Internals of a Real-Time operating system

Seventh College on Microprocessor-Based Real-Time Systems in Physics

Trieste, 28 October - 22 November, 2002

A.J. Wetherilt
Artesis A.S
Tuzla
Istanbul
Turkey

email:anthony.james.wetherilt@arcelik.com

Contents

Chapter 1

Internals of a Real-Time operating system

by Anthony James Wetherilt

Abstract

RINOS, a Real-Time kernel developed for this series of colleges is introduced. Originally written in assembler for the 6809, a C version is now under development that will allow porting to other systems. Components of Real-Time systems are described and illustrated using the C code of this version. The kernel is designed to implement a small memory manager, a task scheduler, software system calls and installable device drivers. Software tools (compilers, debuggers etc) that accompany the **MC6809** version are presented. On top of the kernel, several layers of software have been implemented to provide full high level language library support, including a version of the POSIX 1003.1C (PThreads) standard. Examples are provided that illustrate the use of the kernel and these libraries.

1.1 Introduction

RINOS (for **R**eal-Time **In**tegrated **O**perating **S**ystem) is a full-fledged Real-Time, multitasking kernel for embedded processors. Originally targeted at the **MC6809** and written in the assembly language of that processor, it is now being rewritten virtually entirely in C with the aim of porting it to other small embedded systems. The original design criteria were:

(i) Use software-interrupt system calls to an EPROM based kernel for interfacing to client programmes rather than be linking in with a library based version compile time.

(ii) Allow a variable number of client applications to run concurrently in RAM when the kernel is started.

(iii) Provide a set of functions that would allow the efficient coexistence of, and communications between, a number of processes.

(iv) Provide a means of installing device drivers that can be changed after the start of the kernel and without having to re-assemble the system code.

(v) Provide a means of downloading code to a target board and running the code under the control of an external debugger.

(vi) Provide external libraries for wrapping the operating system calls and increasing functionality. These libraries are to provide compatibility with standard implementations on other systems and allow a measure of cross platform development.

The first four items in this list of design criteria belong completely to the domain of the kernel itself, whereas the remaining items are descriptions of external requirements and impose minimalist limits on the scope of the kernel design. Only those items that are essential to the operation of the kernel should be placed in it and anything that can be handled, as a general, external library function should be left out. Thus the kernel should contain a complete set of functions needed for the concurrent operation of real-time, multitasking processes, with everything else being left for external libraries.

Since C code is generally much easier to understand and considerably more concise than assembler, where possible, C has been used to illustrate the principles outlined, although the currently released versions of the kernel are written entirely in assembler and differ in detail from the some of the descriptions presented here.

1.2 Components of a multi-tasking kernel

In any operating system, certain requirements must be fulfilled in order to achieve the desired performance. Multi-tasking and real-time requirements each place specific demands and constraints on the system that the designer must comply with. Another important issue concerns the manner in which the target hardware interacts with the kernel and how this interaction can be generalized to allow better porting to other hardware platforms.

An operating system is basically a set of functions or system calls that provide a framework for a unit of code to execute and interact with other such units. Each process or task runs in a manner that is governed by the operating system and makes use of the system calls provided by the operating system. Thus when coding a process, the programmer can employ existing and tested functions with ease and does not usually need to be concerned with the underlying details of the platform. A basic set of functions would start with items for the creation and subsequent destruction of processes.

A process that has no interaction with its hardware platform would be of neither much interest nor use, so provision should be made for processes to use the hardware platform through the provision of device drivers, that allow standardized access to, for example, the input and output channels of a serial communications channel. When one considers the usage of hardware it soon becomes apparent that a process often has to wait for some event on a particular piece of hardware for a response with the desired information. For example, the serial communications port inputs a byte roughly once every 2080 μs (at a Baud

rate of 4800). This means that on the scale of a microprocessor, there is a very long wait between input bytes, a time that could possibly be put to better use by processing something else. An efficient manner of achieving this is to arrange for multiple processes to divide up the work between them so that, for instance, one process could be monitoring input from the serial port and a second could be doing calculations or something. Then the operating system must provide mechanisms for deciding which process will be running at any one time (scheduling) and how to switch between processes (context switching). In real-time systems, a related problem is how rapidly a process can be switched when an interrupt arrives so that the time used in servicing the interrupt can be minimized. The concept of a **thread** (sometimes called a **lightweight process**) is often used. On some systems context switching involves considerable work and can take many processor cycles. Threads are part of a process and a single process can consist of several threads. Context switching is allowed between the threads which can be a much simpler affair than a full context switch between separate processes. Thus threads are often used to improve response times. Under RINOS, threads and processes are identical.

Once multiple processes are considered, a whole host of secondary issues must be addressed. Firstly, the idea of synchronization of resources (be they variables, sections of code, hardware etc) must be considered. A classic example of the need for synchronization is as follows: Two processes A and B attempt to write to a serial, byte-oriented printer. A tries to write the word 'Hello', and B 'Goodbye'. Let us assume that they write slowly so that the operating system performs several context switches during the time they both write to the printer. What will be the output? Assuming A starts first it will send the 'H' character maybe followed by an'e'. If at this stage we get a context switch B will then send a 'G'. It is not difficult to imagine the effects of several more context switches producing output something like "HeGololdboye" which presumably is the intention of neither A nor B. What is needed here is a lock that the first process can claim and once claimed will prevent access to other processes. When the process finishes its business it releases the lock, which then becomes available for another process to claim. In this way, access to resources can be synchronised between processes. A generalization of this principle is the semaphore with the associated operations Down and Up corresponding to lock and unlock. A second need when using multiple processes is the ability to communicate information between two or more processes in a synchronised manner. Several mechanisms for inter process communication (IPC) are often implemented in an operating system such as shared memory, pipes, messages and signals with each mechanism having its own advantages and disadvantages. Thirdly, some form of control over the running of processes is also needed. For example, each process should have a separate priority, which the scheduler uses to determine when the process should run, and it must be possible to change this priority. It must also be possible to prevent a process from running or put it to sleep and subsequently wake it up again.

Table 1.1: Basic function calls of a multi-tasking operating kernel

Process functions	1	Create a process
	2	Kill a process
	3	Set process priority
	4	Put a process to sleep
	5	Wake a sleeping process
Semaphore operations	6	Up semaphore
	7	Down semaphore
IPC functions	8	Send message
	9	Receive message
	10	Signal a process
	11	Open a pipe
	12	Close a pipe
	13	Write to a pipe
	14	Read from a pipe
Memory management functions	15	Allocate memory block
	16	Deallocate memory block

Finally, an operating system generally exerts control over the allocation of memory resources to processes. Whilst this is not an essential item of an operating system it is a common feature especially on larger systems.

These then are a minimal set of function calls that most multi-tasking operating systems implement in one way or another. Most large systems implement considerably more than this number. In the following sections, these basic functions are discussed and using the RINOS kernelon the ICTP09 board, methods of implementing them in C are presented.

1.3 Basic process management mechanisms

1.3.1 The Task Control Block

In order to control a process, an operating system must be aware of a considerable amount of information regarding the process, such as its priority, its running state, any errors etc. Such information is often stored in a structure, commonly known as a Task Control Block (TCB), with one such TCB existing for each process. The exact information recorded in this structure depends on the precise nature of the operating system but must contain everything needed by the operating system to control the execution of a process. The TCB structure defined for RINOS is given in the following table.

```
typedef struct _tcb_t {
 struct _tcb_t* priority_link; // link to next highest priority task
 BYTE    state;                // State of the task. Possible values:
                               // READY, NO_TASK, SLEEPING, BLOCKED etc.
 registers_t* context;         // Thread stack pointer for context
                               // switch at a system call or hardware
                               // interrupt

 WORD    pid;                  // Unique task identifier
 BYTE    priority;             // Priority level
 BYTE    base_priority;        // Base priority level
 BYTE    page;                 // The page of memory where task resides
 addr_t  code_segment;         // Start of code in memory (load point)
 addr_t  stack_segment;        // Top of stack segment
 WORD    stack_size;           // Length of space reserved for stack
 WORD    parent;               // Parent of this task
 BYTE    exit_status;          // Exit status
 WORD    exit_code;            // Return code of the thread
 func_t* exit_func;            // Pointer to thread's exit function
 void*   exit_arg;             // Pointer to argument of exit_func
 event_t exit_event;           // Event for threads waiting for
                               // termination
 WORD    timer;                // Used to indicate how many clock ticks
                               // a sleeping task has to wait until
                               // wakeup
 struct _tcb_t* timer_list;    // Link to timer list
 struct _semaphore_t*
          blocking_mutex;      // Blocking semaphore
 struct _tcb_t* waiting_list;  // Link to waiting list of semaphores
 struct _semaphore_t*
          owned_mutexes;       // Link to list of mutexes owned by
                               // thread
 message_t* message_list;      // Pointer to the thread's message queue
 addr_t  signal_handler;       // Pointer to thread's signal handler
 BYTE    error;                // Last error status
 BYTE    attributes;           // Thread set of attributes
} tcb_t;
```

On process creation the various fields are initialized to suitable values and the TCB is inserted in a linked list in order of its requested priority, using the `priority_link` field to point at the next TCB in the list and a variable located in the RAM data area as the list head. The final TCB always has this field set to NULL. This TCB represents a special process called the **Null Task** that has the lowest possible priority and can never be put to sleep. This means that although it is always in a runable state it can only run when no other process is available. The state field of the TCB is used to indicate whether the process is running, sleeping, blocked, waiting for a timer to expire, suspended or whether the TCB is itself unallocated. A unique process identifier (pid) value that can be used to identify the thread is placed in the `pid` field. The `attribute` field is a set of bits

defining various properties of the created process and the manner in which the process is allowed to terminate.

1.3.2 Context switching mechanism

Most (if not all) modern microprocessors use a set of registers to hold and manipulate data. The register set and the data stored in each register constitute what is known as the **context**. During an interrupt, the processor usually saves at least a subset of the register set by pushing the contents of the registers onto the hardware stack in RAM and pulling the same set off the stack as the interrupt exits. The interrupt routine optionally can push (and subsequently pull) using special commands available for this purpose, the remaining registers on to the stack to ensure that the complete context is saved as needed. However, most interrupt routines will only save only the registers that are actually used by the routine for reasons of efficiency. Many processors have an instruction for a software interrupt (SWI) that performs this pushing programmatically. This technique provides a convenient mechanism for starting, stopping and swapping processes. When the kernel receives a request to create a new process, it copies the supplied data (requested priority, attribute, requested stack size etc) into the TCB and initializes most of the remaining fields. It then allocates a stack (or uses the stack supplied), copies the default values to the stack for each register in the context and saves a pointer to the context in the `context` field of the TCB. When the process is first run, the context in its entirety is pulled from the stack, the stack pointer adjusted and execution continued from the address placed in the register acting as the processor program-counter register (PCR). Conversely, when a context switch occurs, initiated either by a hardware interrupt or system call, the reverse of this process happens. The context of the running process (including the current PCR) is pushed onto its stack and a pointer to this context stored in the TCB. The kernel then decides which process will run next, obtains the context pointer from that process' TCB, pulls the saved context from the stack and starts execution at the address placed in the PCR. Thus a neat and effective mechanism for context switching is available using the processors' native behaviour during either a hardware or software interrupt.

1.3.3 Process scheduling

At the very heart of a kernel is the process scheduler that regulates at any instant which process is running and for how long the process will continue to run. Two scheduling policies commonly in use are discussed here:

(i) Round robin scheduling. At its very simplest, round robin scheduling divides time into a series of fixed intervals using the system clock interrupt and runs a process for the duration of the interval or time-slice. At the end of the interval a context switch occurs and another process runs for the next time slice. In this manner all processes are given equal shares of processor time with none having priority over another and all yielding at the end of

its time slice. It is also possible for a process to yield voluntarily before its allotted time turning over the processor to the next one in line. Although simple to implement, round robin scheduling has very poor time response and cannot be used for real-time systems.

(ii) Priority based scheduling. In this scheme, each process is awarded a unique priority and the scheduler checks the processes in order of highest priority. If the highest priority process can run it is awarded processor time and will run until either complete or a higher priority process becomes runnable. If the process cannot run (ie it is blocked for some reason) the next process in the priority list is examined and so on until a runnable process can be found. There must always be at least one process that can be executed in this scheme and a special process called the Null Task that cannot be blocked always has be available. As its name suggests, the Null Task does nothing except wait for a higher priority process to preempt it. This type of scheduling is ideal for real-time systems as response times for context switching can be minimized. RINOS uses priority based scheduling.

In practice, neither extreme is totally suitable and compromise solutions are often used. One such solution is the incorporation of priority into round robin scheduling. In this scheme, processes are divided into priority bands and within each priority band round robin scheduling applied. In this way a higher priority processes get more of the processor time but no one process can hog the processor entirely.

1.3.4 Interrupts

The processor can be interrupted asynchronously (i.e. at any time) by a hardware event and, as previously described, the processor will respond by saving the current context on the stack and jumping to a handler for the interrupt. The time it takes to respond to the interrupt is called the **interrupt service latency** and is a measure of the quality of a real-time system. In some systems, not only does a jump to an interrupt handler occur, but a lengthy interrogation of the hardware and filtering through several software layers can also arise. In such cases the latency can easily approach millisecond rather than microsecond times, which may be acceptable for some soft real-time applications but is unlikely to suit the more severe restrictions of hard real-time systems. Interrupts can be disabled for short periods within the kernel, but it is not advisable for client applications to interfere with interrupt operation as this can have a severe effect on latency.

1.3.5 The system clock

In systems where round robin scheduling is used, a system clock is essential for the proper scheduling of all the processes. In priority-based systems, a clock is often found for timing purposes. In most cases the clock interrupts the system at a frequency that is a trade off between to opposing requirements. At higher

clock frequencies, time resolution increases, and timing events have greater precision. On the other hand, the higher the frequency, the longer the time spent by the processor servicing the clock interrupt, the less time available for client programmes, and the longer the potential latency for other interrupts. RINOS sets the clock chip on the ICTP09 board to interrupt once every 10 ms, which is adjudged suitable for this application. The RINOS kernel uses the clock interrupt to time process sleeping periods. When a process is put to sleep, the state field in the TCB has the SLEEPING bit set and the sleeping period (in units of the clock period) is placed in the `timer` field. On each subsequent clock interrupt, the timer field is decremented until zero upon which the state field has its SLEEPING bit cleared, and the process can again run (assuming the process is not blocked in any other fashion).

1.3.6 The dispatcher

The system dispatcher acts as the interface between the client programmes and the kernel. In many operating systems, values are placed in certain registers and the software interrupt command issued. This causes a jump to the SWI handler where the command number is retrieved from its register and the address of the appropriate function obtained from a jump table. This function is then called with the current context as a function argument. The following code fragment illustrates how RINOS performs these actions.

```
#pragma _interrupt_
void system_call(void)
/*
 * Entrance point for all kernel functions
 * The context is made local to avoid reentrancy problems
 * when the hardware interrupts a system call
 * Note that interrupts are assumed masked by the processor on
 * entrance to this function
 */
{
  registers_t* context;
  WORD function_call;
  BYTE error_no;

  // Place current stack frame in the context variable to allow direct
  // accessing of registers

  context = get_frame();

  // Get function call number from caller and check whether it is a
  // device driver call

  if (!((function_call = context->CALLER_REG) & IS_DRIVER)){
    // Is a normal function call. Check nested interrupt level
    if ((intrpt_level++ == 0)){
```

```
      // We have not interrupted the system so we replace
      // the system stack
      SET_SYSTEMSTACK();
      // If the kernel is running, we save the current context
      // in the current thread tcb
      if (kernel_running) current_thread->context = context;
    }

    // It is now safe to reenable interrupts
    EXIT_CRITICAL();

    // Check that system call is valid
    if (function_call < MAXSYSTEMCALLS) {
      // Dispatch function call using jump table
      error_no = dispatch_table[function_call](context);
    }
    else {
      error_no = ERR_BADCALL;
    }
  }
  else {
    // Dispatch a device driver call
    error_no = device_driver(context);
  }
  if (error_no){
    // An error occured, set the error status in the current thread
    // and indicate in the flags register
    current_thread->error = error_no;
    context->FLAG_REG |= SET_ERROR;
    // leaving the function, first decrement interrupt level
    intrpt_level--;
    // This function does not return here!
    run_thread(context);
  }
  // After completing the function call we return back to the system
  // via a context switch. We first check the interrupt level
  // to determine whether or not we should perform the switch.
  // If after decrementing, the interrupt level is > 0 we do
  // NOT perform the context switch but return to this function
  // and pop the context directly from the system stack
  switch_context();
}
```

Several points are worth noting here.

(i) The pragma directive informs the compiler that this is an interrupt handler and that the entire context is to be pulled from the stack on exit from the function. This is a (not even) semi-standard convention that GCC deprecates but other compilers often support. GCC can also be made to support it with a bit of effort.

(ii) Since hardware interrupts can occur at any time and the Null Task in particular has little stack reserved to hold multiple copies of the context, a design decision was to swap to a special, safe stack during system calls. The `intrpt_level` variable is a flag initialized to 0 that indicates whether or not the system stack is currently being used. If the flag is 0, it is safe to switch to the system stack, otherwise the stack is in use and it is not appropriate to make the change. Thus when a hardware interrupt arrives during either another hardware interrupt or a while a system call is being processed, this value is above 0 and the stack is not switched. The flag is decremented on exit from the handler and the system stack replaced if the flag is found to be 0.

(iii) The device driver interface shares the same entry point as normal function calls but since driver calls occur in the client space, it is not desirable to switch stacks.

(iv) If an error occurs and for some reason the system call cannot be completed, the `error_field` in the TCB of the current process is set with the error number and a bit set in the Condition or Flags register of the current context. The setting of this bit indicates to the client programme that an error occurred. Further information can usually be obtained from the operating system. RINOS for example provides the OSGetLastError call that returns the value of the `error_field` from the current TCB.

1.4 Semaphores and events

1.4.1 Semaphores and mutexes

Semaphores are programming objects designed to act as locks that regulate access to shared resources. Typically, they contain a variable that must be set and tested in order to gain access. Under RINOS the variable is decremented and compared with zero. If the value (after decrementing) is negative, access is refused and the process requesting the semaphore is blocked. On the other hand, if the value is zero or positive, access to the resource is granted. The key point here is that the two operations of test and set must be performed without *any interruption whatsoever*. The operations of test and set are said to be **atomic**. The reason for this can be seen from the following example. Two processes A and B compete for access to the same variable previously initialized to zero. A will load the current value of the variable, increment it by one and store the new value back in its location. B will similarly try to decrement the variable. The two operations should result in the value of the variable remaining unchanged, that is, at zero. A loads the value of the variable and let us assume a context switch occurs just at that moment before A has a chance to increment the value. B now runs and loads the value and decrements it to minus one, which is stored. A now resumes running and increments the value in its register to plus one, which it stores. As a result of the two operations the final value is not the expected

value of zero but one. Moreover it is easy to see that the final value depends on when the interruption (in this case a context switch) occurs. The only way to safeguard the integrity of the variable is to insist that once either A or B start their action, a context switch will not occur and the action will run to completion. This is obviously an action that must be managed by the kernel itself as in a well run system a client process should not be able to interfere with interrupts or the internals of the kernel in any way. Thus operations on semaphores are almost always handled as kernel functions.

Several types of semaphore are in common usage. The most general of these, called the **Counting Semaphore** (or more usually just semaphore), allows semaphore values up to a maximum. Two operations exist: **Down** and **Up**. Down on a semaphore decrements the semaphore variable and tests whether the new value is zero or negative. If not the process continues, otherwise the process blocks. The Up operation on a semaphore increments the semaphore value. If the value was zero or negative before the increment, one of the processes currently being blocked by the semaphore is made runnable, and when allowed by the scheduler will be able to access the resource guarded by the semaphore. A special case of the semaphore called the **Binary Semaphore** or more often the **Mutex** (for MUTually EXclusive) can only take values up to one and blocks for all other values.

The structures used by RINOS to represent semaphores and mutexes are as follows:

```
typedef struct _semaphore_t{
  SWORD  Value;                      // The value of the semaphore
  struct _tcb_t* WaitingList;        // Pointer to the first thread
                                     // in a linked list of threads
                                     // waiting on this semaphore
  struct _tcb_t* Owner;              // Current semaphore owner
  struct _semaphore_t* OwnerList;    // Link to list of owner's
              // semaphores
} semaphore_t;
typedef semaphore_t mutex_t;
```

Here the semaphore value is given in the `Value` field. When a process blocks during a Down operation, a pointer to the TCB of the process is linked to a list with its head in the `WaitingList` field. The `waiting_list` field of the TCB points at subsequent items of this list. In keeping with the real-time nature of the kernel RINOS orders this list in order of priority so that the list head in the semaphore points directly at the highest priority process and that this process will be the first to become unblocked as the result of an Up operation. The remaining fields are to ensure that the semaphore will become unblocked if the process owning the semaphore is killed without first releasing it.

1.4.2 Deadlocks and mutexes

While the careful use of semaphores can solve many problems associated with the synchronization of shared resources, there is a subtlety that the user should

be aware of. This concerns the interaction of the blocking property of semaphores and the priorities of the processes using the semaphore, and can lead to deadlock in real-time systems. Consider the following situation: A low priority process, L, grabs a mutex and intends to use the resource it guards. At a later time, a high priority process, H, is created and immediately attempts a Down on the mutex, and blocks as a result. The aim of real-time systems is to provide as rapid a response as possible to high priority processes, but here we have a situation where the high priority process H is blocked waiting for the low priority process, L, to finish its work with the shared resource and then release the mutex. This is known as **bounded priority inversion** and has the effect of increasing the latency of high priority processes. A much more serious situation occurs if a third, medium priority process, M, starts to run immediately following process L. Process H must still wait for L to finish with the mutex, but now since M is the highest priority, runnable process, L never gets to run and H is blocked indefinely. This condition, referred to as **unbounded priority inversion**, realized a degree of notoriety when it occurred during the 1997 Mars Pathfinder mission, resulting in the onboard computer periodically resetting itself with considerable loss of data. Two solutions are commonly implemented:

(i) The Priority Inheritance protocol When a process locks a mutex, the kernel boosts its priority to a level equal to that of the highest priority process waiting for the mutex and restores the original priority when the process finally releases the mutex. This has the effect of allowing the process owning the mutex to inherit the priority of the highest process and should thus prevent unbounded priority inversion. Since the priority boost is applied automatically by the kernel, it is transparent to the client and hence requires little extra programming. Theoretically, it appears that priority inheritance does not give full protection against deadlock and is still the subject of some controversy [ref]. Nonetheless, almost all real-time systems offer the priority inheritance protocal as a means of resolving priority inversion.

(ii) The Priority Ceiling protocol In this protocol, each mutex is initialised to a priority level or ceiling. Any process successfully locking the mutex has its priority boosted to the ceiling level. As long as the priority ceiling is greater than the priority of any process that may attempt to lock the mutex, this scheme works well, but it requires some (potentially considerable) analysis by the programmer to determine the correct level for the priority ceiling. Theoretically, it does appear to prevent deadlock, but at the expense of an additional requirement that a process can only lock a mutex if its priority is *greater than the priority ceilings of all other locked mutexes*. This protocol always requires two priority boosts each time a process attempts to acquire a mutex and is thus often considered inefficient. The full version of the protocol is not implemented by any commercial system, but a reduced version (priority ceiling protocol emulation) in which the additional scheduling constraint is relaxed is common (it is a specification of POSIX 1003.1c).

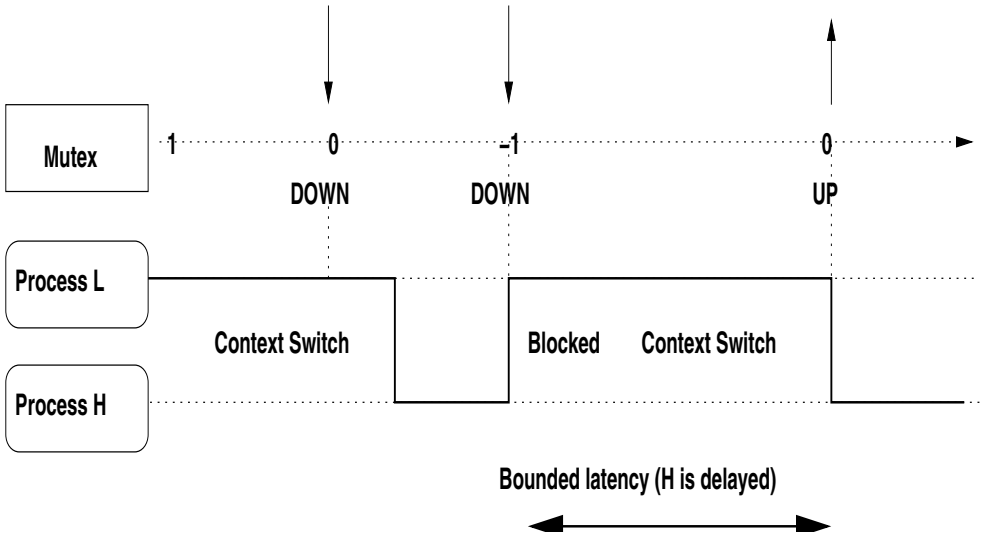

Figure 1.1: Unbounded priority inversion. This is inconvenient rather than dangerous

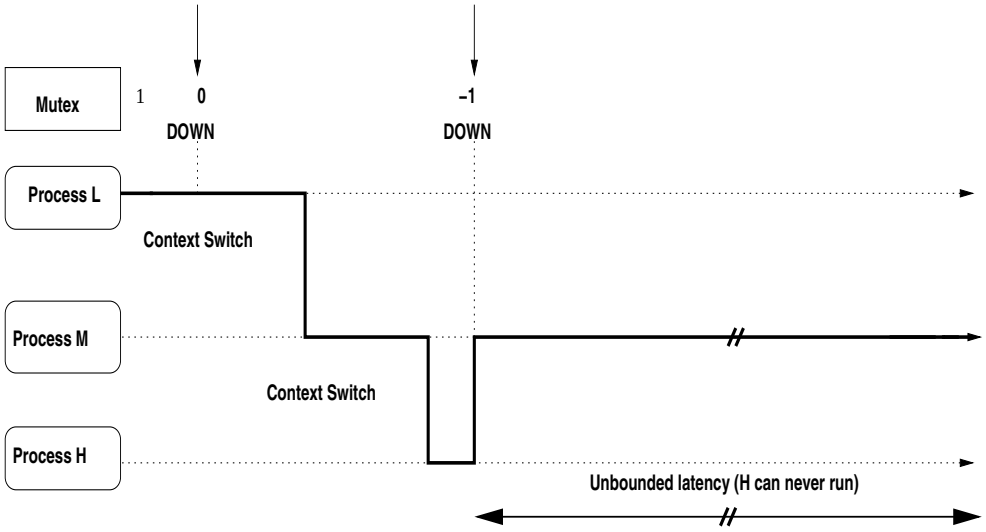

Figure 1.2: Bounded priority inversion. Thissituation is potentially lethal as the high priority process may never run again.

RINOS implements the priority inheritance protocol. The Down operation is coded as in the following:

```
BYTE DownSemaphore(tcb_t* pThread,
   semaphore_t* pSemaphore,
   BOOL IsMutex)
/*
  Perform the DOWN operation on a semaphore or mutex
 */
{
  if (pSemaphore == NULL) return ERR_BADSEMA;
  // Ensure no possible interruption
  ENTER_CRITICAL();
  if (pSemaphore->Value-- > 0) {
   // Semaphore is available, claim it
    if (IsMutex) {
      pSemaphore->Owner = pThread;
      // Attach to new thread's list of owned mutexes
      pSemaphore->OwnerList  = pThread->owned_mutexes;
      pThread->owned_mutexes = pSemaphore;
    }
  }
  else {
    // Semaphore is claimed by another thread, we block
    // First check priorities of current owner and calling thread
    if (pThread->priority >= pSemaphore->Owner->priority && IsMutex) {
      // Need to apply a priority boost to the current owner
      // We first remove the thread from the priority list
      delete_thread_priority(pSemaphore->Owner);

      //... and then reinsert at the new priority value
      set_priority(pSemaphore->Owner, pThread->priority + 1, FALSE);
    }
    // We now insert the calling thread into the
    // semaphore's waiting list
    priority_insert(pThread, pSemaphore);

    // Next set the thread's blocked semaphore
    // to point at this semaphore
    if (pSemaphore->IsMutex) pThread->blocking_mutex = pSemaphore;
    // Finally set calling thread to BLOCKED state
    pThread->state |= BLOCKED;
  }
  // It is now safe to restore interrupts
  EXIT_CRITICAL();
  return ERR_OK;
}
```

In this function, the semaphore value is tested and decremented and if available, marked as being owned by the calling process. Otherwise, the semaphore's

current owner has its priority boosted (if it is lower than the priority of the calling process). The original priority is stored in the `base_priority` field of the owner's TCB so that it can be restored when required at a later time. If the process is blocked a pointer to the mutex is copied to the `blocking_mutex` field of the TCB and the state field marked as BLOCKED. All this code is executed with interrupts masked, to fulfill the requirement of atomicity.

1.4.3 Events

A related concept to a semaphore is that of the **event**. An event has the property of blocking when any process performs a Down operation on it. The Up operation on an event unblocks all processes waiting for the event. An event therefore acts as a sort of group trigger and is often used to react to a hardware signal such as a button push. RInos implements events with the structure:

```
typedef struct _event_t {
  SWORD Value;
  struct _tcb_t* WaitingList;
} event_t
```

Here the value field is either 1 indicating that the event has occurred (is **signaled**) or 0 indicating that it has not (is **pending**). As for semaphores, the `WaitingList` field is a linked list to the processes waiting for the event. Unlike semaphores, events require an extra operation to initialize or reset them following a signal.

1.4.4 Joining a process

It is sometimes needed to wait for a child process to finish a particular action before the parent resumes operation. In the jargon, the child process is said to be **joined**. Most POSIX 1003.1c compatible systems provide this facility. RINOS implements this feature using an event, `exit_event`, in the TCB of each process. If a process wants to join a child process, it calls the OSJoin system call, which performs the Down operation on the event of the appropriate process thus blocking the calling process. The event becomes signaled as the process terminates.

1.5 Messages and pipes

1.5.1 Messages

Message queues are familiar to all those who programme common graphical windowing systems and provide a convenient method of passing information between processes. Messages are basically software structures that can contain a pointer to a data block and other optional information. They are sent from one process to the mailbox of another, where they are queued and read as desired. Under RINOS messages are implemented as follows:

```
typedef struct _msgtag {
  WORD    sender;
  struct _msgtag* next;
  void*   data;
  WORD    value;
  event_t event;
  BOOL    wait;
  BOOL    in_use;
} message_t;
```

The `sender` field is the pid of the process that sent the message. An optional pointer to a block of data can be sent using the `data` field and also a WORD value. When sending the message, the calling process links the message structure into a list with its head in the **message_list** field of the receiver's TCB where it can be retrieved. If the receiving process is blocked awaiting a message, it will become unblocked and will return with the values contained in the data and value fields placed in the processes' context. Sometimes, it is required that the sending process waits for the message to be read. This is achieved by the sender setting the event in the `event` field. The receiver will acknowledge receipt by signaling the event when it actually retrieves the message from its mailbox.

1.5.2 Pipes

Pipes are basically a first in, first out queue (FIFO) that has data written at one end and read at the other, together with a set of semaphores for regulating access. Data can be of specified width (ie byte, integer, float etc) that must typically be specified when first opening the pipe. RINOS implements pipes using the following structure:

```
typedef struct {
  void*       buffer;
  short       width;
  void*       front;
  void*       rear;
  semaphore_t full;
  semaphore_t empty;
  mutex_t     lock;
} pipe_t;
```

Here `buffer` is a pointer to a block of memory allocated to form the circular FIFO. It is organised as an array of 32 objects of size given by `width`. Two other pointers, `front` and `rear` point to the front and rear of the pipe and are adjusted as data is written to or read from the pipe. Two counting semaphores `full`, and `empty` initialized to 32 and 0 respectively are used to keep track of how data are inserted and removed from the FIFO. Finally, since the pipe itself represents a shared resource, a mutex `lock` is available for regulating access to the pipe.

The pipe works in the following manner:

Writing to the pipe

Writing proceeds as follows:

(i) A Down is performed on the `full` semaphore. This semaphore will block when there are 32 items already in the buffer i.e. the buffer is full.

(ii) Prior to accessing the pipe resources, the `lock` mutex must be obtained to prevent multiple accesses. Again this will block while another process is using the pipe.

(iii) The datum is written in the buffer at the position pointed to by `front` and the pointer updated by the set data width. If this update would bring the pointer past the end of the buffer, the pointer is reset to the start of the buffer.

(iv) The `empty` semaphore now has an Up operation performed on it to indicate that another datum has been placed in the buffer.

(v) Finally, the `lock` mutex is released since access to the pipe has finished.

Reading from the pipe

Reading from the pipe proceeds in the reverse sequence:

(i) A Down is performed on the `empty` semaphore. This operation will block when there are no items in the buffer i.e. it is empty.

(ii) The `lock` mutex is locked by performing a Down on it, to prevent multiple accesses.

(iii) The datum pointed to by the `end` pointer is read and the pointer updated. Again if this action would take the pointer past the end of the buffer, it is reset to the beginning.

(iv) An Up is performed on the `full` semaphore to indicate that there is one more item in the buffer.

(v) The `lock` mutex is released since access to the pipe has finished.

1.6 Memory management

RINOS uses a very simple memory management scheme. The first 256 bytes of each RAM area form a table used to represent allocation state of blocks of memory. Since the ICTP09 board has 32k of continuous memory, each byte in the table represents a block of 128 bytes of RAM. A value of 0xff in an entry indicates that the block is free and a value of 0xef indicates that the item is the last of an allocated block. Otherwise, the value is the pid of the process owning the

Table 1.2: Basic device driver functions

No.	Function
1	Read single byte from channel 1
2	Write single byte to channel 1
3	Read string from channel 1
4	Write string to channel 1
5	Input/Output control (IOCTL)
6	Initialise
11	Read single byte from channel 2
12	Write single byte to channel 2
13	Read string from channel 2
14	Write string to channel 2

block (0 for the kernel itself). When a request to allocate memory is received, the number of 128 byte units required is calculated and the memory table searched for a block of contiguous free memory sufficient to satisfy the request. If such a block is found, the table entries are marked as belonging to the process and the last entry in the block marked with 0xef. A pointer to the start of the block in memory is returned to the client programme.

When memory is freed, the start of the block in the memory table is found and all entries up to and including the final entry marked with `0xef` are marked as free.

This simple scheme is sufficient for small, embedded systems without memory management hardware.

1.7 Device drivers

Device drivers are usually supplied because of the difficulty and specialization required in addressing native hardware directly. A device driver should supply a set of well-defined functions to the client process and simplify the process of using the hardware. Device drivers under RINOS are fairly simple but have many of the features of more complex systems. In their design, it was assumed that access to each device was through streams that supported either single, byte oriented accesses or multiple byte orientated accesses. Thus the serial communications output port can be addressed either as a succession of calls to the driver or a single call with a null terminated string. For generality, each driver was assumed to have two channels; when this was not true, the second channel was left unused. Similarly, a driver was assumed to have both input and output channels.

The device driver functions are presented in the following table:

Since RINOS is a real-time kernel, all device drivers have been written to use interrupts, and an important part of each driver is the interrupt handler. Drivers

exist for two serial channels (read/write), a parallel port connected to a 16 character LCD display (write only) and some input buttons (read only), two channels of AD conversion (read only) and two channels of DA conversion (write only). Future developments should probably include drivers for Ethernet connectivity.

1.8 Software tools and libraries

In order to use the ICTP09 board and RINOS kernel effectively, a number of other tools have had to be developed. This section presents an overview of these tools and the reader is referred to the 6809 Manual [ref] for further details.

1.8.1 The command interpreter shell

In order to interact with the kernel, code must first be placed in the RAM of the target board. A small and simple command interpreter shell has been written that allows commands to be input over a serial line from a PC terminal. The shell is loaded following initialistion of the kernel as a child process and intercepts the input stream to the serial driver. If the ESC (0x1b) character is detected in the stream, the shell is invoked and the user can enter an interactive mode in which code can be downloaded to the board, run and debugged using a set of basic commands. Remote debugging is possible using these commands.

1.8.2 The gcc compiler

A complete set of compilation and dubugging tools have been provided mainly through the efforts of Rinus Verkerk. The GNU C compiler has been adapted to produce position independent M6809 assembler code, which together with a suitable assembler and linker found produces relocatable code in ELF format, linking together the standard C libraries, and a startup module crt0, that performs initialisation and setting up command line arguments.

Embedded within the ELF output file is sufficient information for high level debugging of the code.

The gcc compiler can be invoked using the following command

```
cc09 [-Wall] [-v] [-o output] prog.c
```

Here, `prog.c` is the input file(s) and the first two options inform the compiler that you want to view any warnings generated and to see all steps involved. The third option allows the name of the output file to be changed (in this case to output).

1.8.3 The Java translator

Carlos Kavka has put together a system for running java programmes on the ICTP09 board. The **j09** script first calls the standard Java compiler provide with the JDK from Sun Microsystems and then translates the resulting Java byte

code into 6809 assembler code. The assembler code is passed through the same assembler programme used by gcc and finally linked together with the startup module crt0 to produce an ELF executable file.

The command:

```
j09 Test.java
```

will cause the java file Test.java to be compiled, assembled and linked to produce the executable file, Test, which can be run on the ICTP09 board.

1.8.4 The XGUM symbolic debugger

A symbolic debugger running under XWindows has been developed to ease the process of debugging the ICTP09 board. It is invoked by:

```
xgum
```

XGUM consists of a client front end that connects via a pipe to a server specific to the requirements of the debugging session. Presently two servers are available for debugging the ICTP09 board directly and an ICTP09 simulator that allows debugging on the pc itself. Both servers load the RINOS kernel. XGUM is independent of the source code language and can handle both C and Java.

1.9 Libraries available under RINOS

Libraries are available for provision of a number of functions under RINOS. First, support for the C language is made available via the **libc.a** library. This library contains the standard functions such as printf, putchar, malloc etc demanded by the ANSI C standard. The

library is automatically linked in as part of the compilation chain and the functions can be referenced by use of the standard headers

```
<stdio.h>, <stdlib.h>, <string.h> etc.
```

The second group of library routines generally consists of wrapper functions for RINOS system calls and allow the user basic level access to the kernel and device driver services. These routines are found in **libcreal.a** and are automatically linked in as required. The routines are accessed by inclusion of the ¡syscalls.h¿ header file which also includes all basic information on structures and types defined and used by functions making system calls. This header is also automatically called by the standard header files (stdio.h) etc and only if none of these headers are included is it necessary explicitly to declare ¡syscalls.h¿ in a file. A second library, **libIOreal.a** performs a similar function by bridging the low level I/O services of the kernel to C level functions. These functions are in turn used by the standard I/O functions of the C library. If low level output is required, the header file <ICTP_IO.h> should be included.

The two members of the third group of library functions, **libgcc.a** and **libmath09.a**, are used by the compiler during code generation. **libgcc.a** is used by the compiler to perform integer arithmetic operations and to convert between the various integer types. **libmath09** performs a similar function with floating point functions. Neither library should be called explicitly in a user-defined function.

Finally, an implementation of the POSIX 1003.1c (pthreads) library is made available to simplify the mechanics of preparing multi-threaded programmes. The implementation is reasonably complete within the confines of the 8 bit microprocessor platform and its use is encouraged as the functions for thread creation and mutex usage in particular offer greatly simplified functionality over the native RINOS functions. As extensions to this library, the following functions allow simple manipulation of (counting) semaphores and events

```
int event_init(event_t *event, const eventattr_t *attr)
int event_destroy(event_t *event)
int event_signal(event_t *event
int event_wait(event_t *event)
int semaphore_down(semaphore_t *sema)
int semaphore_init(semaphore_t *sema, const semaphoreattr_t *attr)
int semaphore_up(semaphore_t *sema)
int semaphore_destroy(semaphore *sema)
int semaphore_init(semaphore *sema, const semaphoreattr_t *attr)
int semaphore_destroy(semaphore_t  *sema)
```

All the definitions and types used in these function definitions can be found in the header file <pthread.h> which should be included when any reference is made to any function member of the library.

1.9.1 Programming examples

Introduction

Although programming in C under RINOS is quite straightforward, there are several points that should be noted. Code is presented that illustrate some of these points and demonstrate the use of several of the available libraries.

Creating threads and mutexes

This example uses the pthreads library to create a child thread and a single mutex. First a thread attribute is created and the desired priority of 20 is set using a variable of type struct sched_param.

Before the child is created, a static mutex is claimed using the down_user_sem() function. This is actually a RINOS wrapper function rather than the pthreads equivalent showing that the two libraries can be mixed at will. The child thread is then created using the pthread_create() function. Finally the mutex is released and the parent exits, at which point the child gains the mutex and is able to run. Note the use of the static initialiser for the mutex. Static initialisation is a convenvient method for all types of semaphore creation. In this example the thread

owning the mutex will experience a priority boost if its priority is lower than any thread waiting on the mutex. Please refer to the `sycalls.h` and `pthread.h` header files for further semaphore types.

```
#include <pthread.h>
// Child prototype
void* child_thread(void *arg);
// Static mutex construction
struct semaphore  mutex = {MUTEX | SMBOOST ,1, 0,0,0};
int  main()
{
  pthread_t      child ;
  pthread_attr_t attr;
  // Child will have high priority, default is 10
  struct sched_param priority = {20};
  pthread_attr_init(&attr);
  // Initialise thread attribute
  // Set priority of thread
  pthread_attr_setschedparam(&attr,&priority);
  // Get semaphore before anyone else can
  down_user_sem(&mutex);
  pthread_create(&child, &attr, child_thread, NULL);
  // Finally release mutex
  up_user_sem(&mutex);
  return NULL;
}
void* child_thread(void *arg)
{
  // Try to get mutex
  down_user_sem(&mutex);
  return NULL;
}
```

1.9.2 Mutex, semaphore and event handling

The previous example showed how a mutex can be created using a static initialiser. The base type for all three semaphore types is the **semaphore** structure defined in syscalls.h which is redefined in pthread.h as **pthread_mutex_t**, **semaphore_t** and **event_t** for mutexes, counting semaphores and events respectively. Under pthreads, a mutex would defined as:

```
pthread_mutex_t mutex =  PTHREAD_MUTEX_INITIALIZER;
```

This would initialise the mutex to a value of 1, and allow priority boosting by default. The priority boosting uses the prority inheritance protocol and _POSIX_THREAD_PRIO_PROTECT is defined by the implementation. The functions

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

are defined as operations on mutexes under pthreads which perform the down and up operations respectively.

Events and semaphores do not form part of the standard pthreads implementation. Extension functions have been added to allow use of the objects in a manner consistent with mutexes. Static initialisation of all semaphore types is similar and follows the outline of the following fragment in which a semaphore and several types of event are defined:

```
#include <pthread.h>// For function prototypes
#define INITIAL_VALUE 10
semaphore_t count = {COUNT,INITIAL_VALUE,0,0,0}
event_t pevent = {EVENT,0};  // Persistent event
event_t revent = {REVENT,0}; // Resetable event
event_t sevent = {SEVENT,0;  // Single event, freed
                             // after signal
main()
{
   // Create a child
   ....
   event_wait(&revent);  // Wait for an event to occur
}

void* child(void* artg)
/*
  Child thread created by main
*/
{
  ....
  // Signal any threads waiting for the event
  event_signal(&revent)
}
```

The same frament using basic RINOS functions would be:

```
#include <syscalls.h>// For function prototypes
#define INITIAL_VALUE 10
semaphore count = {COUNT,INITIAL_VALUE,0,0,0}
semaphore pevent = {EVENT,0};  // Persistent event
semaphore revent = {REVENT,0}; // Resetable event
semaphore sevent = {SEVENT,0;  // Single event, freed
                               // after signal
main()
```

```
{
   // Create a child
   ...
   down_user_sem(&revent);     // Wait for an event
                               // to occur
}

void* child(void* artg)
/*
  Child thread created by main
*/
{
 ....
  // Signal any threads waiting for the event
  up_user_sem(&revent)
}
```

Note again the use of an initial value for the counting semaphore.

As an alternative to static initialisation, all semaphore types can be created dynamically and initialised separately. Under RINOS however, this is wasteful of memory and is not recomended.

Memory allocation under RINOS

As the following examples shows, memory allocation follows standard ANSI C practise using `malloc()` and `free()`. In this example 5 blocks of (paged) memory are allocated and then freed.

```
#include "stdio.h"
#define NBLOCKS 5
void main(void)
{
  void* memptr[NBLOCKS];
  int index;
  // Allocate blocks of memory
  for (index = 0; index < NBLOCKS; index++){
    memptr[index]  = malloc(256);
  }
  // Free them again
  for (index = 0; index < NBLOCKS; index++){
    free(memptr[index]);
  }
}
```

Global or shared memory can similarly be accessed using the pair of non-standard functions:

```
void* globalalloc(int size)
void  globalfree(void *p)
```

Accessing system variables

When downloading a programme to RINOS, an optional, run-time argument list can be specified. The startup module, crt0 makes this list available to the programme via the standard argument passing mechanism of argc and argv. As usual, argc is the number of arguments passed to the programme and argv is an array of strings containing the individual arguments. The startup file crt0 is also responsible for making available several other global resources.

In earlier versions of RINOS, input and output functions acted on file descriptors rather than file pointers. In the current release it is expected to use the file pointers in the following list:

File	Filename	Description	Type
stdin		Standard input	read only
stdout		Standard output	write only
stderr		Standard error	write only
fpia	“lcd”	LCD file	read/write
facia1	“com1”	ACIA1 file	read/write
facia2	“com2”	ACIA2 file	read/write
fadc	“adc”	ADC file	read only
fdac	“dac”	DAC file	write only

With the exception of stdin, stdout and stderr, all the preceding files should be opened prior to use and closed when finished in the standard manner. Thus the following code fragement will open and later close the the first serial port for writing:

```
#include <stdio.h>
  FILE* f; // File pointer definition
  ...
  if ((f = fopen(''com1'',''w'')) != NULL){
fprintf(f,...);
 }
 ...
 ...
fclose(f);
```

Finally, crt0 provides access to the pushbutton on the LCD board. During kernel initialsation, a resetable event semaphore is reserved and initialised. On each press of the pushbutton, any threads waiting for this event will be woken.

The startup module stores a pointer to the event semaphore in the global variable **pshbttn** and all any thread wishing to wait on this event has to do, is to perform a down using one other of the two functions:

```
down_user_sem(pshbttn);
event_wait(pshbttn);
```

The second of these is illustrated in the following code fragment:

```
/*
  Push button test
*/

#include <pthread.h>
extern semaphore_t* pshbttn; // pushbutton semaphore
                             // pointer
void main(void)
{
    event_wait(pshbttn);      // wait for pushbutton to
                              // be pressed
    ....                      // Got event, now do
                              // something ...
}
```

1.10 Bibliography

1. Software for the 6809 Microprocessor board. C. Verkerk, A.J. Wetherilt, and C. Kavka.
2. Modern Operating Systems. Andrew Tannebaum
3. The ASSIST09 Monitor, Motorola
4. The μ/COS Real Time Kernel, Jean Lebrosse
5. PThreads Programming Nichols, Buttlar & Farrell, O'Reilly.
6. The Risks Digest.
   http://catless.ncl.ac.uk/Risks/19.54.html
7. Against Priority Inheritance, Yodaiken V. FSMLabs Technical Report, July 2002.
8. Priority Inheritance: The Real Story. Locke D. July 2002.
   http://www.linuxdevices.com/articles/AT5698775833.html