

A Compact Task Dispatcher for Embedded Systems

Ron Kreyborgⁱ

Many small embedded systems have severe restrictions on program memory. This article describes a small but powerful task dispatcher aimed specifically at resource starved systems.

Introduction

I have been building embedded system controllers for many years, with most systems being a collection of actuators or indicators controlled by a variety of events. The actuators might be motors or relays or lights or solenoids, while the events might be switch closures or a voltage level or a certain frequency. In all cases there was a small executive program that was common to all of them and whose responsibility it was to control both when the switch or other input was looked at, in what order, and what had to be done. This was the task dispatcher.

While the old micros like the Z80 and 8080 required external memory and IO chips, micros like the AVR series from Atmel, the PIC series from Microchip, and the latest H8 and H16 series from Motorola combine RAM, ROM, timers and IO all in the one chip. This makes for physically small but very powerful systems. The small size usually also applies to resources like code and ram space and therefore requires very efficient code. The dispatcher described here, while presented in C, is small enough to be reliably implemented in the assembly language of the destination micro.

In the Beginning

Except for a few systems where *real-time* is absolutely critical, most embedded system programs have a very similar structure. An interrupt driven clock ticks away every few milliseconds and one or more tasks run at every clock tick. These tasks examine internal flags or I/O ports and make a decision based on their state. A change of *input* state can cause a consequent change of state for some *output* port, starting some external event such as a motor running or a light turning on. In most cases the task takes only a small fraction of the time between clock ticks. The input state the task is looking for changes once the output action has occurred. In the light example the control task will now be looking for events that turn the light off.

Thus if you look at each task there must be code to decide what mode it is in, what state or states are involved, and

what output action is appropriate. Without some form of task dispatcher, this type of function usually consists of lots of if-then-else clauses. In effect the task must determine everything about itself every time it runs. This raises the first point about this type of code structure:

- Too many complex if-then-else sequences as each task decides what mode it is in, what it should be looking for, and what needs to be done.

In many cases there is also a requirement for external events to be timed in some way. In a motor example, suppose a new request commands run the motor clockwise when it had earlier been commanded to run anti-clockwise. The sequence of events here is different to a simple stop request. The motor must be stopped, allowed to come to rest, and then restarted in the new direction. In the absence of tachometer output this would be implemented by turning the motor off and starting a pre-computed delay. When the delay timed out the motor would be restarted in the opposite direction.

Another simple example could be a flashing alarm light. When the alarm event happens the light is turned on and a delay started. When this delay times out the light is turned off and the first delay re-started. When that one times out the light is turned back on, and so on regularly flashing the light until the alarm event goes away. The delays are typically implemented by counters in registers or RAM that are decremented at each clock tick. This raises the second point about this type of code:

- Too many timers that may be required to run, requiring considerable code and processor time to set up and manage each separate timer.

To simplify how I managed tasks and timers, what I needed was an executive that would allow one task to schedule another when the state of the device it was controlling had changed. Thus individual tasks would be considerably simplified in that they need only manage the event or events that take that particular device to the next state. It should also manage multiple software timers with as little overhead as possible. And finally it should be as small as possible, as code space is always limited.

What's a Dispatcher?

Many years ago I came across an Intel application note that described a very simple operating systemⁱⁱ. It showed an innovative way of managing multiple timers and a simple queuing technique for managing tasks. However it used an awkward system of numbered tasks so I rewrote it to use task pointers. Over the years I have converted it to various assembly languages and for the most part it has been the executive of choice for my embedded systems. Recently I re-wrote it in C as part of testing my SmallC for AVRⁱⁱⁱ and I will use that language throughout this article to describe its functionality.

It consists of two main parts, a ready task queue dispatcher manager and a delayed task queue manager. Both queues share the same data structure. The code is shown in Listing 1. The system is initialised by a single call to the `InitMulti` function. You add tasks to the tail of the ready queue by passing the name of the task entry point function to the `QueTask` function. Tasks are taken from the head of this queue by calls to the `Dispatch` function. A typical main program looks like

```
InitMulti();
QueTask(TaskA);
while (TRUE) {
    Dispatch();
}
```

The `QueTask` call puts `TaskA` on the ready queue. When `Dispatch` is called it will find `TaskA` on the ready queue and run it. It does not return until `TaskA` is finished. Once `TaskA` finishes it will move to the unlisted or idle state. However, a task can put itself back on the ready queue by either calling `QueTask` or the simpler `ReRunMe` just before it exits. Running tasks can (and often do) put other tasks on the ready queue as part of their job. From a resource allocation point of view, you are guaranteed that only one task will be running at any one time. In a complex controller the single initial `QueTask` call in the program above would be repeated for each peripheral being controlled.

Note we are not multi-tasking using time-slicing. Each task is run in the order it is put on the ready queue, and each task runs until it voluntarily exits. Interrupts can occur of course, but each task must return to allow other tasks to run.

In many embedded systems this level of task control is all that is required. Generally a task will execute in only a few dozen to a few hundred instructions, typically a few dozen microseconds. Many real world events occur in time scales much longer than this, in the order of tens of *milliseconds*. This allows complete dynamic control with micro clock ticks of from 1 through to 25 milliseconds.

Running Tasks

What has been described so far is itself quite useful. As an absolutely simple example, managing a single led would require two tasks, one looking at the event that turns the led on, and another that looks at turning it off.

```
TurnLedOn(){
    if (switchOn) {
        outp(PORTB,0x01);
        QueTask(TurnLedOff);
    }
    else
        ReRunMe(0);
}

TurnLedOff() {
    if (!switchOn) {
        outp(PORTB,0x00);
        QueTask(TurnLedOn);
    }
    else
        ReRunMe(0);
}
```

The `switchOn` variable is set elsewhere to match the on/off switch state. Because the led has only two states there are only two functions required. While the switch is off `TurnLedOn` continuously re-runs itself via the `ReRunMe` function. When the switch turns on the output port is set to turn the led on and the `TurnLedOff` task scheduled. Because `TurnLedOn` is not rescheduled it reverts to the idle state. At the next `Dispatch` call `TurnLedOff` will run, also re-running itself until the switch turns off.

While many real world devices have more than two states, the important point is that your code is now made up of functions that simply manage a single state, and need only worry about events that move it from that state to another.

With, say, six devices to control and little processing required per task, an AVR type micro could be running each task every few hundred microseconds. If current drawn is critical, an additional task could be included that calls the micro's *sleep* function. Thus the micro rips through the current set of tasks then powers down, perhaps for many milliseconds until the wakeup event occurs. The sleep task re-schedules itself on the tail of the ready queue and the cycle repeats.

Adding a Timer Capability

In nearly every controller I have ever built it was necessary to know about elapsed time. A light needed to be on for 200 milliseconds, a motor should be off for 2 seconds before being turned on again, an overload condition could last for only 10 seconds before some action was required. The list goes on.

The `QueDelay` function provides this facility. It is called in the same way as `QueTask`, but with an additional *delay-time* parameter. It does some neat re-arrangement of the other tasks in the delay queue (explained later) causing this task

to remain idle for now but come off the delay queue after *delay-time* ticks of the system clock have occurred. Once taken off the delay queue the task is automatically put on the ready queue.

Having a timer function means we can add a delay capability to the `ReRunMe` function. Now if a function is only required to run every 5 clock ticks, it can conclude with

```
ReRunMe(5);
```

The `ReRunMe` function examines the passed parameter and if zero, simply calls `QueTask`. However for a non-zero parameter it calls `QueDelay` instead.

Using `QueDelay` a task can make an event occur for a certain period. For example the task below demonstrates controlling a motor that is required to run for 2 seconds after the occurrence of a flag, where the clock tick occurs every 25 milliseconds.

```
CheckRunMotor() {
    if (flag) {
        StartMotor();
        QueDelay(StopMotor, 80);
    }
    else
        ReRunMe(0);
}
```

When the flag occurs the motor is started and eighty clock ticks later `StopMotor` turns it off and re-schedules `CheckRunMotor`.

As another example, take the earlier case where we simply turned a led on and off in response to some switch. With the timer function we can make the led flash in a set pattern.

```
TurnLedOn(){
    if (switchOn) {
        Led = 1;
        LedOn();
        QueTask(TurnLedOff);
    }
    else
        ReRunMe(1);
}

TurnLedOff() {
    if (!switchOn) {
        Led = 0;
        QueTask(TurnLedOn);
    }
    else
        ReRunMe(1);
}

LedOn() {
    outp(PORTB, 0x001);
    QueDelay(LedOff, 8);
}

LedOff() {
    outp(PORTB, 0x00);
    if (LED)
        QueDelay(LedOn, 20);
}
```

Using the same clock tick time of 25 milliseconds, the `LedOn` and `LedOff` functions will continue to cycle the led on for 200 milliseconds and off for 500 until the `TurnLedOff` function eventually detects the switch off command. You can see that as soon as the `Led` variable is set to zero, both the `LedOn` and `LedOff` functions automatically become idle at the end of the next `LedOff` period.

Note how the `ReRunMe` function called with a parameter of 1 means that the led on/off switch flag is only examined at every 25 millisecond clock tick.

The same technique can be used for tasks such as switch de-bouncing. In the following example we will assume the clock ticks every 10 milliseconds. The `input` function from port PIND reads the actual pin levels of port D.

```
int state;

void CheckSwitch(void) {
    int val;
    val = inp(PIND);
    if (val ^ state)
        QueDelay(CheckSwState, 3);
    else
        ReRunMe(1);
}

void CheckSwState(void) {
    int val;
    val = inp(PIND);
    if (val ^ state) {
        state = val;
        ChangedFlag = 1;
    }
    QueDelay(CheckSwitch, 3);
}
```

The `CheckSwitch` function compares all port D inputs with the static variable `state` every clock tick. If a change occurs (ie the exclusive-or is non-zero) it schedules the `CheckSwState` function to have another look three clock ticks or 30 milliseconds later. If the change was transient no state changes are made. However, if they are still different the `state` variable is updated with the new state and a flag set indicating to another task that a switch has changed state. In either case `CheckSwitch` is re-scheduled and the cycle continues. The bit states in the `state` variable define whether the switch was opened or closed.

While these are necessarily simple examples, I guess by now you are starting to think of how you can use this in your own applications. The important points to keep in mind are what does occur in moving from state to state and how each change can be implemented as a function. I usually sit down with pencil and paper and draw a simple state diagram that describes each state and how the system moves between them.

The Queue Data Structure

The dispatcher uses a single data structure in which it dynamically builds both the ready and the delay queues.

The *task* data structure has four fields per entry and looks like

```
struct {
    char    status;
    int     delay;
    void    (*tpntr)();
    char    next;
} task[TOTALTASKS+1];
```

Three bits are used in the *status* byte. If the task is on the ready queue, the *ready* bit will be set. If the task is on the delay queue then the *delay* bit will be set, with the *delay* field containing the number of ticks required for the delay. If the task is on either queue the *busy* bit will be set. The function pointer field *tpntr* points to the task code and the *next* field points to the next entry in the queue. The *task[0]* entry is not used so zero can be used as an empty flag for the queue pointers. However the *task[0].next* entry is actually the delay queue head and is used to simplify the code in the *DoQueueDelay* function.

The ready and delay queues both have head pointers which contain the list entry number for the first task on each queue. The *next* field for that entry points to the next entry in the queue. The last entry will have a zero in this field. For the ready queue there is also a tail pointer pointing to the last entry. Tasks are added to the bottom or tail entry of this queue, and taken from the top or head of the queue.

How the Ready Queue Works

When you pass an address to *QueueTask* it searches the task structure for an empty entry. If there is no room left in the list it returns a status of zero. Otherwise it performs the simple linked queue operation of adding the new task to the end of the ready queue and inserting the task's details in the blank entry (note there is no relationship between the order of task entries in the task structure and their order in the queues). If this new task is the only task it will also be the head of the queue. Figure 1 shows the data structure after a single task has been added to the ready queue. The status entry and returned value for this task will be set *busy* and *ready*. Figure 2 shows what happens when another task is added. Note you can navigate from the *TaskHead* pointer through the queue to the last entry via the *next* fields.

Calling *QueueTask* doesn't actually make anything happen, it just builds the queue and sets up the head and tail pointers. Running the task at the head of the queue must wait until the *Dispatch* function is called. If the *TaskHead* pointer is zero then *Dispatch* does nothing. Otherwise it copies *TaskHead* to *RunningTask* and relinks *TaskHead* to the next entry in the queue. It then calls *RunTask* which runs the task via the entry's function pointer field *tpntr*.

Figure 3 shows what happens to the queue after the *Dispatch* function has been called. At the point in time shown here the task is running and pointed to by the *RunningTask* pointer. The *TaskHead* pointer has been updated to point to the next task on the queue.

When the user task exits, control returns to *RunTask* which clears the status bits and the task's *next* field. A task can reschedule itself before exiting by calling *ReRunMe*, which sets *NewTask* to equal *RunningTask*. The *RunTask* function then puts the just completed task back on the tail of the respective queue ready to run again. Otherwise the queue entry for the task is cleared and that task becomes idle. The *RunTask* function then returns to *Dispatch* which returns to the user's main program loop.

Observe that the routines that manipulate the queues are surrounded by *gintoff* and *ginton* instructions. These are SmallC extensions that turn global interrupts off and on respectively. They are only required if you allow interrupt routines to add tasks to the queues. While possible this is not recommended and I would suggest the interrupt set a flag that is picked up from within the main program loop that includes the call to *Dispatch*. If the interrupt flag is set the corresponding task can be queued and the flag cleared, ready for next time.

The Timer Queue

A very nice feature of this dispatcher is that the number of simultaneous timers is not limited nor are there any additional overheads in supporting multiple timers. It does this by arranging the delay queue into ascending delay order such that the value of each delay includes the sum of all previous delays. This means that only the delay value at the head of the queue need be decremented at each clock tick to effectively decrement all other delays. As an example, if we had three tasks A, B and C on the queue with delays of 5, 8 and 14 milliseconds respectively and a system clock tick of one millisecond, the delay queue would look like:

A	5	first delay
B	3	second delay (5+3 = 8)
C	6	third delay (5+3+6 = 14)

There is a processing cost in doing this of course, but the time is expended when a task is put on the delay queue via *QueueDelay*, a considerably more infrequent event than the operations occurring at each clock tick in the *DecrementDelay* function. Including the links, the initial queue will look like:

Task	A	B	C
Next	B	C	0
Delay	5	3	6

with **DelayHead* pointing at task A. Suppose now a fourth task D with a delay of 10 milliseconds is added to the queue. After adding task D the queue will look like:

Task	A	B	D	C
Next	B	D	C	0
Delay	5	3	2	4

Additions are in italics and changes in bold.

Delays are decremented by the micro's internal timer which is configured to tick at a rate suitable for your application. The corresponding interrupt routine must call the `DecrementDelay` function in this module. In SmallC for the AVR micros, the *interrupt* qualifier ensures the current C environment is saved while `DecrementDelay` is executed. You will need to consider this when porting the code to other compilers.

Calling `QueDelay` puts a new task on the delay queue. After some initial checking, this calls `DoQueDelay` to actually insert the new task. The local variable `IntVal` is set to the `NewDelay` value passed in by the caller and `Pntr0` is set to `*DelayHead`, the pointer to the head of the delay queue (actually `task[0].delay`). The variable `Pntr1` is set to zero. The while loop then moves progressively through the queue subtracting each delay from the remains of `NewDelay` until either the end of the queue is reached or `IntVal` goes negative, indicating that the sum of all the previous delays exceeds the new delay. At this point `Pntr0` contains the index where the new entry should go, and `Pntr1` contains the index of the immediately preceding entry. Thus the assignments `task[NewTask].next = Pntr0` and `task[Pntr1].next = NewTask` insert the new entry and relink the queue structure. If `Pntr0` is zero it means the end of the queue was reached and the new delay is larger than all the other delays and should therefore go on the end of the queue. If `Pntr0` is not zero but equals `Pntr1` then there were no delays previously listed and the *next* field can be set to zero. The delay setting for the new task is set to `OldIntVal` and the delay for the subsequent task set to the negative of the currently negative `IntVal` value, making it a positive value. Finally the function pointer is filled in, the status set, and the `NewTask` and `NewDelay` variables cleared.

Possible Changes and Extensions

The *delay* field in the list structure is defined as an *int* while in many systems a *char* may be adequate. This change will save a few bytes and requires only a few casts to implement. Likewise the *status* field is not used within the module, so this could also be removed if required.

It is possible to list the same task more than once on a queue. This makes little sense and could cause problems in some systems. However the possibility can occur in systems where the task has some global responsibility, like clearing error states, and can be called from a number of tasks. In this case the `GetNewTask` function could be modified to also search for a matching task pointer and return a status flag if this should occur.

An interesting option is to add another task queue (still in the same structure) to give high and low priority queues. The low priority queue works as described here and with tasks being put on the high priority queue via an interrupt. Assume a lengthy low priority task is running and an interrupt occurs that must schedule a high priority task. The interrupt routine calls `QueTask` to add the task to the high priority queue and then calls a new function called

`Preempt`. If a high priority task is already running `Preempt` simply returns. Otherwise it transfers `RunningTask` to `PreemptedTask` and calls `Dispatch` which calls `RunTask` which has been modified to always run tasks from the high priority queue first and therefore here runs the task added by the interrupt routine. Once this completes it returns to `RunTask` which returns to `Dispatch` which restores `RunningTask` from `PreemptedTask` and then returns to the `Preempt` function which returns to the interrupt which returns to the originally interrupted low priority task. Torturous but simple to implement.

Conclusion

The dispatcher stands alone and can be compiled into a library and simply linked as required. The only change necessary is matching the `TOTALTASKS` constant to suit your application. This need not of course, match the total number of tasks, but only the maximum number that will be running at any one time. You should carefully examine what is going on in your program to arrive at this value. In a complex controller with many devices and many states it may be worthwhile adding code to monitor the task structure during development and display in some way the maximum value `NewTask` reaches in `GetNewTask`.

The complete C code is shown in Listing 1 and a simple test program demonstrating multi-tasking in Listing 2. While by no means a complete real time operating system (RTOS), I think you will find it entirely adequate for most embedded systems.

ⁱ Ron Kreymborg is currently with the CRC for Meteorology at Monash University in Melbourne, Australia (ron@shm.monash.edu.au). He has worked with microprocessors since the very early days, both in hardware and software. He programs in many languages and on many operating systems.

ⁱⁱ "Multitasking for the 8086", AP-61, July 1979

ⁱⁱⁱ See <http://www.shm.monash.edu.au/~ron>