit fields. The first field is called `pkt_link` and is used to link packets together to form lists, the next is called `pkt_id` and holds the identity of the task or device the packet is to be sent to, and the third field (`pkt_type`) holds the type (or purpose) of the packet. Next there are two fields (`pkt_res1` and `pkt_res2`) to hold the results of a request and finally there are up to six fields (`pkt_arg1` and `pkt_arg6`) which hold the arguments of a request. Since BCPL is typeless all these fields can contain values of any kind, such as integers, strings, pointers or even functions.

The call `qpkt(`*pkt*`)` willto add a packet to a work queue, and `taskwait` to extract a packet from a work queue, possibly causing the task to wait if no packet is available.

A packet is a vector (or array) of typically fewer than 11 32-bit fields. There is a link field (`pkt_link`) used to chain packets together, a field (`pkt_id`) used to identify which task or device the packet is to be sent.

The packet is appended to the end of a work queue at the destination. The identity of the sender replaces the `pkt_id` field so that it can be returned to the sender by just a simple call of `qpkt`.

When a task is ready to process a packet it calls `taskwait()` which de-queues the first packet on the task's work queue, if one is present, otherwise it suspends the task until a packet arrives. Normally packets are thought of as requests that are sent to server tasks or devices. The reply is usually returned using the same packet. While a server task is processing the request, the client is free to continue with other work but more usually it suspends itself in `taskwait` to await the reply.

Tasks have distinct priorities and the scheduler ensures that the runnable task with the highest priority has control. For example, if as the result of an interrupt, a device returns a packet to a client task that is suspended in `taskwait` and is of higher priority than the currently running task then control will be given to this higher priority task. The scheduling algorithm is simple to implement and has predictable properties. Greater fairness can be implemented within the application by dynamically changing priorities, but this is rarely needed.

A task executing the following infinite loop:

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

will suspend itself in `taskwait` until a packet arrives. When this happens it immediately returns it to the sender using `qpkt`. `REPEAT` causes this operation to be repeated indefinitely. This bounce task can be exercised by another task running the following code:

```
{ LET link, id, type, res1, res2 = notinuse, bouncetaskid, ?, ?, ?
  LET pkt = @link
```

```
  writef("Sending a packet 10 million times to task %n*n", bouncetaskid)
  FOR i = 1 TO 10_000_000 DO
  { qpkt(pkt)   // Send the packet
    taskwait()  // Wait for it to return
  }
  writef("Done*n")
}
```

The variable `pkt` points to the first of five consecutive words of memory named `link`, `id`, `type`, `res1` and `res2` which form the packet to be repeatedly sent to the bounce task. A safety check in `qpkt` requires the `link` field to have value `notinuse (=-1)`. The `id` field is set to the identity of the bounce task. The usual packet fields `type`, `res1` and `res2` have been included although they are not used in this demonstration. As can be seen, this program will send the packet to the bounce task and wait for it to return 10 million times. When run on a 1 GHz machine using a Cintpos interpreter with all debugging aids enabled it takes about 108 seconds to complete, indicating that control can pass from one task to another in about half a micro-second. This is faster than the time to execute a single machine instruction on the machines on which Tripos first ran.

With the current BCPL implementation of `qpkt` and `taskwait`, it takes the execution of about 121 Cintcode instructions from the moment `qpkt` is entered to the moment `taskwait` in the destination task returns with the packet. Although this figure depends slightly on the relative priorities of the two tasks it is a fair measure of the cost of transferring control from one task to another. The above figure was obtained using single step execution which is a standard interactive debugging aid available in Cintpos.

A typical task in the Cintpos system is the file handler which accepts requests from clients to open, read, write and close files. These typically cannot be processed instantaneously since the file handler may need to send a packet to the disk device requesting a disk read or write operation. For much of the time the file handler will be suspended in `taskwait` waiting for either a new request from a client or a reply from the disk or possibly a clock device. Tasks such as this are called multi-event tasks since they cannot predict what kind of packet will arrive next. A command language interpreter task (CLI), on the other hand, is a single-event task since it has a single thread of execution and always knows what packet it is expecting to receive next. Whenever it sends a request to another task or device it waits for the reply before proceeding. In such tasks the calls of `qpkt` and `taskwait` are usually invoked in a wrapper function `sendpkt` whose definition is as follows:

```
LET sendpkt(link, id, type, r1, r2, a1, a2, a3, a4, a5, a6) = VALOF
{ UNLESS qpkt(@link)        DO abort(181)
  UNLESS taskwait() = @link DO abort(182)
  result2 := r2
  RESULTIS r1
}
```

In BCPL, function arguments are called by value and are laid out in consecutive memory locations and so behave like an initialized vector. In `sendpkt` the arguments form the packet to be sent eliminating the need to allocate, initialize and later release another vector for the packet. The expression `@link` yields the pointer to the packet and is the appropriate argument for `qpkt`. If `qpkt` fails, possibly because of a bad task (or device) identifier, it will return `FALSE` causing `abort(181)`. Otherwise, `sendpkt` go on to call `taskwait` suspending the task until the packet is returned. In this implementation a safety check is made to ensure that the expected packet is indeed received.

The normal convention is for tasks and devices to place their results in the fields `r1` and `r2` of the packet. These are returned, respectively, as the result of `sendpkt` and in the global variable `result2`.

The code in the sender task of the demonstration given above could have been implemented using `sendpkt` as follows:

```
{ writef("Sending a packet 10 million times to task %n*n", bouncetask)
  FOR i = 1 TO 1_000_000 DO sendpkt(notinuse, bouncetaskid)
  writef("Done*n")
}
```

This is more convenient code but is slightly slower because of the extra safety checking done in `sendpkt`. All the standard library functions that communicate with other tasks or devices use `sendpkt`, including, for example the `delay` function whose definition is as follows.

```
LET delay(ticks) BE sendpkt(notinuse, clkdevid, ?,?,?, ticks)
```

A packet sent to the clock device returns to the sender when the specified number of ticks have taken place. A task can suspend itself for 5 seconds by executing:

```
delay(5*tickspersecond)
```

Notice that the calls of `sendpkt` above have fewer arguments that its definition expects. BCPL allows this, either giving the unspecified arguments undefined values or by throwing away unwanted ones. As an aside, the BCPL formatted output function `writef` is also variadic and is related to the C function `printf` which extends and improves `writef`, but the implementation of `printf` is more complicated since the size of C arguments depend on their types and so cannot be treated as elements of a vector.

2.2 Other Kernel Primitives

Although most of the flavour of Cintpos derives from the `qpkt-taskwait` mechanism, it is worth briefly noting the other kernel primitives. Tasks can be created and deleted using `createtask` and `deletetask`. The priority of a task can be changed using `changepri`. A task can be held by a call of `hold` causing it stop execution until explicitly released by a call of `release`. These are analogous to the deprecated functions `suspend` and `resume` in Java. Tasks each have a field of 32 flag bits that can be set and tested using `setflags` and `testflags`, and memory can be allocated and freed using `getvec` and `freevec` which are somewhat similar to the C functions `malloc` and `free`.

Cintpos is unlike Tripos by not allowing the dynamic creation and deletion of devices, nor does it provide the function `dqpkt` to retrieve previously sent packets.

3 BCPL coroutines

BCPL uses a stack to hold function arguments, local variables and anonymous results, and it uses static variables and an area call the global vector to hold non-local quantities. Each task has its own global vector. It is often convenient to have separate runtime stacks within a task so that different activities can proceed in pseudo-parallelism. The coroutine mechanism provides this facility.

A coroutine is created by a call of the form: `createco(`*fn*`,` *size*`)` where *fn* is the main function of the coroutine and *size* is the size its stack. The result is a pointer to the newly created coroutine stack. Control can be passed to a coroutine by the call: `callco(`*cptr*`,` *arg*`)` where *cptr* is the result of a previous call of `createco` and *arg* is a value to be passed to the coroutine. The first time this happens *fn* will be applied to *arg*. If this function returns a result, *res* say, the coroutine will suspend itself and control passed back to the calling coroutine causing `callco` to return the result *res*. Alternatively, *fn* may explicitly call `cowait(`*res*`)` which has the same effect. The implementation of `createco` actually leaves the newly created coroutine suspended in the `cowait` call of the following infinite loop:

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

A few locations near the base of a coroutine stack hold system information including *fn*, *size*, `c`, and *parent* which is a pointer to the calling coroutine, if any, for use by `cowait`.

Any inactive coroutine can be deleted using `deleteco`. Another important coroutine function is `resumeco`. It takes the same arguments as `callco` but has a subtly different effect. It causes the specified coroutine to resume execution exactly as `callco` does, but the parent of the calling coroutine becomes the parent of the called coroutine. Two typical and important uses of `resumeco` are given later in this paper in the definitions of `die` and `coread`.

BCPL coroutines were designed in 1978 and their specification has remained unchanged ever since. They are likely to remain unchanged for the foreseeable future and are thus safe to use in programs that are expected to have a long life.

A coroutine within a task will keep control until it explicitly calls one of `callco`, `cowait` or `resumeco`, and so it is safe for it to manipulate data shared by other coroutines in the task without having to use synchronization primitives such as semaphores, or mutexes, provided the shared data is left in a consistent state when it passes to another coroutine. Sometimes a coroutine wishes to retain exclusive access to some resource, such as a file, for a longer period and this will require the use of synchronization primitives but, as will be seen, these are easy to implement. Sharing resources between tasks is possible but requires calls of Cintpos kernel primitives such as `qpkt` and `taskwait` which are necessarily more costly.

As an example of the use of coroutines, we will re-implement the bounce demonstration using them. Firstly two coroutines `bounce_co` and `sender_co` are created by the following code:

```
LET bouncefn(val) BE val := cowait(val) REPEAT

LET senderfn(count) BE
{ writef("Calling the bounce coroutine %n times*n", count)
  FOR i = 1 TO count DO callco(bounce_co, i)
  writes("done*n")
}

bounce_co := createco(bouncefn, 200)
sender_co := createco(senderfn, 200)
```

The sender coroutine can be started by:

```
callco(senderfn, 10_000_000)
```

This completes its 20 million control transfers between the two coroutines in about 24 seconds indicating that the coroutine primitives are many times faster than `qpkt` and `taskwait`. More specifically, in the current implementation only 10 Cintcode instructions are executed between the start of `callco` in `senderfn` and the return from `cowait` in `bouncefn`, and 11 between the start of `cowait` in `bouncefn` and the return from `callco` in `senderfn`.

As an aside, `bouncefn` could have been defined as the identity function `LET bouncefn(val) = val`, since body of `createco` already contains a suitable loop.

4 Multi-event tasks

As was mentioned in section 2.1, tasks can be classified as single- or multi-event tasks. A single-event task typically uses calls of `sendpkt` (defined above) to request services from other tasks, suspending itself for each request until it is answered. Single-event tasks are thus like conventional single threaded programs. A multi-event task, on the other hand, has an organization that resembles the event loops found in many windowing systems. It typically has a main loop in which `taskwait` is called suspending the task until the next packet arrives. The packet is then passed to the coroutine that was waiting for it, or given to a main coroutine if the packet is a new request. If the main coroutine is busy processing a previous request, the packet is queued and processed later. Many tasks within Cintpos and within process control applications are best organized in this fashion. Rather than implementing this mechanism afresh for each multi-event task, it is convenient to invoke the library function `gomultievent`. This is defined as follows.

```
LET gomultievent(maincofn, size) = VALOF
{ LET mainco = createco(maincofn, size)
  LET oldsendpkt, wkq = sendpkt, 0
  UNLESS mainco RESULTIS FALSE // Unsuccessful return
  multi_done, mainco_busy := FALSE,  FALSE
  sendpkt,    pktlist     := sndpkt, 0
  callco(mainco, 0)            // Ask mainco to start everything up

  UNTIL multi_done DO          // Start of multi-event loop
  { LET pkt = taskwait()       // Wait for a packet
    LET co  = findpkt(pkt)     // Find which coroutine, if any, owns it
    IF co DO { callco(co, pkt); LOOP }
    IF mainco_busy DO          // If the main coroutine is busy
    { LET p = @wkq             // append the packet onto the
      WHILE !p DO p := !p      // end of the work queue
      !pkt, !p := 0, pkt
      LOOP
    }
    { callco(mainco, pkt)      // Give the packet to the main coroutine
      IF mainco_busy | wkq=0 BREAK
      pkt := wkq               // De-queue the next request
      wkq := !pkt
      !pkt := notinuse
    } REPEAT                   // Process a waiting request
  }
  sendpkt := oldsendpkt        // Return to single event mode
  deleteco(mainco)             // and delete the main coroutine
  RESULTIS TRUE                // Successful return
}
```

This function makes use of the following global state variables:

`multi_done` which becomes `TRUE` when all the multi-event coroutines have completed their work,

`mainco_busy` which is `TRUE` whenever the main coroutine is busy processing a request packet, and

`pktlist` which holds a list of nodes giving the mapping between packets and the coroutines that own them.

When running in multi-event mode, the standard version of `sendpkt` is replaced by the multi-event version `sndpkt`. So this is the version called (indirectly) by standard library functions such as `writef` when used in multi-event coroutines. The definition of `sndpkt` is as follows.

```
AND sndpkt(link, id, type, r1, r2, a1, a2, a3, a4, a5, a6) = VALOF
{ LET ocis, ocos, ocurrentdir = cis, cos, currentdir
  // The following three variables form the pktlist node.
  LET next, pkt, co = pktlist, @link, currco
  pktlist := @next // Insert [next,pkt,co] as first node in pktlist
  UNLESS qpkt(pkt)    DO abort(181) // Dispatch the packet
  UNLESS cowait() = p DO abort(182) // Wait for the reply
  // Restore the saved global state
  cis, cos, currentdir := ocis, ocos, ocurrentdir
  result2 := r2    // Recover the two results
  RESULTIS r1
}
```

It saves and restores the global state variables: `cis` the currently selected input stream, `cos` the currently selected output stream and `currentdir` the currently selected filing system directory, so that the normal input and output library function continue to work as expected. It creates a `pktlist` node containing a link to the next in the list, the packet and the current coroutine. After this node is placed at the head of `pktlist`, the packet is dispatched by the call of `qpkt` with a safety check to ensure that is was valid. Then, unlike `sendpkt`, it uses `cowait` to wait for the packet to be returned, knowing that this will result from `callco` in the multi-event loop in `gomultievent`.

The call of `findpkt` in `gomultievent` will search `pktlist` for a specified packet, de-queuing it if found. Its definition is as follows:

```
LET findpkt(pkt) = VALOF
{ LET a = @pktlist  // a is the address of the next link word
  { LET p = !a
    UNLESS p RESULTIS 0             // The packet was not found.
    IF p!1 = pkt DO { !a := !p      // Remove from pktlist and
                      RESULTIS p!2  // return the coroutine.
                    }
    a := p
  } REPEAT
}
```

If `findpkt` finds a node in `pktlist` that matches the given packet, it unlinks the node and returns the corresponding coroutine pointer. Notice that the saved global variables `cis`, `cos` and `currentdir` are restored in `sndpkt` and that the space for the `pktlist` node and also the packet itself will be released when control returns from `sndpkt`. This is clearly more efficient than using a general purpose space allocator.

After creating the main coroutine and initializing the global state variables `multi_done` and `mainco_done`, `gomultievent` enters multi-event mode by overriding `sendpkt` and setting `pktlist` to zero. It then calls `main_co` to create and start the multi-event coroutines. Control returns to `gomultievent` when `main_co` has completed its initialization and is ready to receive request packets. Typically, the main coroutine activates the multi-event coroutines as necessary but the conventions of how this is scheduled is application dependent and not relevant to `gomultievent`. All that is needed is for `multi_done` to be set to `TRUE` when all the multi-event coroutines and the main coroutine agree to return to single event mode.

Careful study of the event loop in `gomultievent` will show that, when a packet is received by `taskwait`, if owned by a waiting coroutine it will be given to that coroutine. If not, it must be a request packet for the main coroutine but if this is busy it is just appended to the end of the work queue (`wkq`). Otherwise, it is given to the main coroutine for processing. In due course the main coroutine will calls `cowait`. If this happens when `mainco_busy` set to `FALSE`, the main coroutine has finished processing a packet and is ready for another one. In this case if there are any packets in wkq, the first is de-queued and given to the main coroutine. If mainco_busy is TRUE when the main coroutine call cowait, it will typically be in sndpkt and will remain suspended until the packet it is expecting is found by findpkt in the event loop. It is up to the application programmer to decide which requests to handle entirely in the main coroutine and which to sub-contract to other multi-event coroutines.

5 Coroutine suicide

Sometimes a coroutine in a multi-event task completes the job it was given and must delete itself. It is a common mistake to think that a coroutine can delete itself using the call: deleteco(currco). Unfortunately this will not work reliably since the stack frame that was active just before the call of deleteco resides in the coroutine stack that is being returned to free store (and possibly reallocated and used by another task). The local variables and more importantly the function return link information is thus not valid after deleteco returns. A common solution to this problem is to create a killer coroutine whose sole purpose is to delete other coroutines. It can be created by the following assignment.

```
kill_co := createco(deleteco, 100)
```

Any other coroutine can now commit suicide by executing:

```
resumeco(kill_co, currco)
```

since this causes kill_co to delete the current coroutine before giving control to its parent. Note that the loop inside createco allows kill_co to be used repeatedly. Sometimes it is convenient to define the suicide function as follows:

```
LET die() BE resumeco(kill_co, currco)
```

If kill_co is required it should be created and deleted by the main coroutine of gomultievent.

6 Synchronization primitives

Process control applications typically take input from many sources such as keyboards, bar code readers, external communication lines and sensor devices of all kinds. These generate data asynchronously and at unpredictable times. The output of the system is typically written to files and sent down communication lines possibly to devices such as printers, displays, robots or other computers. Often different parts of the system must access shared resources such as disk files or shared data structures in memory. So that different activities can proceed in pseudo parallelism, the access to these shared resources must be controlled using synchronization primitives. Many such primitives have been proposed in the literature and provided either directly in programming languages or supplied in library packages. A few examples are (1) the communication primitives in Occam[?], (2) synchronized objects and methods in Java[?], (3) mutexes and condition variables in the POSIX Pthreads library[?] and (4) the similar but different facilities available in the WIN32 API[?]. These mechanisms are fairly new and their specifications are still being revised. The different implementations also vary in subtle ways.

This section shows how a variety of synchronization primitives can be implemented efficiently using coroutines running within multi-event tasks, and having the advantage that once implemented they will remain the same unless the application programmers wishes to change them.

6.1 Occam Style Channels

A channel in Occam provides a synchronized way of transmitting data from one Occam process to another. One process can execute an input statement to read a value from a channel while another may execute an output statement to send a value down the same channel. An input statement will not complete until an output statement on the same channel is executed, and likewise, an output statement will not complete until an input statement on the same channel is executed. Once the data has been transmitted both processes may continue independently.

We use coroutines running in multi-event mode to model the Occam processes and use the functions `coread` and `cowrite` to provide the synchronous communication between them. A channel is represented by a channel word that will contain a pointer to the first coroutine to reach a `coread` or `cowrite` call involving the channel. If no such call has yet been made it will be zero. The definitions of `coread` and `cowrite` are as follows.

```
LET coread(ptr) = VALOF TEST !ptr
  THEN { LET cptr = !ptr
         !ptr := 0         // Clear the channel word
         RESULTIS resumeco(cptr, currco) // Get value from cowrite
       }
  ELSE { !ptr := currco    // Set channel word to this coroutine
         RESULTIS cowait() // Wait for value from cowrite
       }

LET cowrite(ptr, val) BE TEST !ptr
  THEN { LET cptr = !ptr
         !ptr := 0
         callco(cptr, val)      // Send val to the waiting coread
       }
  ELSE { !ptr := currco         // Wait for coread to be ready
         callco(cowait(), val)) // Send val to coread
       }
```

In both functions, `ptr` is a pointer to a channel word. If the channel word is zero, neither `coread` nor `cowrite` is waiting to communicate, otherwise it points to the first coroutine making the call of `coread` or `cowrite`. A value can be passed when control passes from one coroutine to another and this is how the value is passed from `cowrite` to `coread`.

If `coread` is executed first, it will find that the channel word is zero and will set it to point to itself by `!ptr := currco`, and will then suspend itself in `cowait` waiting for `cowrite` to send a value. When `cowrite` is called it will find the channel word is non zero and so knows that it points to the waiting coroutine. All it has to do is clear the channel word by `!ptr := 0` and send the value by the calling `callco(cptr, val)`.

If, however, `cowrite` is called first, the implementation is somewhat more subtle. As before the coroutine suspends itself, but this time in the statement `callco(cowait(), val)`, after updating the channel word to point to itself. At this moment `cowrite` is waiting to receive the identity of the `coread` coroutine. When `coread` is called, it notices that `cowrite` is ready to send a value which `coread` obtains by calling `resumeco(cptr, currco)`. This gives `cowrite` the identity of the `coread` coroutine so that the call `callco(cowait(), val)` in `cowrite` knows where to send the value. At the same time the parent of `coread` becomes the parent of `cowrite`, so that when `coread` next suspends itself, control will return to `cowrite`. This scheme has the merit that execution preference is given to `coread` which typically results in the slightly more efficient communication described earlier.

The mechanism just described is efficient since there is little code to execute and, as has been seen, the coroutine primitives are themselves efficient. However, the implementation is subtle and has other subtle implications. Firstly, these functions must be used within multi-event coroutines under the control of the event loop in `gomultievent`. Typically, `gomultievent`'s main coroutine would declare and initialize the channel word before creating both the producer and consumer coroutines. These would then be started successively using `callco`.

Assuming the consumer was started first, it will run until it chooses to suspend itself, typically in the `cowait` call in `coread`. This will return control to the main coroutine which will call `callco` to start the producer which would typically run until it reaches a call of `cowrite`. This will find that the channel word is non zero and so immediately passes a value to the consumer which would be given control at that moment. Control will return to the producer when the consumer coroutine next calls `cowait`. This may not be, as expected, in `coread`, but could be in `sndpkt` if the consumer, for example, invokes a service from the file handler task or called `delay`. If this happens the consumer will become suspended in `sndpkt` waiting for the reply to be completed. In the mean time control will pass to the producer coroutine allowing it to make progress. The reply from the file handler will not reach the consumer until it is received by `taskwait` in `gomultievent` and this will not happen until all the multi-event coroutines of the task are suspended. Normally multi-event coroutines require very little CPU time but, if one does, it would probably be wise for it to subcontract the work to a lower priority task so as not to interfere with the multi-event scheduling. Provided none of the coroutines require much CPU time they will rapidly all become suspended waiting for packets. In this state the event loop will be suspended in its call of `taskwait`.

7 Stream locks

A common situation is when several multi-event coroutines wish to write messages to a log file. Clearly only one should be allowed to write to the file at a time. To ensure this happens, a coroutine wishing to write to the log could call `lock_logfile` which only returns when it has obtained exclusive access to the log file. When this coroutine has finished outputting its message, it releases the lock by calling `unlock_logfile`. These two functions can be defined as follows.

```
LET lock_logfile() BE TEST log_wait_queue
  THEN { LET link, co = 0, currco      // Lock node [link, co]
         TEST log_wait_queue=-1
         THEN log_wait_queue := @link  // Make list of length one
         ELSE { LET p = log_wait_queue // or append lock node
                WHILE !p DO p := !p    // to the end of the queue.
                !p := @link
              }
         cowait() // Suspend until unlock_logfile() called
         // We now own the lock and log_wait_queue will be non zero
         }
  ELSE log_wait_queue := -1 // Mark as locked

LET unlock_logfile() BE TEST log_wait_queue=-1
  THEN log_wait_queue := 0
  ELSE { LET co = log_wait_queue!1
         log_wait_queue := !log_wait_queue
         UNLESS log_wait_queue DO log_wait_queue := -1
         callco(co)
       }
```

These have been optimized on the assumption that the lock is almost always free. The variable `log_wait_queue` is zero when the file is unlocked, it equals `-1` if the current coroutine owns the lock and no others are waiting for it, otherwise it contains a list of coroutines waiting for the lock. If `lock_log_file` finds that `log_wait_queue` is zero, it just sets it to `-1` and returns. If `unlock_log_file` finds `log_wait_queue` is `-1` it resets it to zero and returns. In the rarer case when `lock_log_file` finds that `log_wait_queue` is non zero it appends a node to the `log_wait_queue` being careful with the `-1` marker and then suspends itself in `cowait`. If `unlock_log_file` finds `log_wait_queue` is not equal to `-1`, it must contain a list of waiting coroutines. It de-queues the first node in the queue and transfers control to the coroutine it contains using `callco`, being careful to set `log_wait_queue` to `-1` if no more coroutines are waiting.

8 Condition variables

Condition variables can be used when an activity must wait for some condition involving shared variables to become satisfied. When another activity changes a value that might cause the condition to be satisfied, it must awaken the first activity so that it can re-evaluate the condition. This can be implemented for use within a multi-event task using a single word for the condition variable and two functions `wait` and `notify`. Assuming, `condwaitlist` is the condition variable it should be initialized to zero, and can be used in code of the form:

```
UNTIL <complicated condition> DO wait(@condwaitlist)
```

When a variable in the condition is updated by another coroutine it should call:

```
notify(@condwaitlist)
```

This will awaken every coroutine waiting on the condition variable so that they can all re-evaluate the condition and possible continue normal execution. Simple implementations of `wait` and `notify` are given below.

```
LET wait(ptr) BE
{ // These form a waitlist node [link, cptr]
  LET link, cptr = !ptr, currco
  !ptr := @link  // Insert the node at the head of the list
  cowait()       // Suspend until the waiting condition
}                // may have changed.

LET notify(ptr) BE
{ // Wakeup all coroutines on the given  wait list
  // so that they can each re-evaluate the condition.
  LET p = !ptr
  !ptr := 0  // Clear the condition wait list
  WHILE p DO { LET cptr = p!1; p := !p; callco(cptr) }
}
```

A condition variable is just a, possibly empty, list of coroutines containing nodes of the form (*link*, *cptr*), where *link* is either zero or points to another node and *cptr* is the coroutine pointer to a coroutine suspended in `wait`. As can be seen, `wait` simply inserts a node at the start of the wait list pointed to by `ptr`. Notice that the wait list node is formed from two adjacent local variables which cease to exist as soon as control returns from `wait`.

Every coroutine on a wait list can be woken up by calling `notify`. This simply executes `callco` for every coroutine in the list. But, to avoid a possible execution loop, it must first reset the condition variable to zero. The implementation of `wait` given above causes the most recent coroutine to wait to be the first to be released. In rare situations where this strategy is not appropriate, it would be easy to modify `wait` to append the node to the end of the list.

Observe that `notify` is somewhat analogous to `pthread_cond_broadcast` in the Pthreads library[?] and that `wait` is analogous to `pthread_cond_wait`, but are both much simpler since coroutines are non preemptive removing the need to declare, lock and unlock a mutex.

9 Discussion

The synchronization mechanisms just described show how easily such primitives can be implemented using coroutines within multi-event tasks under Cintpos, and it should be clear that many other synchronization primitives could be implemented with similar ease.

As we have seen in the discussion of the Occam channel example, the detailed flow of control between the coroutines is not always easy to follow and this may cause some users to have doubts about whether they actually work. The following observations should help.

- A well written multi-event task will have coroutines that never require much CPU time before transferring control to another coroutine. Such transfers may result from direct calls of `cowait`, `callco` or `resumeco`, or these may also be called indirectly via calls to the synchronization primitives or library functions (such as `writef` or `delay`) that involve calls of `sndpkt`.

- Whenever a multi-event coroutine becomes suspended it will have transferred control to another multi-event coroutine or returned to the main event loop. Whenever a coroutine is suspended, there must be a reference to it somewhere in the system so that it can be resumed later. This reference may be the parent link in the coroutine it called using `callco`, it may be in a `pktlist` node that will cause the coroutine to be called by the event loop when the required packet is received, or it may be in one of the queues maintained by one of the synchronization primitives, such as `log_wait_queue` used in `lock_logfile`.

- The main coroutine will create most of the coroutines used in a multi-event task and schedule work for them depending on what request packets are received.

- Provided all these coroutines have been implemented correctly, whenever a packet is received by `taskwait` in the event loop, control will pass briefly through a sequence of one or more coroutines before returning to the event loop where the task will typically suspend itself in another call of `taskwait`. The exact order in which control passes from one coroutine to another should not matter, just as the scheduling processes in a conventional operating system should not affect the correct working of the system as a whole.

10 Final comments

In summary, the strategy suggested in this paper for the implementation of complex real-time process control applications that are expected to have lifetimes exceeding 20 or even 50 years is as follows.

- Base the entire system on a simple interpretive abstract machine that can be implemented easily in any suitable language and run on any hardware. The abstract machine should include an interrupt mechanism and should be able to handle asynchronous devices such as clocks, disks and communication lines.

- Use a simple implementation language that has a simple compiler to implement the entire application including a multi-task operating system kernel and all the standard tasks such an operating system needs. The operating system, the implementation language, compiler and debugging aids should be simple enough to be easily understood in detail and maintained in house by a single person.
- Implement the necessary synchronization primitives as part of the application in order to retain control of their precise specification and behaviour. These are best provided using multi-event coroutines running cooperatively within separate tasks.
- Be aware that, if the system is implemented in a currently fashionable language under a current modern operating system purchased from an outside vendor, then it will be large, complex and and probably bug-ridden and that maintenance for these critical parts of the system is not likely to be available in 50 years time since the language, operating system and vendor may not then exist. Consider how few operating systems and languages of as little as 35 years ago are still available today and properly maintained.

The hope is that this paper demonstrates that using BCPL coroutines running under the Cintpos portable operating system is a good basis for implementing a complex process control application that is expected to have a long life.

Acknowledgments
I would like to acknowledge the work of the many systems programmers in the Ford Motor Company (Europe) who have implemented a significant process control system in BCPL running under Tripos that has been used successfully for more than 20 years. Many of their ideas have been incorporated into this paper, but any mistakes are entirely my own.

References

[1] M. Richards. My www home page. http://www.cl.cam.ac.uk/users/mr/.

[2] M. Richards. A coroutine mechanism for bcpl. *Software-Practice and Experience*, 10(?):765–771, February 1980.