It’s a fact of life that many embedded systems survive perfectly well without a multitasking real-time operating system (RTOS). I have always wondered if I could find a single criterion that would tell me if it would be an advantage or a liability to include an RTOS for a particular project.

How hard is it to guess if an RTOS is hidden inside an embedded box simply by looking at it, using it, and reading the owner’s manuals? Do you think your VCR has an RTOS hidden inside? What about your car? And your mobile phone? (Maybe, just as some car models proudly state their engine size and the number of valves on the back, an embedded system should clearly state “DRIVEN BY XOS. Number of tasks: 24. RAM size: 2MB.”)

Chances are your guess will be wrong. A really complicated, large system may use an infinite loop plus a couple of hard working interrupts. On the other hand, a smaller system with less than 20,000 lines of C code may have a full-blown commercial RTOS. Let’s consider two extreme cases. First, consider a primitive software-driven refrigerator. Our refrigerator software simply monitors and controls the temperature and handles the door-open switch and the lamps. Even a hard-core advocate of operating systems would hardly insist on procuring a suitable RTOS.

At the other end of the scale, consider something much more complex (and not normally viewed as an embedded system), something probably sitting in a close proximity to you right now. A personal computer. I guess only a few really brave individuals would proclaim that a proper multitasking operating system on the desktop is an unnecessary luxury. But wait! You only need to look back a decade or so. Back then, the world’s PCs weren’t driven by multitasking operating systems. In fact, DOS simply provided some software interfaces to the hardware, some memory management functions, and a few other bits and pieces.
Somewhere in the middle, between a fridge and a desktop PC is a fine line between a “yes we need one, let’s get the sales rep” and “no, let’s just get a few guys and start coding.”

Our fridge can become more complex because marketing now wants a keypad so the user may manually enter defrost schedules. Suddenly, we need to provide a real-time clock, a display, and a keypad. Since we now have a display, it might be wise to provide some cool graphics to entertain the users as they fetch beers from the fridge. Marketing also wants to make the software compatible with the next generation of voice-activated fridges.

For small to medium embedded products, there is no simple answer to the “yes” or “no” question. The following drawbacks should be considered if you choose to include an RTOS:

- If you decide to buy a commercial off-the-shelf product, the one-off or per-site royalty costs can be considerable
- Your RTOS will require extra resources
- Time-critical aspects of the system may be negatively affected. For example, interrupt latencies may make your high-speed comms more difficult
- It is never easy to port an RTOS, no matter what ads in electronic and software magazines claim. Take into account the time and effort involved in porting the RTOS you choose
- If you choose a pre-emptive RTOS, remember that the power of pre-emptive multitasking comes at a price. Make sure that all the engineers you are about to hire understand that a single thread of code may be interrupted and execution passed to another task, that data shared between tasks must be protected from simultaneous access, and so on
- Many in-circuit emulators and debuggers refuse to work reliably under an RTOS. You should be prepared to calm down angry engineers who are trying to hit an elusive breakpoint while the kernel code keeps rescheduling away.

**LISTING 1** A sample main control loop

```c
int main(void)
{
    Init_All();
    for (;;) {
        IO_Scan();
        IO_ProcessOutputs();
        KBD_Scan();
        PRN_Print();
        LCD_Update();
        RS232_Receive();
        RS232_Send();
        TMR_Process();
    }
    // should never ever get here
    // can put some error handling here, just in case
    return (0);  // will keep most compilers happy
}
```

**FIGURE 1** Event-driven task: data/event flow diagram
On the other hand, if you decide to take a short cut and not use an RTOS in a system where you really need one, things will be much worse. You will find that your software is getting more and more clumsy, the system keeps falling over and hangs in the most unexpected places, until finally you call it a day and start again from the scratch, using a few pieces of code you can salvage from the mess.

To find the right balance, it’s important to realize that a system without an RTOS can exhibit multitasking behavior. It seems to me that often the main reason behind a decision to port an RTOS where it is unnecessary is the lack of understanding that by using some simple means and some not-so-complex code, an efficient, fast, and reasonably balanced system can be built. Moreover, should an RTOS be required some time in the future, this is no longer a painful exercise, as the system can be engineered as self-contained tasks, even in the absence of a “true” RTOS. In the following sections, I will describe such a system.

**Main control loop**

Our system exhibits a set of neat features, which are normally attributed to the existence of an RTOS:

- The software is broken into stand-alone, well-defined tasks. It is perfectly valid to say that a particular function belongs to a particular task, and the same can be said about data structures
- Event-driven tasks are supported. These tasks have input event queues and execute only when a suitable “trigger” event arrives. Otherwise, these tasks are idle and consume very little processing time
- All tasks may send event messages to event-driven tasks
- Periodic tasks (that is, tasks that do not require a trigger to run) can be

```c
typedef unsigned int word;
typedef struct
{    word InPtr;       /* Head of buffer */
     word OutPtr;      /* Tail of buffer */
     word Count;       /* Counter */
     EVENT_TYPE Store[BUFFER_SIZE]; /* Actual data buffer space */
}  INPUT_EVENT_QUEUE_TYPE;

#define OK      0
#define EMPTY   0xFFFF
#define FULL     0xFFFF

INPUT_EVENT_QUEUE_TYPE EventBuf;

void InitEventBuffer(void)
{
    EventBuf.InPtr = 0;
    EventBuf.OutPtr = 0;
    EventBuf.Count = 0;
}

static word GetEvent(EVENT_TYPE *event)
{
    if (EventBuf.Count)
    {
        EventBuf.Count--;
        // copy from the buffer
        memcpy(event,&EventBuf.Store[EventBuf.OutPtr],sizeof(EVENT_TYPE));
        if (EventBuf.OutPtr >= BUFFER_SIZE - 1)
        {
            EventBuf.OutPtr = 0;
        }
        else
        {
            EventBuf.OutPtr++;
        }
        return (OK);
    }
    return (EMPTY);
}
```

Listing 3 continued on p. 150
executed at a pre-defined speed. Depending on the requirement, this speed can be either exactly defined or relative to the speed of execution of the main control loop.

- Some basic means of inter-task communications are provided. These include stopping and restarting tasks, slowing them down, and speeding them up.
- Software timers provide a convenient method of performing a variety of duties that require exact timing (for example, flash a cursor at a fixed rate). The software timers can be one-shot or run forever.

In addition, as our system is not pre-emptive (tasks cannot be interrupted by another task) we have the luxury of not worrying about protecting our data with semaphores/mutexes. All of our tasks relinquish control only when the entry function of the task returns.

As an example, let’s consider an embedded system with a keypad, an LCD, and an RS-232 port that runs some comms. The system also has some I/O and a parallel printer. Each change of state of an input or output results in an RS-232 message sent out, a printout, and an LCD update. Received RS-232 messages can result in printouts, LCD updates, and output status updates. We may have to start a flash pattern on a particular lamp as a result of:

- An input or output becoming active (fridge door open)
- Keypad entry (defrost schedule entered),
- Comms message received (the owner is e-mailing us to cool the beer down just before he arrives)

Let’s have a quick look at our main control loop in Listing 1. Nothing new or original here. Three things are immediately obvious:

- Each function called in our infinite

---

**LISTING 3, cont’d. Input event queue implementation**

```c
word PutEvent(EVENT_TYPE *event)
{
    if (EventBuf.Count < EH_BUF_SIZE)
    {
        EventBuf.Count++;
        // copy to the buffer
        memcpy(&EventBuf.Store[EventBuf.InPtr], event, sizeof(EVENT_TYPE));
        if (EventBuf.InPtr >= BUFFER_SIZE - 1)
        {
            EventBuf.InPtr = 0;
        }
        else
        {
            EventBuf.InPtr++;
        }
        return (OK);
    }
    return (FULL);
}
```
loop represents an independent task
• Each of these tasks must return in a reasonable time, no matter what thread of code is being executed
• We have no idea at what frequency our main loop runs. In fact, the frequency is not constant and can significantly change with the changes in system status (as we are printing a long document or displaying a large bitmap, for example)

So what happens inside the functions called from our infinite loop?
The majority of tasks in our system are event-driven tasks. They do not execute until a suitable message is received. Each of these tasks has a dedicated input event queue. For example, `IO_ProcessOutputs` is an event-driven task. It handles the state of outputs and does nothing if there are no state changes to be performed on the outputs. However, if an output needs to be turned on, an event message is sent to this task. In our system, three tasks will send event messages to the `IO_ProcessOutputs`:
• The input scanner (`IO_Scan`) task, when input state change dictates an output state change
• The RS-232 receive task, when an RS-232 message is received, requesting an output to be turned on/off
• The keypad scanner task, (`KBD_Scan`), when an entry is completed with the request to turn on/off an output

The handling of the events inside `IO_ProcessOutputs` is exactly the same, no matter which of the three tasks has sent the event. (The event-driven task structure is described in the next section.)

Other tasks in our system are periodic. They run without a trigger event. Some need to run faster, others slower. For example, we may need to scan the inputs at a much faster rate than the LCD update. How do we achieve different execution speeds if the functions are called from the same control loop?

I have also promised to provide some simple means of inter-task communications. For example, we may want to stop scanning the inputs after a particular keypad entry and restart the scanning after another entry. This would require a call from a keypad scanner to stop the I/O scanner task. We may also want to slow down the execution of some tasks depending on the circumstances. Let’s say we detect an avalanche of input state changes, and our RS-232 link can no longer cope with sending all these messages. As a solution, we would like to slow down

---

**LISTING 4  Sending an event to a task**

```c
// somewhere in RS232 module
OUTPUT_EVENT_TYPE OutputEvent;
OutputEvent.NewState = 1;  // new state - on
OutputEvent.Number = 1;   // turn on output number one
OUTPUT_PutEvent(&OutputEvent); // inject an event
```

**LISTING 5  Example of an event-driven task**

```c
// this task is called from main control loop
void IO_ProcessOutputs(void)
{
    word ret;
    OUTPUT_EVENT_TYPE OutputEvent;

    // our normal execution counter processing goes here..
    // ..
    // end of execution counter processing

    if ((ret = OUTPUT_GetEvent(&OutputEvent) != EMPTY))
    {
        // buffer not empty
        // process OutputEvent turn on/off the required output
        IO_OutputStateChange(OutputEvent.Number, OutputEvent.NewState)
    }
}
```
the I/O scanner task from the RS-232 sending task. All this is achieved using the proven and reliable, if extremely unoriginal, technique of execution counters, and is described in the Periodic Tasks section.

In addition to all these features, we need to perform a variety of small but important duties. For example, we want to dim the LCD exactly one minute after the very last key was pressed. We also want to flash a cursor on the LCD at a periodic, fixed and exact frequency. Since dedicating a separate task to each of these functions is definitely overkill, we handle them through software timers. Rather than being explicitly called from the main control loop, the function to turn the cursor on/off is called indirectly from TMR_Process task, which is the only non-user-defined task in the main control loop.

**Event-driven tasks**

Figure 1 shows the concept of an event-driven task. In our implementation, each event-driven task has a single input queue, implemented as a ring buffer. Two functions are provided, PutEvent, to be used by any task when an event needs to be inserted in the queue, and GetEvent, to extract the message. GetEvent will be used exclusively by the task itself and will not be called by other tasks. See Listing 2.

Please note that the EVENT_TYPE structure is unique for each task. In other words, the task itself determines the format of events it expects to receive. For example, in the IO_ProcessRequests task, we would want to include the number and the new state of the output. In the printer task we could simply construct the PRN_EVENT_TYPE as a buffer large enough to contain a single null-terminated string. Since EVENT_TYPE is likely to be different for each task, the user will have to define a unique structure, based on INPUT_EVENT_QUEUE_TYPE, for each event-driven task. Moreover, each task will have its own GetEvent, PutEvent, and an initialization function.

A simple ring buffer structure allows us to achieve asynchronous reads and writes to the buffer, and to
store up to \texttt{BUFFER\_SIZE} entries. Any other task, or the task itself, can inject events of \texttt{EVENT\_TYPE} into the input ring buffer. The events will be inserted at the position of \texttt{OutPtr}, which will grow until the “owner” task executes and reads events from the buffer. When the task extracts events, the position of \texttt{InPtr} is adjusted. When \texttt{OutPtr} and \texttt{InPtr} are equal, the buffer contains no unprocessed events. The “Count” member contains the number of unprocessed events in the buffer.

Listing 3 shows the implementation of \texttt{InitEventBuffer}, \texttt{GetEvent}, and \texttt{PutEvent}. It’s quite simple to imagine how other tasks would activate the outputs. All they need to do is to construct an event of \texttt{OUTPUT\_EVENT\_TYPE} and call \texttt{OUTPUT\_PutEvent}, as shown in Listing 4.

With all other tasks happily using the \texttt{OUTPUT\_PutEvent} function, the only thing we need to do is implement our output controller task. This is straightforward, as shown in Listing 5.

Note that by simply changing “if” to “while” in the function above we can significantly change the behavior of the task. Instead of processing one event per iteration of the task, it will process all the events in the buffer. Another valid idea would be to implement a mixture of two methods, where a task extracts a maximum of X events at a time. Even better if X can be changed by other tasks. Changes to Listing 5 to implement these ideas are straightforward, and I leave them to the reader.

Periodic tasks

Nothing is stopping us from calling any function from the main control loop. However, we have to consider two things first:

- We don’t want to call our tasks too often. Why would we call a watchdog refresh function every 20ms if

```c
Listing 6  Execution counter processing

volatile unsigned int LCD_ReloadValue = LCD_TASK_DEFAULT_SPEED;
volatile unsigned int LCD_ExecCounter = LCD_TASK_DEFAULT_SPEED;

void LCD_Process(void)
{
    #ifdef EXACT_TIMING
        disable();  // disable interrupts for a short instance
    #endif
    if (LCD_ExecCounter == TASK_DISABLED)
    {
        #ifdef EXACT_TIMING
            enable();
        #endif
        return;
    }
    #ifdef EXACT_TIMING
        // note contention possible with interrupt routine
        if (LCD_ExecCounter)
    #else
        if (--LCD_ExecCounter)
    #endif
    {
        #ifdef EXACT_TIMING
            enable();
        #endif
        return;
    }
    // if we reach this point, we are about to run our task
    // reload the execution counter
    LCD_ExecCounter = LCD_ReloadValue;
    #ifdef EXACT_TIMING
        enable();
    #endif
    // application task code follows
    // ...
    // end of function
}

// and this is our fixed time interrupt routine...
interrupt void TIMER_IRQ_10ms(void)
{
    // other stuff
    #ifdef EXACT_TIMING
        if ((LCD_ExecCounter != TASK_DISABLED) && (LCD_ExecCounter))
            --LCD_ExecCounter;
    #endif
    // other stuff
}
```
The reason for using a variable rather than a constant for reload value is that this helps us to dynamically manipulate the execution speed of the task.

**LISTING 7**  A paced memory checking task

```c
void MEMORY_Check(void)
{
    static unsigned int State = 0;
    static unsigned int Csum;
    byte *Ptr;
    // the usual exec counter processing goes here
    if (State == 0)
    {
        // zero the csum
        Csum = 0;
    }
    // derive the memory pointer
    Ptr = (void *)(RAM_START + State*100); // do CRC 100 bytes at each
    // iteration
    AddToCsum(&Csum, Ptr, 100);
    if (++State == 100)
    {
        if (Csum != GetRAMCsum())
            // error  csum mismatches  do something
            Panic();
        State = 0;
    }
}
```

we really only need to do it every second?

- We don’t want to delay the execution of other tasks for too long

In regards to the first problem, a mechanism is available to help slow down the tasks. This can be done in terms relative to the main control loop, or in absolute terms (we will call it exact timing). In both cases, we need two variables per task. One is called an execution counter and the other reload value. The execution counter counts down from reload value. When it reaches zero, the task is called, otherwise the entry function to the task exits without further processing. The reason for using a variable rather than a constant for reload value is that this helps us to dynamically manipulate the execution speed of the task. See Listing 6.

Some points about the code:

- In the exact timing system, the execution counter is decremented in a fixed frequency interrupt routine, while in relative timing system the task itself decrements the counter. In an exact frequency system, we know how frequently our task will execute (minus the latency of other tasks). By setting `LCD_TASK_FREQUENCY` to 100, and using a 10ms interrupt, we know that our task will execute every second plus the execution time of other tasks that may have been called before ours, but after the LCD tasks execution counter was decremented to 0.

- The volatile keyword will prevent optimization by the compiler. Some optimizing compilers will otherwise (in an exact timing system) assume that the `LCD_ExecCounter` will always be set to a non-zero value, especially if the 10ms interrupt handler is coded in a different file.

- `TASK_DISABLED` will be #defined to the largest possible unsigned integer. Setting the execution counter
The periodic tasks must be written in a manner that will guarantee that they return in a reasonable time. This is not always easy.

**LISTING 8** A periodic LCD update task

```c
word WinTaskState = WIN_TASK_IDLE;

void Window_Update(void)
{
    //... reload counters go here
    switch (WinTaskState)
    {
        case WIN_TASK_IDLE:
            return;
        case WIN_TASK_CONST:
            UpdateWinConst();
            WinTaskState++;
            break;
        case WIN_TASK_VARS:
            UpdateWinVars();
            WinTaskState++;
            break;
        case WIN_TASK_GRAPH:
            UpdateWinGraph();
            WinTaskState++;
            break;
        case WIN_TASK_ANIMATIONS:
            if (UpdateWinAnim())
            {
                // all work done, back to idle state..
                WinTaskState = WIN_TASK_IDLE;
            }
            // if not all animations updated, stay in this state for
            // a little while..
            break;
        default:
            // panic
    }
}
```

to **TASK_DISABLED** disables the task until further intervention. This can be done from any other task. For example, an important event can disable our printing process until further notice (a simple form of inter-task communications)

- Our 10ms interrupt is undoubtedly doing a fair bit of processing in the exact timing system. However, with a small number of tasks, it’s hard to believe that the processor will be struggling to do the comparisons and pre-decrements. It will be a good idea to set this interrupt to a lower priority than more critical interrupts in your system (or allow it to be pre-empted by other interrupts). It is debatable if some neater mechanism should be introduced (such as delta queue) to prevent too many counters decremented from an interrupt routine. As the number of tasks for the system of our size rarely exceeds 20 to 30, the delta queue mechanism may be unnecessary (see Software Timers for discussion of delta queues).

- In exact timing systems, care should be taken to avoid preemption of the task by a fixed-time interrupt routine. The exact mechanism is CPU- and compiler-specific. In some cases, it may not be a problem, as 16- or 32-bit reads and writes may be atomic operations. The easiest solution is to disable all interrupts for a short period in the section of code dealing with the execution counter (as shown in Listing 6, by using “disable” and “enable” function calls). This is acceptable in the vast majority of cases.

The periodic tasks must be written in a manner that will guarantee that they return in a reasonable time. This is not always easy. For example, consider a background memory checking task. If the amount of memory to check is large, we will not want to do the whole check in one iteration, as it may take too long. A useful approach to such tasks is to build them as finite state machines, with each iteration taking it to the next state. The processing in each state is limited to the longest permissible time-slice. Sample code for memory checking function is shown in Listing 7.
This piece of code will execute 100 times to check 10,000 bytes of RAM. We control how often the memory check is done using the usual execution counter and reload value technique. Another variation of the same idea is shown in Listing 8. The task is doing nothing until somebody sets the WinTaskState to any non-idle state. Up to four following iterations will then update relevant portions of information in the window. For example, to repaint the entire window, we would set WinTaskState to WIN_TASK_CONST, which will update all the constant information, all variable information, and all graphics and animations in the window. After this update the window update task will become idle again until the next trigger via a WinTaskState. Conveniently, if we only want to update the animations, we can set the WinTaskState to WIN_TASK_ANIMATIONS (in this case, we will not repaint the constants, variables, and static graphics on the screen). Figure 2 shows the LCD update task’s state transition diagram.

Software timers
The software timers add a “real” multitasking behavior to our system. There are hundreds of actions that require events to happen at a fixed time once or periodically. Some examples are:

- A blinking cursor
- An output that needs to be periodically switched on/off
- The need to show a user message that will disappear or change after a fixed period
- The need to light up the keypad or the LCD and turn it off after a period of inactivity
- A short reset pulse to the printer
- A flashing lamp

Do we really want to dedicate periodic tasks to these little, yet important, functions? Most of these will also require exact timing, making our 10ms interrupt work very hard. Our

The software timers add a “real” multitasking behavior to our system.

**LISTING 9** Expired timer data structure and ring buffer

```
typedef struct
{
    word ID;   /* Timer ID */
    dword Parameter;   /* Parameter */
    void (*TimeoutHandler)(word Id, dword Parameter);
} EXPIRED_TIMER_EVENT_TYPE;
```

```
typedef struct
{
    byte InPtr;   /* Head of buffer */
    byte OutPtr;  /* Tail of buffer */
    word Count;   /* Number of expired timers in the buffer */
    EXPIRED_TIMER_EVENT_TYPE Store[MAX_TIMERS];   /* Actual data buffer space */
} EXPIRED_TIMER_BUF;
```

**LISTING 10** Application interface of the software timer module.

```
word TMR_InstallTimeoutHandler(word timer_handle, void(*timeout_func)(word, dword))
word TMR_Start(word timer_handle, word timeout, dword parameter);
word TMR_Stop(word timer_handle);
```

**LISTING 11** Software timer task

```
EXPIRED_TIMER_EVENT_TYPE *Timer;
void TMR_Process(void)
{
    if ((Timer = TMR_GetEvent()) != INVALID_TIMER)
    {
        if (Timer->TimeoutHandler != NULL)
        {
            // call the user timeout function indirectly..
            Timer->TimeoutHandler(Timer->ID, Timer->Parameter);
        }
    }
}
```
main control loop will become long and messy. So we must find a neater solution.

You may have noticed a call to \texttt{TMR\_Process} in the main control loop. This is the only non-user-defined task in our system. In our implementation, the \texttt{TMR\_Process} task itself is an event-driven task, and works exactly as the user-defined event-driven tasks described in this article.

Internally, the timer task could use something like the structures in Listing 9 to define the timer task input event queue. We use the \texttt{Parameter} field to add some flexibility to our module. The application may choose to use it for a variety of purposes, or simply ignore it.

The software timer module in Listing 10 provides the three essential functions for use by the application tasks.

Before calling \texttt{TMR\_Start} and \texttt{TMR\_Stop} functions, a user-defined timeout function for each timer must be installed. This is done by calling \texttt{TMR\_InstallTimeoutHandler} with two parameters: a unique timer ID and a pointer to a timeout function. After this, a timer can be started and stopped by using the \texttt{TMR\_Start} and \texttt{TMR\_Stop} functions.

The implementation of \texttt{TMR\_Process} is simple. Internally, it will use the \texttt{TMR\_PutEvent} and \texttt{TMR\_GetEvent} functions, familiar to us from the discussion of the event-driven tasks. See Listing 11.

A data/event flow diagram in somewhat simplified RT/SASD notation is shown in Figure 3.

You have already guessed that the 10ms interrupt is, in our case, the only provider of the timers into the expired timers ring buffer. As we can have hundreds of software timers running at any given time, we should implement this part of our system in a slightly more intelligent way. Decrementing hundreds of timer counters at each 10ms interrupt seems like a bad idea. Instead, we will use the well-known technique of a delta list. The timers are inserted in the delta queue based on their timeout values. Only the timer that is next due to expire is decremented in the interrupt routine. For example, if we have timers that are set to expire in 10, 60, and 200 clock ticks, our delta queue looks like:

10
50 (equals 60 minus 10)
140 (equals 200 minus 60)

And the only one decrementing is the timer with 10 ticks left.

The slight drawback of this technique is the fact that we need to do a bit more processing when we add or remove a timer to/from the delta queue. Plus, there is a tricky case of two timers due to expire at the same time (some delta queue implementations cheat and add an extra tick to avoid this situation).

So what happens when the 10ms interrupt routine decrements the head of the delta queue to zero? Of course, it doesn’t call the timeout handler directly, as this would take a large, application-determined amount of time and can’t be executed with interrupts disabled. Instead, it removes it. 


The point where a proper RTOS becomes more of an aid than an impediment is hard to define and depends on the specific system.

from the delta queue and simply calls the `TMR_PutEvent` with the ID and parameter of the expired timer. This inserts the expired timer into the expired timer ring buffer.

I will leave the full implementation of the software timer module to the reader. An exercise in delta queue coding always proves to be a good brainteaser. There are some interesting aspects to be taken care of. For example, when the application calls `TMR_Stop`, we must make sure that we handle the removal of the timer from the delta queue (if the timer is running) or the expired timers ring buffer (if it has expired but the `TMR_Process` hasn’t pulled it out yet).

With all the complexity hidden in the software timer module, the application code implementing a blinking cursor looks quite simple indeed. See Listing 12.

Tidying up
The simple and well-known methods described in this article are sufficient to build reasonably complex and well-balanced applications. However, it would be nice to tidy up a number of unstructured elements in the examples in this article. Some of the things we don’t like:

- Having too many globals called `XXX_ExecutionCounter` and `XXX_ReloadValue`. It would be much better if they were grouped in a single data structure.
- We don’t like the fact that each task in the very beginning does exactly the same processing of the execution counters. It would be nice to handle it in a single piece of code.
- We would like to refer to the tasks by their unique identifiers. Instead of writing `IOSCAN_ExecutionCounter = TASK_DISABLED`, it is much nicer to write `OS_Stop(TASK_ID_IO_SCAN)`.
- We do not want to `#define` values for reload values for each task. We would rather read them from a tidy data table.
- When tasks have to be slowed down or speeded up, for modularity, rather than directly manipulating the reload/execution values for the task, it is much better to do this via a generic function call. For example, to halve the execution speed of the I/O scanner task:

  ```
  OS_SetReloadVal(TASK_ID_IO_SCAN, OS_GetReEntryValue(TASK_ID_IO_SCAN) / 2)
  ```

This can be done by introducing a uniform task structure, which would include common aspects of the task functionality. All RTOSes no matter how big or tiny, have a task table in one form or another. And it is not surprising that the result of our little discussion may evolve into something reminiscent of a “real” RTOS.

Indeed, we were talking about tasks, input queues, sleep and idle states—concepts normally found in RTOS manuals. So, with a little effort, the reader may create his own, “almost real” RTOS, with some help from the discussions and examples in this article.

Simple, yet complex
Reasonably complex real-time embedded applications can be built without a “true” multitasking RTOS. Unfortunately, the point where a proper RTOS becomes more of an aid than an impediment is hard to define and depends on the specific system.

---

**LISTING 12**  Example of implementation of a blinking cursor

```c
// somewhere in the initialization section of the application code
TIMER_InstallTimeoutHandler(TIMER_ID_CURSOR_BLINK, CursorBlink);
// start the 330 ms timer  blink around 3 times per second
TimerStart(TIMER_ID_CURSOR_BLINK, 33, 0);
// end of initialization

// this is the actual timeout handler function
// parameter is  the next state of the cursor  on or off
void CursorBlink(word handle, dword parameter)
{
// do the actual drawing / erasing on the LCD
CursorOnOff((byte)parameter);
// restart the timer. next time it will expire with the
// third argument in the opposite phase
TimerStart(TIMER_ID_CURSOR_BLINK, 33, parameter ? 0 : 1);
}
```

Michael Melkonian is a software developer with MIL/DKS Building Automation in Sydney, Australia. In the past 10 years, Michael was involved in embedded software development for a number of companies in a variety of fields. You can contact him at mindflow@tpgi.com.au.

Special thanks to Jim Chase, for kindly reading and reviewing this article.

**Recommended reading**