PRACTICAL REAL-TIME SCHEDULERS

Robert Ward
The C Users Journal
Lawrence, KS

Robert Ward is the author of *A Programmer's Introduction to Debugging C* and president of R&D Publications Inc., which publishes *The C Users Journal, Tech Specialist*, and *Unique*. He is president of the C Users Group and holds an M.S. in computer science from the University of Kansas.
Practical Real-Time Schedulers

Robert Ward
Publisher, The C Users Journal and TECH Specialist

WHAT IS REAL TIME?

Real-time is not a synonym for "real-fast". Contrary to popular opinion, making everything "real fast" won't necessarily make a real-time program work correctly. A much better synonym is "on time" since, in a real-time program, certain events must happen at a specific time.

Making input and output events happen "on time" is pretty straightforward if you have only one I/O path to worry about. But real-time programs, especially embedded real-time systems, are often also multi-tasking programs. Most real-world, real-time programs are expected to simulate several pieces of simultaneously operating hardware.

When analyzing a project that is both multi-tasking and real-time, the designer must recognize that some tasks are less urgent than others. For each real-time task, "on time" may have a different meaning, depending upon the time constraints associated with that particular task. A continuously running built-in self-test, for example, usually runs without any time constraints, even if it is testing a real-time system. Some real-time events may need to happen at a specific wall-time; others at a specific interval from some external event. The real-time program must properly balance these varying needs at every instant, under every imaginable combination of input conditions.

This article will show how an appropriate general purpose scheduler can significantly reduce the design complexity in such programs and also significantly increase your confidence in the feasibility of the design even before you write any significant amount of code.

WHAT IS A SCHEDULER?

A scheduler is simply code that decides which task to perform next. Thus a scheduler can be as simple as the loop in Listing 1. This "slop-cyclic" scheduler cycles repeatedly through each task (cyclic) at a rate that may vary depending upon the time required to execute each task (slop). A "rate-cyclic" scheduler can be almost as simple, as shown in Listing 2. The rate cyclic scheduler cycles through all the tasks at a constant rate of once per clock tick.
If you are accustomed to writing real-time systems as one large loop with input polling and capture code sprinkled throughout the system, it may seem pretentious to describe Listing 1 as a "scheduler." After all, one could argue, the execution sequence is, like the polling loop, just a hard-wired loop — the so-called scheduler just adds calling overhead.

Even this trivial scheduler, though, offers several important advantages. First, all the scheduling information is in one place. If you must alter the code (and the timing relationships between the pieces), you know just where to look to make the necessary scheduling adjustments. For example, if after writing the project, you found that task3() and task4() didn’t execute as rapidly as expected, causing task1() to “miss events”, you might solve the problem by making the change in Listing 3. Now task1() gets scheduled more than once during the cycle. You can even drop one task into the “middle” of another by breaking one task into several sub-parts (see task2a() and task2b() in Listing 4).

With the scheduling code isolated in a single module, the designer also reserves the option to completely change the scheduling mechanism. For example, instead of splitting task2 into two parts (as in Listing 4), you might obtain the same effect by installing a more sophisticated pre-emptive scheduler like the one I’ll develop later in this article.

A distinctly separate scheduler can also make development and testing much easier. You might plan to use a simple loop like Listing 1 to perform initial testing and characterization of your task code and then install a more sophisticated scheduler for final test and production.

PRIORITY AND PRE-EMPTION

In the trivial schedulers of Listings 1-4, all tasks are of equal importance and the order of task execution is statically determined by how the code is written. This egalitarian approach forces the programmer to adjust for differences among the tasks by adjusting the code, for example by making multiple calls to task1() and by splitting task2() into two parts. If the scheduler were more competent, we wouldn’t need to make these coding compromises. What we need is a scheduler that can recognize that some tasks (task1() for example) are more important than others (have higher priority) and that sometimes long tasks like task2() may need to be interrupted (be pre-empted) so that some shorter, more important task can run “on-time”. The scheduler should not only recognize these differences, it should be able to dynamically adjust the execution order to accommodate them.

Prioritized scheduling can be accomplished by augmenting the simple slop cyclic scheduler so that it uses a different control structure driven by several “ready” lists. Listing 5 presents the basic structure for a very simple environment where each task is assigned to a different priority level and each ready list consists of a separate flag in the structure ready.

The ready flags are set by an interrupt service routine that captures related input, or by some other task (for example, task2a() would set task2b()’s flag, thereby “scheduling” task2b()).
Listing 5 will schedule events dynamically based on their readiness, but it still lets each task run to completion. The next level of scheduler sophistication, pre-emption, complicates matters considerably, but is still easy to implement once the calling conventions are understood.

The pre-emptive scheduler pre-supposes an environment where virtually all events are serviced by interrupts. These interrupts create natural "break points" at which other tasks can be pre-empted. (Actually, you can get the same effect by sprinkling special calls, gotos and stack manipulations throughout each task, but you won't want to maintain the result.) Each interrupt service routine ends with a call to the scheduler. The scheduler then examines all higher priority ready lists to see if some more urgent task needs to run. If not, the scheduler simply returns, allowing the interrupted task to continue. If there is a more urgent task waiting, the scheduler calls it (see Listing 6).

Listing 6 assumes several conventions and data structures not explicitly shown. The functions push() and pop() manipulate a stack of current priority levels. The functions acquire_queue() and release_queue() are used to enforce mutual exclusion on all of the shared resources supporting the scheduler. In this example, acquire_queue() would perform a busy_wait.

The function highest_pri() searches a ready list for actions more urgent than the action just interrupted. If a more urgent task is found, get_next() copies its descriptor from the ready list to the scheduler's local "next" structure. The task is then invoked via a pointer to function.

Before the more urgent task is executed, its work request is deleted from the ready list and the ready list is "closed up" by moving all later requests forward. A dynamically allocated linked list would be more efficient, but this implementation makes for a simpler presentation, and still preserves the FIFO behavior of the work request queue.

Separate queues for each priority level would also improve performance, especially the response time for high priority tasks.

The scheduler must always be called with interrupts enabled — otherwise it will provide only one level of pre-emption and may cause some interrupts to be missed.

This implementation is very stack-intensive. Each interrupt can potentially generate three stack frames for each interrupt (interrupt, interrupt's call to scheduler, scheduler's call to task). In an environment where many interrupts can arrive simultaneously, the stack may expand very rapidly.

SOME DESIGN ADVICE

A real-time design should begin with a careful analysis of the possible events and the relationship among them. The goal is to decouple (with respect to time) as many actions as possible. Decoupling will often greatly increase your ability to service critical events, by allowing the great bulk of the processing to occur in the time...
between critical events. This analysis should identify the time-critical events (those that really must be done NOW), and prioritize the other events according to their relative criticality. Generally, actions subject to similar constraints should be made members of a priority class and broken into execution units that are small relative to the time-tolerance of the next, more urgent class.

Early in the design analysis, you should compute the probable CPU utilization. If the tasks assigned to the system will consume more than 70 percent of the CPU throughput, you should probably consider the design impractical and either find faster algorithms for certain modules, add additional hardware, or run on a faster CPU. In truly asynchronous environments, a processor utilization of greater than 70 percent greatly increases the likelihood that one of your ready queues will grow to an unmanageable length. You can make an exception to the 70 percent rule if you can prove that your waiting lists will never grow beyond some small fixed length.

The structure of the program will mirror the classification of events. Critical events will be serviced by interrupt handlers, non-critical events will be processed according to their priority by a general-purpose scheduler, and queues will handle communication among the pieces. Since these systems are almost always concurrent, it is imperative that the programmer be comfortable with the issues of deadlock avoidance and shared resource management.

With careful analysis of events and adequate throughput, a simple cyclic scheduler is often adequate. In some applications where some actions consume very large amounts of time compared to the time-tolerance of higher-priority tasks, it may be necessary to implement a pre-emptive scheduler.

A CASE STUDY

Suppose you are to build a real-time system with four major functions:

Process Control. This function consists of monitoring a sensor on a production line and adjusting a control output to keep the process within acceptable limits. The sensor is to be sampled every 100 ms (±1 ms) and necessary output corrections must be made within 100 ms. Statistical quality control methods are to be used to decide if the sample input represents an unacceptable deviation. Test programs have shown these calculations to require 7 ms. A programmable internal timer is to be used to control the sampling interval. The timer can generate interrupts.

Manual Override. The system should accept human specifications for the control output from a keyboard. This keyboard debounces inputs, but once the character is validated, it must be read within 70 µs. The keyboard’s “character ready” status line is connected to the CPU’s interrupt line.

Time-of-day Clock. An hour and minute time-of-day display. Presumably it will take its timing from the internal timer. The display is mechanical, each digit on a “flip board”, driven by a stepper motor. The motor must be pulsed through 60 steps to change a digit. Each step takes 10 ms, to complete. Pulses are directed to
the appropriate digit position by a multiplexor, thus the position must be selected and then the pulses sent to change a digit.

Synchronous Communications Support. The system functions as a “repeater” in a communications network. Supporting hardware captures data a block at a time and requests your system to perform a crc-16 on the data. If the data is correct, your system must initiate a write 75 ms (± 100 μs) after the block was marked received. A failure to meet this requirement will result in the subordinate hardware missing a polling cycle and loss of the block. The subordinate hardware’s “block ready” shares the interrupt line with the keyboard. The block ready signal remains set until reset by the CPU.

Table 1 summarizes these specifications and adds estimates for each task’s execution time. These execution time estimates can be based on expected code size for each task, on prior experience with similar problems, on padded measurements of execution speeds for certain critical inner loops, or on measurements taken from “prototype” implementations for each task (perhaps written in a high-level language). Since adding a scheduler to the design makes each task a piece of stand-alone code, time spent coding each task for these measurements isn’t just wasted. Most of your characterization code can be used in the finished design.

CRITICAL OPERATIONS

Capturing a keystroke and (because it shares an interrupt with the keyboard) capturing a block ready indication are the only critical tasks in the system. These will be processed by an interrupt handler with interrupts turned off throughout the service.

PRIORITIES

Level 0. Capturing a data sample, capturing a clock tick and initiating a block write are “nearly” critical. It also makes sense for all three to be handled in the same interrupt service routine. Since they have lower priority than the critical events, interrupts will be enabled during as much of the service routine as possible. Thus the data sampling interrupt routine could be interrupted during its execution. We’ll assume that the first 15 μs of this process can’t be interrupted. Note that this priority level isn’t recognized in the scheduler because it is fully processed in the interrupt handler — I just wanted to show that even interrupt handlers differ in their urgency.

Level 1. The block checksum and sample analysis will be grouped at the next level of priority. The checksum has been broken into several short parts so that it can’t “block out” the input analysis for more than a few milliseconds. Each part will schedule its successor after it completes. This ensures the sample analysis will be able to “sneak” in between two parts (the sample analysis is scheduled by an interrupt routine, possibly while the checksum is executing).

Level 2. All clock control and keystroke parsing will be performed at level 2 (or background) activities. Notice that even though none of these events consumes more than 10 ms, if three such events were allowed to be interspersed with level 1 events, the block check would miss its “output deadline.”
Level 3. This level is reserved for pure "waiting" activities, such as waiting for the clock stepping motor to time-out.

Listing 7 presents the structure of the entire application in a C-like pseudo code. This code would use the scheduler of Listing 6.

THROUGHPUT REQUIREMENTS

Processor utilization is computed by combining the frequency estimates and time consumed estimates from Table 1. Table 2 shows that this design falls well within the 70 percent rule, and should probably be feasible.

Total utilization isn't the only prerequisite to feasibility, however. The design must also meet the response time restrictions. Response time performance is evaluated by calculating the worst time performance for each event. Worst case analysis should always include the possibility that the program has just responded to some interrupt or that multiple copies of the analyzed interrupt arrive at the closest interval possible. Table 3 analyzes the design's latency when performing a process control cycle.

ADDITIONAL IDEAS

When a variety of events happen at non-harmonic intervals, consider implementing a timer scheduling queue. Events can specify the timing of other events by putting a timer programming request in a special queue.

If your system has multiple interrupting events and no vectored interrupts, restrict the interrupt handler to just capturing the interrupt and queueing it. The highest priority task then examines the information in this queue and schedules other work.

To make certain two tasks of equal priority get fair scheduling, partition them into pieces (as with the checksum above) and let each piece upon completion schedule its successor. This scheme will allow the shorter tasks to be scheduled. This trick can often eliminate the need for a pre-emptive scheduler.

CONCLUSION

An appropriate scheduler can greatly simplify real-time designs by allowing the individual task modules to remain ignorant of their interaction with other real-time tasks. A distinct scheduler also simplifies debugging and performance analysis. If you aren't comfortable with the concurrency issues implicit with handling the ready queues and other shared resources in the dynamic schedulers, you can still use the static versions and preserve the option of incorporating a more complex scheduler when the project eventually demands it.

Little schedulers like those developed here are usually all the real-time support a controller needs. They offer distinct advantages over a commercial real-time kernel: the scheduler is smaller, simpler to understand, comes complete with source code, and is much less expensive.
Listing 1

while (FOREVER) {
    task1();
    task2();
    task3();
    task4();
}

Listing 2

while (FOREVER) {
    /* sleep until awakened by clock interrupt */
    sleep(clock_tick);
    task1();
    task2();
    task3();
    task4();
}

Listing 3

while (FOREVER) {
    task1();
    task2();
    task1();
    task1();
    task3();
    task1();
    task4();
}

Listing 4

while (FOREVER) {
    task1();
    task2a();
    task1();
    task2b();
    task1();
    task3();
    task1();
    task4();
}
Listing 5

while (FOREVER) {
    if (ready.level1) {
        ready.level1 = 0;
        task1();
    } else if (ready.level2a) {
        ready.level2a = 0;
        task2a();
    } else if (ready.level2b) {
        ready.level2b = 0;
        task2a();
    } else if (ready.level3) {
        ready.level3 = 0;
        task3();
    } else if (ready.level4) {
        ready.level4 = 0;
        task4();
    }
}

Listing 6

This scheduler treats LEVEL1 as the highest level of priority.

#define EMPTY -1
#define LEVEL1 = 3
#define LEVEL2 = 2
#define LEVEL3 = 1

struct actions { /* stores a list of ready actions */
    int priority;
    void (*action)();
    char * arg;
} ready[MAX_WAIT];

int clevel;
int queue_end; /* the index of the next free action */
int next_index;
int highest_pri()
{
    int highest;
if (queue_end == 0) return EMPTY;

highest = EMPTY;
for (i=0; i < queue_end; i++) {
    if (ready[i].priority > highest) {
        highest = ready[i].priority;
        next_index = i;
    }
}
return highest;

/*
this function must not be called unless the queue has been
allocated for it's use */

void get_next(struct actions next) {
    int i;

    copy_action(next, ready[next_index]);
    queue_end -= 1;
    for (i=next_index; i<queue_end; i++)
        copy_action(ready[i], ready[i+1]);
}

main () {

    /* initiate interrupt handlers */
    while (TRUE) {
        clevel = EMPTY;
        scheduler();
    }
}

void scheduler() {
    struct actions next;

    acquire_queue();
    if (highest_pri() > clevel) {
        get_next(next);
        push(clevel);
        clevel == next.priority;
        release_queue();
        (*next.action)(next.arg);
        acquire_queue();
        clevel=pop();
    }
release_queue();
}
/* general outline of interrupt handler */

entry point:
  save critical data
  enable interrupts if possible
  acquire queue
  schedule dependant actions
  release queue
  adjust stack
    adjust actual return frame
    create return frame that will vector to scheduler
  return from interrupt

Listing 7

Psuedo code for interrupt handler:

On keyboard interrupt do {
  save context
  if keyboard has input, save level2 work request to queue
  if block is ready {
    reset indicator
    add level1 check_1st_blk request to queue
  }
  restore context
  enable interrupts
  return from interrupt
}

On timer interrupt do {
  save context
  if write ok flag set{ /*done with interrupts off to avoid clash */
    initiate write
    clear flag
  }
  enable interrupts
  capture sample, save with level1 work request to queue
  step minute counter, on overflow {
    reset counter
    put level2 minute change work request in queue
  }

return from interrupt

Pseudo code for tasks

Analyze sample {
    perform statistical analysis
    if out of bounds {
        compute correction
        output correction
    }
    return
}

Check Block Pt 1{
    compute partial checksum
    save result with level1 pt2 work request in queue
    return
}

Check Block Pt 2{
    continue checksum
    save result with level1 pt3 work request in queue
    return
}

Check Block pt3 {
    continue checksum
    save result with level1 pt4 work request in queue
    return
}

Check Block pt4 {
    complete checksum
    if ok, set write ok flag
    return
}

Parse input {
    save input parameter in command line buffer.
    If input keystroke is a terminal symbol{
        parse buffer;
        output manual correction;
        clear buffer;
    }
    return;
}

Minute change{
    increment minutes-ones
add level3 work request for digit change to minutes-ones to queue request digit step;
on overflow {
    add level3 work request for digit change to minutes-tens to queue request digit step;
}
on tens overflow {
    add level3 work request for digit change to hours-ones to queue request digit step;
}
on hours ones overflow {
    add level3 work request for digit change to hours-tens to queue request digit step;
}
on hours-twelve overflow {
    add level3 work request for digit change to hours-ones to queue
    for (i=0; i<8; i++)
        request digit step;
    add level3 work request for digit change to hours-tens to queue request digit step;
}
}
digit change {
    set multiplexor to select requested digit
}
request digit step {
    for (i=1; i<60; i++) {
        add level3 work request for one_pulse to queue
    }
}

one_pulse {
pulse stepping motor
busy-wait for 5 ms
release pulse
busy-wait for 5 ms
return
}
**Table 1**

<table>
<thead>
<tr>
<th>Events</th>
<th>Trigger</th>
<th>time consumed</th>
<th>latitude</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>capture sample</td>
<td>timer interrupt</td>
<td>70 us</td>
<td></td>
<td>10/s</td>
</tr>
<tr>
<td>analyze sample</td>
<td>input available</td>
<td>7 ms</td>
<td>-0, +92 ms</td>
<td>10/s</td>
</tr>
<tr>
<td>output correction</td>
<td>analysis complete</td>
<td>35 us</td>
<td>n/a</td>
<td>10/s</td>
</tr>
<tr>
<td>capture keystroke</td>
<td>G.P. interrupt</td>
<td>35 us</td>
<td>-0, +35us</td>
<td>5/s</td>
</tr>
<tr>
<td>parse input</td>
<td>keystroke stored</td>
<td>1 ms</td>
<td>500 ms</td>
<td>1/s</td>
</tr>
<tr>
<td>clock tick</td>
<td>timer interrupt</td>
<td>15 us</td>
<td>+-500 us</td>
<td>10/s</td>
</tr>
<tr>
<td>minute change 1/60s</td>
<td>clock tick</td>
<td>150 us</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>digit change 4/60s</td>
<td>minute change</td>
<td>50 us</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>digit step 320/60s</td>
<td>digit change</td>
<td>10 ms</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>capture block ready</td>
<td>G.P. interrupt</td>
<td>35 us</td>
<td>-0, +35us</td>
<td>7/s</td>
</tr>
<tr>
<td>check 1st q</td>
<td>capture block</td>
<td>13 ms</td>
<td>+- 3 ms</td>
<td>7/s</td>
</tr>
<tr>
<td>check 2nd q</td>
<td>1st q checked</td>
<td>13 ms</td>
<td>+- 3 ms</td>
<td>7/s</td>
</tr>
<tr>
<td>check 3rd q</td>
<td>2nd q checked</td>
<td>13 ms</td>
<td>+- 3 ms</td>
<td>7/s</td>
</tr>
<tr>
<td>check 4th q</td>
<td>3rd q checked</td>
<td>13 ms</td>
<td>+- 3 ms</td>
<td>7/s</td>
</tr>
<tr>
<td>initiate block write</td>
<td>4th q checked</td>
<td>35 us</td>
<td>+- 100us</td>
<td>7/s</td>
</tr>
</tbody>
</table>
# Table II

<table>
<thead>
<tr>
<th>Events</th>
<th>factors</th>
<th>time used</th>
<th>capacity used</th>
</tr>
</thead>
<tbody>
<tr>
<td>capture sample</td>
<td>70 us * 10/s</td>
<td>700us/s</td>
<td>.0007</td>
</tr>
<tr>
<td>analyze sample</td>
<td>7 ms * 10/s</td>
<td>70 ms/s</td>
<td>.07</td>
</tr>
<tr>
<td>output correction</td>
<td>35 us * 10/s</td>
<td>350us/s</td>
<td>.000350</td>
</tr>
<tr>
<td>capture keystroke</td>
<td>35 us * 5/s</td>
<td>175us/s</td>
<td>.000175</td>
</tr>
<tr>
<td>parse input</td>
<td></td>
<td>1ms/s</td>
<td>.001</td>
</tr>
<tr>
<td>clock tick</td>
<td>15 us * 10/s</td>
<td>150us/s</td>
<td>.000150</td>
</tr>
<tr>
<td>minute change</td>
<td>150us * 1/60s</td>
<td>150us/60s</td>
<td>.000003</td>
</tr>
<tr>
<td>digit change</td>
<td>50 us * 4/60s</td>
<td>200us/60s</td>
<td>.000004</td>
</tr>
<tr>
<td>digit step</td>
<td>10 ms * 320/60s</td>
<td>53.3ms/s</td>
<td>.053300</td>
</tr>
<tr>
<td>capture block ready</td>
<td>35us * 7/s</td>
<td>245us/s</td>
<td>.000245</td>
</tr>
<tr>
<td>check 1st q</td>
<td>13ms * 7/s</td>
<td>91ms/s</td>
<td>.091000</td>
</tr>
<tr>
<td>check 2nd q</td>
<td>13ms * 7/s</td>
<td>91ms/s</td>
<td>.091000</td>
</tr>
<tr>
<td>check 3rd q</td>
<td>13ms * 7/s</td>
<td>91ms/s</td>
<td>.091000</td>
</tr>
<tr>
<td>check 4th q</td>
<td>13ms * 7/s</td>
<td>91ms/s</td>
<td>.091000</td>
</tr>
<tr>
<td>initiate block write</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total Utilization .489927
<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start checksum</td>
<td>0</td>
</tr>
<tr>
<td>Receive GP interrupt</td>
<td>35 us</td>
</tr>
<tr>
<td>Receiver Timer interrupt</td>
<td>15 us</td>
</tr>
<tr>
<td>Receive second GP intr.</td>
<td>35 us</td>
</tr>
<tr>
<td>complete timer intr</td>
<td>55 us</td>
</tr>
<tr>
<td>complete checksum part</td>
<td>13 ms</td>
</tr>
<tr>
<td>Perform analysis</td>
<td>7 ms</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>20.14 ms</strong></td>
</tr>
</tbody>
</table>