

**COSC 6390B Project**

**A Comparison of Two  
Pre-Run-Time Schedule-Dispatchers  
for RTAI**

**by**

**William Soukoreff  
932 021 611  
will@cs.yorku.ca**

**for**

**Dr. Xu**

## Abstract

RTAI is an enhancement to the Linux operating system that provides support for real-time applications. However, RTAI only provides a priority-based run-time scheduler. This project is a comparison of two approaches of implementing pre-run-time schedule-dispatchers, using RTAI on Linux.

The first approach, called a *Master Process Schedule-Dispatcher*, implements a single real-time process under RTAI that acts as a scheduler, executing C functions (process segments) at specified times. The master process schedule-dispatcher is simple, and should work effectively for simple real-time applications. However, because it is only a task under RTAI (and not a real scheduler), this approach lacks the ability to preempt other processes. The second approach considered is the replacement of the RTAI scheduler with a pre-run-time schedule-dispatcher. This approach is more complicated, however it is also more powerful – having the ability to preempt processes if necessary, and the ability to achieve better timing. Both approaches have been implemented; the implementation details are reported. An example real-time application is presented.

A small experiment was performed in an attempt to measure the accuracy of schedule timing, using both approaches. The second approach (the full schedule-dispatcher) achieves more regular timing.

## 1.0 Introduction

Real-time applications are software systems that are capable of guaranteeing that timing constraints are satisfied. There are two kinds of real-time systems, denoted *hard*, and *soft*. Although soft real-time systems have timing constraints, there is some flexibility, so the constraints do not always have to be satisfied. In hard real-time systems, however, all timing constraints must always be satisfied.

*Real-Time Application Interface* or RTAI is an extension of the Linux operating system that provides support for real-time applications. By itself, the Linux operating system does not provide support for real-time applications. There are several reasons why this is so, but the three most significant limitations of Linux are that (a) the scheduler in the Linux kernel was optimised for a general-purpose time-sharing operating system, (b) the Linux kernel is not preemptable, and (c) real-time systems should really be using pre-run-time scheduling, not run-time scheduling. These three limitations are discussed below.

The scheduler in the Linux kernel is unsuitable for real-time applications. Linux employs a priority-based round-robin scheduling algorithm. This ensures that high priority processes get larger shares of CPU time, but also that no low-priority

processes starve. (Occasionally, the Linux kernel may allow a low-priority process to execute, even though a high-priority process is ready, to avoid starving the low-priority process.) However, without knowing about process deadlines, it is not possible for the Linux kernel to make any guarantees about the times that individual processes will complete. While the Linux scheduler is effective for general-purpose applications, because it has no concept of deadlines, it is useless for real-time applications.

The non-preemptability of the Linux kernel was a design choice intended to simplify the kernel. If the scheduler wishes to preempt a process that is executing in the kernel, then the scheduler must delay the task switch until the process returns from the kernel software. The net effect of this is *task switch latency*; processes segments that need to execute at specific times may be prevented from doing so by other tasks executing in the kernel. The latency makes it impossible to provide real-time timing guarantees with a regular Linux kernel.

From a real-time perspective, the fundamental problem with the Linux scheduler is that it is a *run-time scheduler* – it makes its scheduling decisions at run-time. Although they have their proponents, run-time schedulers are incapable of successfully producing satisfactory schedules in all but the most trivial of circumstances. To be able to make guarantees of satisfying rigid timing constraints, hard real-time systems must employ *pre-run-time scheduling*. Pre-run-time schedulers require foreknowledge of the timing constraints, and the timing characteristics of all of the processes to be scheduled. Using this information *a priori*, either a guaranteed satisfactory pre-run-time schedule, or advanced warning that a satisfactory schedule is impossible, is produced. The case for pre-run-time scheduling is made by Xu (1990), who also provides an algorithm for optimal pre-run-time scheduling.

## 1.1 RTAI – A bandage for Linux’s Real-Time Limitations?

RTAI provides an interface below the Linux kernel, where real-time processes may execute. The whole Linux kernel, then, becomes just another application executing upon the RTAI foundation. To accomplish this, RTAI provides two main components, a *Hardware Abstraction Layer* (or HAL) and a real-time scheduler. The HAL traps all interrupts and routes them to the Linux kernel when they will not be interfering with any of the real-time processes. The RTAI scheduler uses a real-time priority-based preemptive scheduling algorithm.

To be useful, a real-time operating system cannot only support real-time control over processes. A real-time operating system must also provide the usual features and conveniences that applications developers expect from an operating system, such as resource management (memory, CPU time, I/O, etc.), device drivers, high-

level libraries, and development tools. And so the marriage of Linux with RTAI is promising. RTAI provides the real-time support, while Linux provides the convenience of a powerful general-purpose operating system.

The point has already been made that run-time schedulers are insufficient for hard real-time applications with difficult timing constraints. Nonetheless, run-time schedulers are currently in vogue, and so it is not surprising that RTAI provides a run-time scheduler. The run-time scheduler does not mean that RTAI cannot support hard real-time applications – RTAI can support them, so long as they can be mapped into the RTAI priority-based run-time scheduling paradigm. This is a difficult proposition for many complicated real-life applications. The question that presents itself, is, what to do when faced with a problem for which priority-based run-time scheduling is insufficient?

This paper presents two solutions. Two pre-run-time schedule-dispatchers have been implemented, both based on RTAI. This paper presents both schedule-dispatchers, and a brief comparison of the two.

## **2.0 Two Pre-Run-Time Schedulers**

There are (at least) two ways to implement a pre-run-time scheduler on the framework that RTAI provides. A single *master process* could be executed as the lone real-time process on a system – scheduling sub-processes at the correct times. Alternatively, the run-time RTAI scheduler can be replaced by an altogether new pre-run-time scheduler. Both types of scheduler are presented below. Note that in both cases, just the schedule-dispatcher is presented. The problem of generating a valid schedule has been solved elsewhere (see Xu, 1990), and so is not discussed here.

### **2.1 The Master Process Schedule-Dispatcher**

The concept is simple. A single real-time process can be created that monitors the passage of time, and executes sub-processes according to some predefined schedule. RTAI provides all of the time-monitoring functionality that is required.

The source code for the master process dispatcher is presented in three parts below. The first listing contains the sub-process segments (see Figure 1). These are simply C functions within which the segments would be implemented.

---

```
1     static void ThreadA(void)
2     {
3         // Thread contents go here...
4     }
5
6     static void ThreadB(void)
7     {
8         // Thread contents go here...
9     }
10
11    static void ThreadC(void)
12    {
13        // Thread contents go here...
14    }
```

**Figure 1 - Implementation of three Process Segments**

---

The next code segment (Figure 2) contains the C encoding of the schedule. The schedule is stored in a C structure that contains two fields: a pointer to the function that implements the segment, and the start time (in nanoseconds). The entries in the schedule table must appear in chronological order (as they do in the example). A “null” record terminates the schedule list. There is also a constant `GLOBAL_PERIOD` defined, that is set to the period of the schedule (again, in nanoseconds).

This structure allows a periodic schedule consisting of an unlimited number of segments to be encoded for execution. Note that the release times, computation times, deadlines, and individual process periods are not presented in this schedule – but at this stage they are unnecessary. We assume that the user of this scheduler has a means to construct a feasible schedule, and at run-time all we need to know is the start time of each segment. This schedule dispatcher does not perform run-time deadline checking – a feasible schedule is assumed, ensuring that all deadlines will be met.

---

```
1     // this defines the period of the whole schedule
2     // 5000000 ns = 5 ms
3     #define GLOBAL_PERIOD 5000000
4
5     // the schedule
6     struct schedule_struct {
7         void (*segment)(void);
8         int starttime;
9     } sched[] = {
10    { ThreadA, 100000 }, // "ThreadA" starts at 100 us
11    { ThreadB, 400000 }, // "ThreadB" starts at 400 us
```

```
12         { ThreadC, 700000 }, // "ThreadC" starts at 700 us
13         { NULL, 0 }
14     };
```

## Figure 2 - C Structure for Representing the Schedule

---

Only one element of the master process scheduler remains – the scheduler. Figure 3 lists the source code for the scheduler.

Note that there are two other functions, `init_module` (lines 37 – 53), and `cleanup_module` (lines 55 – 61) defined in Figure 3 as well. These two functions are required by RTAI. Under RTAI, each process is implemented as a loadable kernel module. The functions are executed by loading the modules into kernel space; execution is terminated by unloading the module. The use of modules imposes a certain format on the real-time task source code. All modules (and hence all real-time processes in RTAI) must contain `init` and `cleanup` functions. The `init` function is executed then the module is loaded, and the `cleanup` function is executed just before the module is unloaded. It is in these two functions that the real-time process must communicate with the RTAI scheduler to schedule all processes, set their priorities, and configure the timing mode to be used. In the implementation appearing in Figure 3, the `init` and `cleanup` functions schedule only one process, the master process, which will act as the scheduler for the sub-processes defined above (in Figure 1).

The operation of the scheduler is simple. The variable `start_of_period` stores the (real) time that the current period began. Line 13 calculates how much of the current period has elapsed. Lines 15 – 20 execute the segments that are scheduled to have begun execution by now, by calling the corresponding function pointers from the `sched` structure defined in Figure 2. Once all of the segments have been executed, all that remains is to wait until the next period begins (lines 22 – 28). If the period is not over, but there is nothing to do but wait for the next sub-process to begin, then lines 31 – 32 put this process to sleep.

---

```
1     // the RTAI task for the scheduler...
2     static RT_TASK Task;
3
4     static void Scheduler(int t)
5     {
6         int current_process = 0;
7         RTIME start_of_period = 0;
8
9         rt_task_wait_period();
```

```

10     start_of_period = rt_get_time_ns();
11     while(1)
12     {
13         RTIME elapsed_time = rt_get_time_ns() - start_of_period;
14
15         while(sched[current_process].segment != NULL &&
16              sched[current_process].starttime <= elapsed_time)
17         {
18             sched[current_process].segment();
19             current_process += 1;
20         }
21
22         if(sched[current_process].segment == NULL)
23         {
24             current_process = 0;
25             rt_task_wait_period();
26             start_of_period = rt_get_time_ns();
27             continue;
28         }
29
30         elapsed_time = rt_get_time_ns() - start_of_period;
31         rt_sleep(nano2count(sched[current_process].starttime
32                        - elapsed_time));
33     }
34 }
35
36
37 int init_module(void)
38 {
39     RTIME now;
40
41     if(rt_task_init(&Task, Scheduler, 0, 2000, 0, 0, 0))
42         return -1;
43
44     rt_set_onehot_mode();
45     start_rt_timer(0);
46
47     now = rt_get_time();
48     rt_task_make_periodic(&Task,
49                          now + nano2count(GLOBAL_PERIOD),
50                          nano2count(GLOBAL_PERIOD));
51
52     return 0;
53 }
54
55 void cleanup_module(void)
56 {
57     stop_rt_timer();
58     rt_busy_sleep(10000000); // 10 seconds...
59     rtf_destroy(CMDF0);
60     rt_task_delete(&Task);
61 }

```

**Figure 3 - The Master Process Scheduler**

---

Notice that the scheduler uses the `rt_task_wait_period` and `rt_sleep` functions (on lines 9, 25 and 31) to delay so that precious CPU time is not wasted. Both of these functions return control to RTAI which then allows the Linux kernel to execute. Although not explicitly mentioned before, the master dispatcher process described here must run with a higher priority than the priority assigned to the Linux operating system. This is not difficult, because the Linux process always has the lowest defined priority. The fifth parameter in the call to `rt_task_init` on line 41 in Figure 3 gives the dispatcher process the highest priority level (of 0).

### **2.1.1 Observations and Notes on the Master Process Dispatcher**

The dispatcher described above is simple and efficient. As coded it requires that the schedule be hard-coded in C source code, but it would be an easy matter to read a configuration text file and populate the C schedule data structure.

The dispatcher does not support preemption. RTAI does provide functions for suspending, and releasing tasks, and receiving timer interrupts. However, attempting to implement a fully preemptive scheduler as a RTAI task is pushing things too far. Managing preemption should not be the responsibility of a mere process, this functionality really belongs at a lower level (which we will discuss shortly). A properly devised set of segments, with a feasible schedule should make preemption unnecessary in many situations. However, because there is no preemption, each process segment must be implemented as a single function, and it is vital that the functions terminate before their allotted time.

## **2.2 The Second Approach – A New Scheduler for RTAI**

Another approach to solving the pre-run-time schedule-dispatcher problem is to replace the RTAI scheduler with a scheduler-dispatcher. The RTAI distribution contains implementations of two uniprocessor schedulers. Upon inspection of the “single-list” scheduler (`rtai-{version}/upscheduler/up_sched.c.sl`) it became apparent that it is a good foundation from which to build a pre-run-time dispatcher. A few small functions were added, and some modifications made to the two scheduler functions `rt_schedule` and `rt_timer_handler` to implement the dispatcher. Instead of listing the complete source file (which is approximately 50K long), we will discuss only the changes made to the single-list uniprocessor scheduler source code.

Note that to produce the source listings presented below, the source code for the software implemented by the author has been extensively reformatted.

## 2.2.1 Data Structures for Representing a Pre-Run-Time Schedule

The C source code for the data structures used to represent a pre-run-time schedule are listed in Figure 4.

---

```
1     struct rt_sched {
2         struct rt_sched *next;
3         struct rt_task_struct *task;
4
5         RTIME start_time;
6     };
7
8     typedef struct rt_sched SCHEDULE;
9
10    static SCHEDULE* wschedule = NULL;
11    static SCHEDULE* wschedule_pointer = NULL;
12    static RTIME wglobal_period;
13    static RTIME wtime_offset;
```

**Figure 4 - Definitions and Declarations**

---

To implement the pre-run-time dispatcher, four new global variables are defined, `wschedule`, `wschedule_pointer`, `wglobal_period`, and `wtime_offset`. (Note that when implementing the dispatcher, my global variable names conflicted with other global variable names in RTAI and the Linux kernel, so a “w” was prepended to all global variable names to make them unique.) The primary data structure implemented is a singly-linked-list to store the schedule. Time is broken into segments; in each segment one process executes. The nodes in the linked-list are ordered by the `start_time` member of each node. The global variables used to implement this scheme are:

`wschedule`

This is the “head pointer” of the linked-list used to store the schedule.

`wschedule_pointer`

This pointer points to one of the nodes in the linked-list to indicate which segment is next to be executed.

`wglobal_period`

This variable stores the period (time) of the schedule (measured in RTAI clock ticks).

`wtime_offset`

This variable stores the time (measured in RTAI clock ticks) that the current period began. It is only updated when a new period iteration is begun.

We now make the distinction between “real times”, and “period times”. The start times stored in the schedule nodes are period times – they are the times measured within a period, from the start of the period. The purpose of the variable `wtime_offset` is to make it possible to convert from period times to real times. The real time that the current period was begun is stored in `wtime_offset`. This means that the start time for the first segment, in real time, is:

$$\text{wtime\_offset} + \text{wschedule} \rightarrow \text{start\_time} \quad (1)$$

for the second segment, it is:

$$\text{wtime\_offset} + \text{wschedule} \rightarrow \text{next} \rightarrow \text{start\_time} \quad (2)$$

and for the third segment:

$$\text{wtime\_offset} + \text{wschedule} \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{start\_time} \quad (3)$$

and so on. We index through the linked list using `wschedule_pointer`, so at any time the next segment to be executed has a period start time of `wschedule_pointer`  $\rightarrow$  `start_time`, but a real start time of:

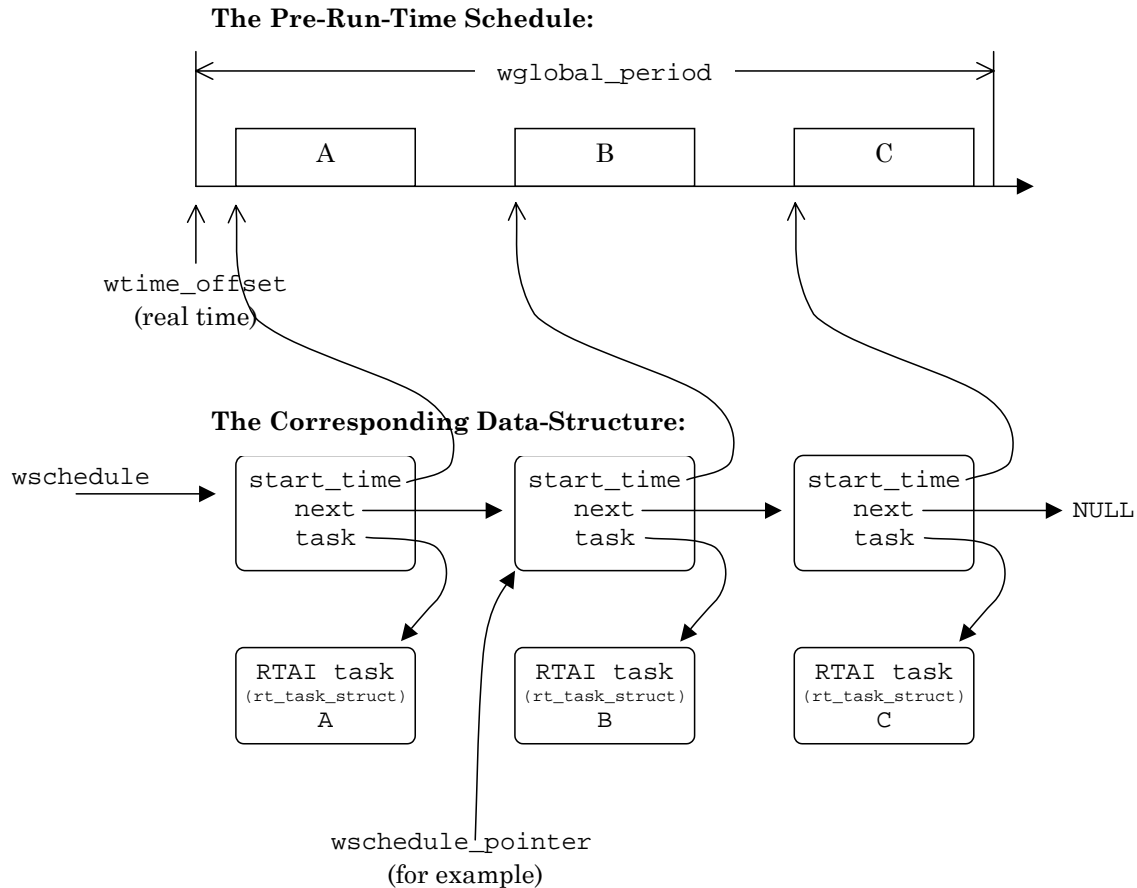
$$\text{wtime\_offset} + \text{wschedule\_pointer} \rightarrow \text{start\_time} \quad (4)$$

Once the final segment within a period has been executed, the next period is begun by assigning:

$$\text{wtime\_offset} += \text{wglobal\_period} \quad (5)$$

and then the real start times of each of the segments are found with the expressions (1) through (4) above.

The scheme for storing the schedule is depicted in Figure 5 below.



**Figure 5 - Depiction of the data structure used for storing a pre-run-time schedule**

This figure illustrates how the global variables defined above are used to represent and follow a pre-run-time schedule. Note that arrows that look like this ( $\rightarrow$ ) are used to represent C pointers, while arrows that look like this ( $\dashrightarrow$ ) represent time values (that conceptually point to positions on a time line).

In the upper half of Figure 5 appears a typical pre-run-time schedule consisting of three segments (A, B, and C). The lower half of Figure 5 demonstrates the corresponding data structures. All start times (i.e. the `start_time` members of the schedule linked-list) are “period times”. The `wschedule_pointer` in Figure 5 is shown pointing to the second node of the linked list. This means that the first segment (A) has already been started (there is no indication whether it has finished), and the second segment (B) will be executed next when its start time matches the elapsed time from the beginning of the period.

## 2.2.2 Modifications made to the `rt_schedule` and `rt_timer_handler` functions

The “single-list” uniprocessor scheduler provided with RTAI was used as a basis for the implementation of this scheme. Figure 6 lists the changes made to the `rt_schedule` function. Lines appearing in a boldface font (lines 7 – 23, and 49 – 71) were added by the author. Lines appearing in a grey coloured font (lines 25 – 32, and 39 – 47) were commented-out by the author.

The idea is to replace the part of the `rt_schedule` function responsible for choosing the next task with code that references the linked-list pointer `wschedule_pointer`, and schedules the next segment at the appropriate time.

---

```
1     static void rt_schedule(void)
2     {
3         RT_TASK *task, *new_task;
4         RTIME intr_time, now;
5         int prio, delay, preempt;
6
7         //////////////////////////////////////
8         if(wschedule_pointer == NULL)
9         {
10            wschedule_pointer = wschedule;
11            wtime_offset = rt_get_time();
12        }
13
14        if((rt_get_time() - wtime_offset) > wglobal_period)
15        {
16            wschedule_pointer = wschedule;
17            wtime_offset += wglobal_period;
18        }
19
20        task = new_task = &rt_linux_task;
21        prio = RT_LINUX_PRIORITY;
22        if (oneshot_timer) {
23        //////////////////////////////////////
24
25        /*
26        while ((task = task->next)) {
27            if (task->state == READY && task->priority < prio) {
28                new_task = task;
29                prio = task->priority;
30            }
31        }
32        */
33
34        preempt = 0;
35        task = &rt_linux_task;
36        intr_time = shot_fired ? rt_times.intr_time :
37                    rt_times.intr_time + rt_times.linux_tick;
38
39        /*
40        while ((task = task->next)) {
```

```

41         if ((task->state & DELAYED) && task->priority <= prio
42             && task->resume_time < intr_time) {
43             intr_time = task->resume_time;
44             preempt = 1;
45         }
46     }
47 */
48
49 ////////////////////////////////////////////////////
50     if(wschedule_pointer != NULL)
51     {
52         if(wschedule_pointer->start_time <= (rt_get_time()
53             - wtime_offset + rt_half_tick))
54         {
55             new_task = wschedule_pointer->task;
56             wschedule_pointer = wschedule_pointer->next;
57             if(wschedule_pointer == NULL)
58             {
59                 wtime_offset += wglobal_period;
60                 wschedule_pointer = wschedule;
61             }
62         }
63         if(wschedule_pointer->start_time + wtime_offset
64             < intr_time)
65         {
66             intr_time = wschedule_pointer->start_time
67                 + wtime_offset;
68             preempt = 1;
69         }
70     }
71 ////////////////////////////////////////////////////
72
73     if (preempt
74         || (!shot_fired && (prio == RT_LINUX_PRIORITY))) {
75         shot_fired = 1;
76         if (preempt) {
77             rt_times.intr_time = intr_time;
78         }
79         delay = (int)(rt_times.intr_time
80             - (now = rdtsc())) - tuned.latency;
81         if (delay >= tuned.setup_time_TIMER_CPUNIT) {
82             delay = imuldiv(delay,
83                 TIMER_FREQ,
84                 tuned.cpu_freq);
85         } else {
86             delay = tuned.setup_time_TIMER_UNIT;
87             rt_times.intr_time = now
88                 + (tuned.setup_time_TIMER_CPUNIT);
89         }
90         outb(delay & 0xFF, 0x40);
91         outb(delay >> 8, 0x40);
92     }
93     } else {
94         while ((task = task->next)) {
95             if (task->state == READY && task->priority < prio) {
96                 new_task = task;
97                 prio = task->priority;

```

```

98         }
99     }
100 }
101
102 if (new_task != rt_current) {
103     if (rt_current == &rt_linux_task) {
104         rt_switch_to_real_time(0);
105         save_cr0_and_clts(linux_cr0);
106     }
107     if (new_task->uses_fpu) {
108         if (new_task != fpu_task) {
109             save_fpenv(fpu_task->fpu_reg);
110             fpu_task = new_task;
111             restore_fpenv(fpu_task->fpu_reg);
112         }
113     }
114     if (new_task == &rt_linux_task) {
115         rt_switch_to_linux(0);
116         restore_cr0(linux_cr0);
117     }
118     rt_switch_to(new_task);
119     if (rt_current->signal) {
120         (*rt_current->signal)();
121     }
122 }
123 }

```

**Figure 6 - Modifications made to the `rt_schedule` function**

This figure illustrates how the function `rt_schedule` was modified for the schedule-dispatcher. Lines appearing in a boldface font (line 7 – 23, and 49 – 71) were added by the author. Lines appearing in a grey coloured font (lines 25 – 32, and 39 – 47) appeared in the original version of this function (from RTAI), and were commented-out by the author.

Lines 7 – 23 handle the initialisation case – when the scheduler is starting the first period ever. Lines 49 – 71 calculate the delay required until the start time of the next segment (lines 73 – 92 of the original `rt_schedule` function configure the timer to provide an interrupt at the appropriate time). Lines 55 – 61 handle scheduling the next segment, if the start time has arrived. (In this case we only have to set `new_task` to point to the task structure for the task we wish to execute, lines 102 – 122, of the original `rt_schedule` function handle the task switch.)

The `rt_schedule` function is called whenever a real-time process finishes, the intent being to cause a task-switch to another process. In regular RTAI this occurs, for example, when a task performs a non-busy wait, or finishes. The `rt_timer_handler`

function, however, is basically an interrupt service routine, called when the preemption timer reaches zero. The function is basically the same as the `rt_schedule` function – all tasks are checked and a task switch is performed if necessary. The changes made to the `rt_timer_handler` function mirror those made to the `rt_schedule` function. The schedule linked-list is checked to see if the next segment needs to be executed, and the timer is programmed to interrupt again, when the start time of the next segment arrives. See Figure 7.

---

```

1     static void rt_timer_handler(void)
2     {
3         RT_TASK *task, *new_task;
4         RTIME now;
5         int prio, delay, preempt;
6
7         //////////////////////////////////////
8         if(wschedule_pointer == NULL)
9         {
10            wschedule_pointer = wschedule;
11            wtime_offset = rt_get_time();
12        }
13        if((rt_get_time() - wtime_offset) > wglobal_period)
14        {
15            wschedule_pointer = wschedule;
16            wtime_offset += wglobal_period;
17        }
18        //////////////////////////////////////
19
20        rt_times.tick_time = rt_times.intr_time;
21        rt_time_h = rt_times.tick_time + rt_half_tick;
22        if (rt_times.tick_time >= rt_times.linux_time) {
23            rt_times.linux_time += rt_times.linux_tick;
24            rt_pend_linux_irq(0);
25        }
26        task = new_task = &rt_linux_task;
27        prio = RT_LINUX_PRIORITY;
28
29        /*
30        while ((task = task->next)) {
31            if ((task->state & DELAYED)
32                && task->resume_time <= rt_time_h) {
33                if (task->state & (SEMAPHORE | SEND | RPC)) {
34                    (task->queue.prev)->next = task->queue.next;
35                    (task->queue.next)->prev = task->queue.prev;
36                }
37                task->state &= ~(DELAYED | SEMAPHORE | RECEIVE |
38                               SEND      | RPC      | RETURN);
39            }
40            if (task->state == READY && task->priority < prio) {
41                new_task = task;
42                prio = task->priority;
43            }
44        }

```

```

45     */
46
47     if (oneshot_timer) {
48         preempt = preempt_always || prio == RT_LINUX_PRIORITY;
49         rt_times.intr_time =
50             rt_times.linux_time > rt_times.tick_time ?
51             rt_times.linux_time :
52             rt_times.tick_time + rt_times.linux_tick;
53         task = &rt_linux_task;
54
55     /*
56         while ((task = task->next)) {
57             if ((task->state & DELAYED)
58                 && task->priority <= prio
59                 && task->resume_time < rt_times.intr_time) {
60                 rt_times.intr_time = task->resume_time;
61                 preempt = 1;
62             }
63         }
64     */
65
66     //////////////////////////////////////
67     if(wschedule_pointer != NULL)
68     {
69         if(wschedule_pointer->start_time <=
70             (rt_get_time() - wtime_offset + rt_half_tick))
71         {
72             new_task = wschedule_pointer->task;
73             wschedule_pointer = wschedule_pointer->next;
74             if(wschedule_pointer == NULL)
75             {
76                 wtime_offset += wglobal_period;
77                 wschedule_pointer = wschedule;
78             }
79         }
80         if(wschedule_pointer->start_time + wtime_offset <
81             rt_times.intr_time)
82         {
83             rt_times.intr_time =
84                 wschedule_pointer->start_time + wtime_offset;
85             preempt = 1;
86         }
87     }
88     //////////////////////////////////////
89
90     /*
91         if(wschedule_pointer == NULL)
92         {
93             if(w_time_offset < rt_times.intr_time)
94             {
95                 rt_times.intr_time = w_time_offset;
96                 preempt = 1;
97             }
98         }
99     */
100
101     if ((shot_fired = preempt)) {
102         delay = (int)(rt_times.intr_time

```

```

103         - (now = rdtsc())) - tuned.latency;
104     if (delay >= tuned.setup_time_TIMER_CPUNIT) {
105         delay = imuldiv(delay,
106             TIMER_FREQ,
107             tuned.cpu_freq);
108     } else {
109         delay = tuned.setup_time_TIMER_UNIT;
110         rt_times.intr_time =
111             now + (tuned.setup_time_TIMER_CPUNIT);
112     }
113     outb(delay & 0xFF, 0x40);
114     outb(delay >> 8, 0x40);
115 }
116 } else {
117     rt_times.intr_time += rt_times.periodic_tick;
118 }
119
120 if (new_task != rt_current) {
121     if (rt_current == &rt_linux_task) {
122         rt_switch_to_real_time(0);
123         save_cr0_and_clts(linux_cr0);
124     }
125     if (new_task->uses_fpu) {
126         if (new_task != fpu_task) {
127             save_fpenv(fpu_task->fpu_reg);
128             fpu_task = new_task;
129             restore_fpenv(fpu_task->fpu_reg);
130         }
131     }
132     rt_switch_to(new_task);
133     if (rt_current->signal) {
134         (*rt_current->signal)();
135     }
136 }
137 }

```

**Figure 7 - Modifications made to the `rt_timer_handler` function**

This figure illustrates how the function `rt_timer_handler` was modified for the schedule-dispatcher. Lines appearing in a boldface font were added by the author. Lines appearing in a grey coloured font appeared in the original version of this function, and were commented-out by the author.

---

The modifications made to the `rt_timer_handler` function mirror the changes made to the `rt_schedule` function presented above. Lines 7 – 18 handle the initialisation case (the first period) and lines 66 – 88 start the next segment if its start time has already elapsed, and calculate the delay time to wait for the following the segment. Lines 101 – 136 are from the original `rt_timer_handler` function; they do the work

of setting the timer, and performing the task switch.

### 2.2.3 Modifications made to the `rt_task_delete` function

As already mentioned, the “single-list” uniprocessor scheduler provided with RTAI was used as a basis for the implementation of the scheduler-dispatcher. A minor conflict arose as a result of adding the schedule linked-list data structure to the RTAI scheduler. When a task is deleted (removed from RTAI) any corresponding entries in the schedule linked-list must also be removed. If these entries are not removed, then it is possible for the scheduler-dispatcher to attempt to dereference a pointer (from a `task` member in a node of the schedule linked-list) to a nonexistent task structure, resulting in a run-time error. This problem has been avoided by slightly modifying the `rt_task_delete` function, so that it checks the schedule linked-list, and removes entries for tasks being deleted.

There is a beneficial side-effect of these modifications. When RTAI shuts-down, the `rt_task_delete` function is called once for every real-time task. By removing the corresponding schedule linked-list nodes we avoid causing a “memory-leak” by not freeing the memory used for the schedule linked-list.

Figure 8 lists the changes made to the `rt_task_delete` function. Lines appearing in a boldface font (23 – 52) were added by the author. Lines appearing in a grey coloured font (lines 12 – 21) were commented-out by the author.

---

```
1     int rt_task_delete(RT_TASK *task)
2     {
3         unsigned long flags;
4         QUEUE *q;
5
6         if (task->magic != RT_TASK_MAGIC) {
7             return -EINVAL;
8         }
9
10        hard_save_flags_and_cli(flags);
11
12        #if 0
13        if (task->state & SEMAPHORE) {
14            (task->blocked_on.sem)->count++;
15            (task->queue.prev)->next = task->queue.next;
16            (task->queue.next)->prev = task->queue.prev;
17        } else if (task->state & (SEND | RPC)) {
18            (task->queue.prev)->next = task->queue.next;
19            (task->queue.next)->prev = task->queue.prev;
20        } else {
21            #endif
22
```

```

23 //////////////////////////////////////////////////
24 {
25     // Should remove task from schedule list here...
26     // this stuff could invalidate the wschedule_pointer
27     // pointer...
28     SCHEDULE* deleteme;
29
30     while(wschedule != NULL && wschedule->task == task)
31     {
32         deleteme = wschedule;
33         wschedule = wschedule->next;
34         sched_free(deleteme);
35     }
36
37     if(wschedule != NULL)
38     {
39         SCHEDULE* temp = wschedule;
40         while(temp->next != NULL)
41         {
42             if(temp->next->task == task)
43             {
44                 SCHEDULE* deleteme = temp->next;
45                 temp->next = temp->next->next;
46                 sched_free(deleteme);
47             }
48             else
49                 temp = temp->next;
50         }
51     }
52 //////////////////////////////////////////////////
53
54     q = &(task->msg_queue);
55     while ((q = q->next) != &(task->msg_queue)) {
56         (q->task)->state &= ~(SEND | RPC | DELAYED);
57     }
58     q = &(task->ret_queue);
59     while ((q = q->next) != &(task->ret_queue)) {
60         (q->task)->state &= ~(RETURN | DELAYED);
61     }
62 }
63 if (!((task->prev)->next = task->next)) {
64     rt_linux_task.prev = task->prev;
65 } else {
66     (task->next)->prev = task->prev;
67 }
68 if (fpu_task == task) {
69     fpu_task = &rt_linux_task;
70 }
71 mmsrq.mp[mmsrq.in] = task->stack_bottom;
72 mmsrq.in = (mmsrq.in + 1) & (MAX_SRQ - 1);
73 task->magic = 0;
74 rt_pend_linux_srq(mmsrq.srq);
75 if (task == rt_current) {
76     rt_schedule();
77 }
78 hard_restore_flags(flags);

```

```
79         return 0;
80     }
```

### Figure 8 - The `rt_task_set_schedule` function

The lines added to this function (23 – 52) search the schedule linked-list for any schedule segments referencing the task currently being deleted. If any are found, the segments (nodes) are removed from the schedule linked-list, and the nodes are freed.

The lines that have been commented-out of this function (12 – 21) remove the task from the semaphore and RPC lists, if necessary. To keep things simple, these lines were simply removed. Therefore, semaphores and RPC functionality is not available when the scheduler-dispatcher is used.

---

#### 2.2.4 Some new helper functions

As well as the modifications to the scheduler functions described above, the pre-run-time scheduler-dispatcher needs facilities to enter the schedule and start pre-run-time mode. Three new functions were added to support these new requirements:

```
void rt_set_preruntime_mode(void);
```

This function enables pre-run-time mode. This function must be called before any of the other functions listed below. Additionally, this function must be called before any of the time functions of RTAI.

```
int rt_task_set_schedule(RT_TASK *task, RTIME start_time, RTIME period);
```

This function is used to enter one segment of a pre-run-time schedule. The `rt_set_preruntime_mode` function must have been called before this function. The two time parameters, `start_time`, and `period`, are measured in nanoseconds.

The schedule segments must be entered in the order that they are to be executed (i.e. it is up to the user to call this function for each schedule segment, in the order of the `start_time` values).

```
void rt_start_preruntime_mode(void);
```

This function starts pre-run-time mode. This function is called after a schedule has been completely entered using the `rt_task_set_schedule` function. This function starts the pre-run-time scheduler-dispatcher.

The source code for these functions is presented below.

The `rt_set_preruntime_mode` function is the pre-run-time equivalent of the RTAI `rt_set_one-shot_mode` and `rt_set_periodic_mode` functions. These functions inform RTAI how the timer should behave. The source code for the `rt_set_preruntime_mode` function appears in Figure 9.

---

```
1 void rt_set_preruntime_mode(void)
2 {
3     rt_set_one-shot_mode();
4     start_rt_timer(0);
5 }
```

**Figure 9 - The `rt_set_preruntime_mode` function**

This function simply puts the RTAI system timer into one-shot mode.

---

The `rt_task_set_schedule` function appears in Figure 10. This function is used to enter a pre-run-time schedule, one segment at a time. This function also sets the period for the schedule. An example of usage of this function is provided below.

---

```
1 int rt_task_set_schedule(RT_TASK* task, RTIME start_time,
2 RTIME period)
3 {
4     SCHEDULE* s;
5     long flags;
6     RTIME now;
7
8     if (task->magic != RT_TASK_MAGIC) {
9         return -EINVAL;
10    }
11
12    if (!(s = (SCHEDULE *)sched_malloc(sizeof(SCHEDULE)))) {
13        return -ENOMEM;
14    }
15
16    s->next = NULL;
17    s->task = task;
18    s->start_time = nano2count(start_time);
19
20    hard_save_flags_and_cli(flags);
21    wglobal_period = nano2count(period);
22
23    if(wschedule == NULL)
```

```

24         wschedule = s;
25     else
26     {
27         wschedule_pointer = wschedule;
28         while(wschedule_pointer->next != NULL)
29             wschedule_pointer = wschedule_pointer->next;
30         wschedule_pointer->next = s;
31     }
32     wschedule_pointer = NULL;
33     hard_restore_flags(flags);
34
35     return 0;
36 }

```

**Figure 10 - The `rt_task_set_schedule` function**

This function begins (lines 8 – 14) by checking the validity of the `task*` pointer passed in the first argument, and by obtaining memory for the new linked-list node. Next the start time and period are converted from nanoseconds to RTAI clock ticks (lines 18, and 21). Finally, the new schedule node is inserted at the end of the schedule linked-list (lines 23 – 32).

The source code for the `rt_start_preruntime_mode` function appears in Figure 11.

```

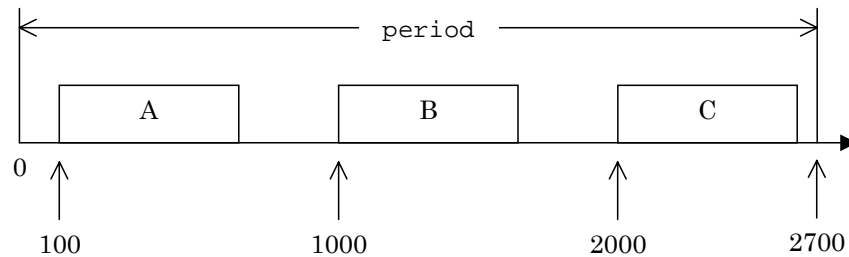
1     void rt_start_preruntime_mode(void)
2     {
3         unsigned long flags;
4
5         wschedule_pointer = NULL;
6         hard_save_flags_and_cli(flags);
7         rt_schedule();
8         hard_restore_flags(flags);
9     }

```

**Figure 11 - The `rt_start_preruntime_mode` function**

This function begins (line 5) by setting the schedule pointer `wschedule_pointer` to be `NULL`, indicating to the `rt_schedule` function that the scheduler should reset itself to the beginning of the schedule. Then the `rt_schedule` function is called. Control may not return from this function, depending upon the schedule.

A simple example is provided to illustrate how the new functions described above are used. Consider the schedule appearing in Figure 12.



**Figure 12 - An Example Pre-Run-Time Schedule**

Assuming that RTAI tasks have been constructed called `taskA`, `taskB`, and `taskC`, the source code listed in Figure 13 implements the schedule, and starts the schedule executing.

---

```
1  rt_set_preruntime_mode();
2  rt_task_set_schedule(taskA, 100, 2700);
3  rt_task_set_schedule(taskB, 1000, 2700);
4  rt_task_set_schedule(taskC, 2000, 2700);
5  rt_start_preruntime_mode();
```

**Figure 13 - Simple Example of the New Functions**

This source code implements the schedule listed in Figure 12.

---

### 2.2.5 Another new function: `rt_segment_done`

There is one final function that needs to be defined. When a segment has finished, it needs a way to communicate this to the scheduler-dispatcher. The `rt_segment_done` function is provided for this purpose.

When a segment is finished, the schedule should be consulted to determine the start time of the next segment. If the next segment does not start immediately, a timed delay may be necessary (during which time the Linux kernel may execute). At the appropriate time a task-switch should be performed to start the next segment. All of this functionality is provided by the `rt_schedule` function, described above. The

source code for `rt_segment_done` appears in Figure 14.

---

```
1 void rt_segment_done(void)
2 {
3     unsigned long flags;
4
5     hard_save_flags_and_cli(flags);
6     rt_schedule();
7     hard_restore_flags(flags);
8 }
```

**Figure 14 - The `rt_segment_done` function**

This function simply calls the scheduler-dispatcher.

---

## 2.2.6 Observations and Notes on the Schedule-Dispatcher

The schedule-dispatcher described in this section is robust. It allows processes to be scheduled, and preempted according to a given schedule. It is relatively easy to describe some process segments (in C) and to describe their schedule (for example, using code similar to Figure 13).

Both schedule-dispatchers have now been described. The following section describes an experiment that was performed to compare the timing accuracy of the two schedule-dispatchers.

## 3.0 Testing the two Schedule-Dispatchers

The two schedule-dispatchers described above were tested by running a slightly-modified version of one of the example real-time applications provided by RTAI. The example application used was the “Preempt Example” found in `rtai-{version}/examples/preempt`. This example demonstrates how RTAI can preempt a lower priority process with a higher priority one. Two processes are executed that communicate with a user-level application. The user-level program simply provides a means for the real-time tasks to convey their output to the user. The steps performed by the tasks are:

1. Begin by measuring the time in nanoseconds.
2. Send a message to the user-level program (a) indicating which task this is, (b)

that this task is starting, and (c) the timestamp measured in step one.

3. Delay for a specified number of nanoseconds.
4. Again measure the time in nanoseconds.
5. Send a second message to the user-level program (a) indicating which task this is, (b) that this task is terminating, and (c) with the timestamp measured in step four.

This provides a convenient framework for testing our implementations, and for capturing time-stamped data of the start and end times of the real-time processes. These examples were modified to run with the new schedule-dispatchers presented earlier. The source code for one of the process segments used by the replacement RTAI scheduler (the schedule-dispatcher described in section 2.2 above) appears in Figure 15.

---

```
1     static void ThreadA(int t)
2     {
3         static struct {
4             char task, susres;
5             unsigned long flags;
6             RTIME time;
7         } msg = {'A', };
8
9         while (1) {
10            msg.time = rt_get_time_ns();
11            msg.susres = 'r';
12            rt_global_save_flags(&msg.flags);
13            rtf_put(CMDF0, &msg, sizeof(msg));
14
15            // 20000 ns = 20 us
16            rt_busy_sleep(20000);
17
18            msg.time = rt_get_time_ns();
19            msg.susres = 's';
20            rt_global_save_flags(&msg.flags);
21            rtf_put(CMDF0, &msg, sizeof(msg));
22
23            rt_segment_done();
24        }
25    }
```

**Figure 15 - A Typical Process Segment Used For Testing**

Lines 3 – 7 define the structure used to pass data back to the user-level program that provides us with output. The members of the

structure are: `task` – containing a character identifying the process segment as “A”, `status` – containing a character indicating whether the process segment is starting (“r” for release) or terminating (“s” for stop), `flags` containing flags that are of no interest to us, and `time` to convey the time that the process started or finished, measured in nanoseconds.

When the process segment starts execution (at line 10) it queries the current time. Lines 11 – 13 send the “I’m starting” message to the user-program. Line 16 provides a timed delay to simulate the work that would normally be performed by this process segment. Lines 18 – 21 construct and send the timestamped “I’m done” message to the user-program. Finally, we terminate on line 23, by calling `rt_segment_done`.

The next time that the process segment is scheduled, it will continue execution from where it left off (after line 23). Because of the while loop on line 9, the same process is repeated with each invocation.

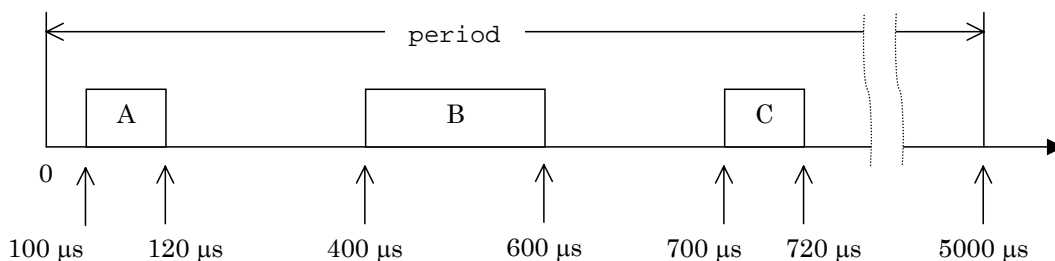
---

The source code for the test process segment used for the master-process schedule-dispatcher is nearly identical to the source code presented in Figure 15, except that the while loop (on line 9) is not necessary.

Note that using the source code in Figure 15, it is easy to create tasks of varying computation times. The value specified to the `rt_busy_sleep` function on line 16 specifies, in nanoseconds, how long the process segment should execute.

### 3.0.1 A Test Run

To test the schedulers, the schedule presented in Figure 16 was implemented using both schedule-dispatchers.



**Figure 16 - An Example Pre-Run-Time Schedule**

Assuming that RTAI tasks have been constructed (similar to Figure 15) called `taskA`, `taskB`, and `taskC`, the source code listed in Figure 17 implements and executes the schedule, using the replacement scheduler (from section 2.2). Note that, although not explicitly listed here, equivalent process segments, and an equivalent schedule implementation, were also created using the master-process schedule-dispatcher from section 2.1 above.

---

```
1     int init_module(void)
2     {
3         rtf_create_using_bh(CMDF0, 200000, 0);
4         if(rt_task_init(&TaskA, ThreadA, 0, 2000, 0, 0, 0))
5             return -1;
6         if(rt_task_init(&TaskB, ThreadB, 0, 2000, 0, 0, 0))
7             return -2;
8         if(rt_task_init(&TaskC, ThreadC, 0, 2000, 0, 0, 0))
9             return -3;
10
11        rt_set_preruntime_mode();
12
13        // period = 5,000,000 ns = 5 ms
14        // TaskA Starts at: 100000 ns = 100 us
15        // TaskB Starts at: 400000 ns = 400 us
16        // TaskC Starts at: 700000 ns = 700 us
17        rt_task_set_schedule(&TaskA, 100000, 5000000);
18        rt_task_set_schedule(&TaskB, 400000, 5000000);
19        rt_task_set_schedule(&TaskC, 700000, 5000000);
20
21        rt_start_preruntime_mode();
22
23        return 0;
24    }
25
26    void cleanup_module(void)
27    {
28        stop_rt_timer();
29        rt_busy_sleep(10000000); // 10 microseconds...
30        rtf_destroy(CMDF0);
31        rt_task_delete(&TaskA);
32        rt_task_delete(&TaskB);
33        rt_task_delete(&TaskC);
34    }
```

**Figure 17 - Implementation of the Test Schedule**

This source code implements the schedule listed in Figure 16.

---

Both schedule-dispatchers were tested by using process segments as described

above. Figure 18 presents the output from a sample run using the replacement scheduler (the schedule-dispatcher described in section 2.2 above).

```
1      A r 202,139,297
2      A s 202,174,175
3      B r 202,435,653
4      B s 202,643,154
5      C r 202,731,082
6      C s 202,756,598
7      A r 207,139,548
8      A s 207,168,292
9      B r 207,431,244
10     B s 207,635,187
11     C r 207,730,051
12     C s 207,754,484
13     A r 212,127,626
14     A s 212,151,528
15     B r 212,428,208
16     B s 212,632,007
17     C r 212,728,894
18     C s 212,752,814
```

### Figure 18 - A Sample Run with Replaced Schedule-Dispatcher

Each line in Figure 18 represents a timestamped event. Line 1 shows that process segment A started at time 202,139,297. Line 2 shows that A ended at time 202,174,175.

Three complete iterations of the schedule period are displayed in Figure 18, the first on lines 1 – 6, the next on lines 7 – 12, and the final on lines 13 – 18.

The timestamps given in Figure 18 indicate that the timing of the scheduled segments is not exact. Consider the time between periods (the time elapsed between lines 1 and 7) which is  $207,139,548 - 202,139,297 = 5,000,251$  instead of exactly five million nanoseconds as expected. According to Figure 16, 300,000 nanoseconds should elapse between the start time of processes A and B. Instead (lines 1 and 3)  $202,435,653 - 202,139,297 = 300,582$  nanoseconds elapse. These times are close to the expected values, but not exact. Evaluating how close these numbers are is the subject of the next section.<sup>1</sup>

---

<sup>1</sup> Even less accurate are the computation times. Process segment B should take 200 microseconds, but instead takes (line 4 minus 3 of Figure 18)  $202,643,154 - 202,435,653 = 207,501$  nanoseconds. However, consulting the RTAI source code reveals that the algorithm of the `rt_busy_sleep` function (that we are using to simulate the computation time of our process segments) is simply a busy wait that polls the system timer. This is not a particularly accurate way to measure time. However, we

## 4.0 Comparing the two Schedule-Dispatchers

Now that the schedule-dispatcher algorithms have been described, and suitable test programs have been found, we can consider the accuracy of the two schedule-dispatcher implementations.

### 4.0.1 Methods and Materials

A 166 MHz Pentium system had RedHat Linux 6.2 installed on it. The kernel was upgraded to version 2.2.19. The RTAI patches (version 1.7) were applied to the kernel.

Two copies of RTAI were installed. One copy was not modified at all, it is for the master-process schedule-dispatcher test. The second copy was modified, the regular uniprocessor scheduler was replaced with the schedule-dispatcher described in section 2.2 of this paper.

### 4.0.2 Procedure

Three real-time process segments were created, identical to those described in section 3.0 above. The schedule used was that appearing in Figure 16. For both cases, the real-time schedule was executed until 100 messages had been received from the process segments. (So 100 start and finish messages were received; this corresponds to approximately 16 iterations of the schedule period.)

The output data appeared identical in format to the listing in Figure 18. The data were cut-and-pasted into Microsoft Excel, for subsequent analysis.

### 4.0.3 Results

Having the data in Excel made it easy to calculate mean and variances of various times during the schedule. In order to evaluate how accurate the timing are, we shall consider three times:

1. The elapsed time from when the A process segment is started, until B is started. This should be approximately 300,000 nanoseconds (calculated by subtracting the start time of A from the start time of B:  $400,000 - 100,000 = 300,000$ ).

---

are not really concerned with the accuracy of the computation times – the start times are our concern.

2. Similarly, the elapsed time from when B is started, until C is started. This should be approximately 300,000 nanoseconds (calculated by subtracting the start time of B from the start time of C:  $700,000 - 400,000 = 300,000$ ).
3. Finally, the elapsed time from when C is started, until the next A is started (in the following period). This should be the start time of the next A segment minus the start time of the current C segment, or,  $5,000,000 + 100,000 - 700,000 = 4,400,000$  nanoseconds.

Having the data file in Excel made it trivial to calculate these values, and to find average values. The results appear in Table 1.

**Table 1 - Average Times Between Segment Start Times**

<b>Time</b>	<b>Scheduler</b>	<b>Mean Time (nanoseconds)</b>	<b>Variance (nanoseconds)</b>
C to A	Expected Time	4,400,000.00	-
	Master Process	5,102,816.60	8,664,413.83
	Replaced Scheduler	4,401,200.53	5,669,138.27
A to B	Expected Time	300,000.00	-
	Master Process	299,648.25	1,953,654.33
	Replaced Scheduler	299,071.25	8,228,778.87
B to C	Expected Time	300,000.00	-
	Master Process	299,557.94	2,853,218.73
	Replaced Scheduler	298,999.81	1,841,390.70

The terminology “C to A” means the time between the start of process segment C, and the start of the following process segment A.

The term “Master Process” refers to the master-process schedule-dispatcher described in section 2.1 of this paper. The term “Replaced Scheduler” refers to the dispatcher described in section 2.2.

The figures presented in this table represent the average of the measured elapsed times. Fifteen times were averaged for the “C to A” figure, and sixteen times were averaged for the “A to B”, and “B to C” times.

#### 4.0.4 Discussion

The average values appearing in Table 1 are promising. In general, both schedule-dispatchers seem to be reasonably accurate. Clearly the worst behaviour comes for the “C to A” case with the master-process schedule-dispatcher. The problem here must be due to inaccurate timing from one period to the next. For this, the `rt_task_wait_period` function is used (see Figure 3 line 25). Perhaps this function is not very accurate.

### 5.0 Conclusions

Two schedule-dispatchers were implemented, and compared. The first was the master-process schedule-dispatcher. It was found to be easy to implement, reasonably accurate, but unable to perform preemption. The second schedule-dispatcher was written as a replacement to the standard RTAI scheduler. This approach provided more accurate timing, and was able to perform preemption of segment processes.

### References

Xu, J. and Parnas, DL. (1990) Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3), pp. 360 – 369, March 1990.