

Designing an efficient RTOS for a resource-constrained 8-bit Microprocessor

Until recently, tool support for software development on resource-constrained 8-bit microcontrollers has been limited: often, applications have been constructed as cyclic or interrupt-driven systems programmed in assembly code. These devices have been designed to obtain maximum performance from minimum silicon, resulting in an architecture that is very constrained in terms of memory. Traditionally, running a commercial RTOS on such 8-bit platforms has not been considered practical due to the resource requirements - both memory and processor overhead - of a conventional RTOS.

PREEMPTIVE MULTITASKING IN AN 8-BIT ENVIRONMENT

TIt may come as a surprise to application developers that an RTOS using preemptive multitasking can provide a way to improve an application's effective use of the limited resources of an 8-bit CPU without incurring significant memory overheads. In fact, with the right tool support, it is possible for an RTOS to reduce total system cost while also making application development and maintenance easier. There are three areas where using a preemptive RTOS offers benefits over a cyclic executive:

- Avoids the inefficiencies of a cyclic executive;
- Provides the ability to deal with sporadic events with a short latency and still meet deadlines;
- Allows separation of timing and architectural issues from the application code.

The inefficiencies described above occur because a cyclic executive works by executing all of its tasks at the same frequency or at harmonics of some given frequency. In a real-time system, each task will have a frequency at which it must be executed in order to meet the objectives of the system, and the frequency at which the cyclic executive runs will be the fastest of these frequencies (also called the minor cycle). This typically results in a number of tasks being executed more frequently than is strictly necessary, resulting in wasted CPU time.

IMPROVING SCHEDULING EFFICIENCY

This effect becomes even more pronounced when sporadic events with a short deadline need to be considered. If the response to an event is required within one minor cycle of the event occurring, even though the event might occur on average, once every hundred cycles, every minor cycle needs to allow for this event to occur. Compounded with this are the problems that are introduced when a single task needs to be split across several cycles (for example, a task that occurs once every 10 cycles, but requires 3 cycles worth of execution to complete).

This may result in a sequence of minor cycles, which differ in small respects. If this is the case, these minor cycles are aggregated into a major cycle. An example of all of the above is shown in Figure 1.

If a preemptive RTOS is used instead of a cyclic executive, it is possible to improve on all of the above concerns. Tasks can be executed at their natural frequency, rather than executing at some multiple of the major cycle. It isn't necessary to waste processing power allowing time for sporadic tasks that won't happen most of the time. Nor is it necessary to introduce additional structuring constraints in the system (for example, partitioning a task into three parts, so that the task fits into a cycle). In fact, by exchanging a cyclic scheduler for a preemptive one, the user is able to save a large proportion of CPU time that would have been wasted executing cyclic components ineffectively. It also allows the removal of the code and data space overheads of cyclic control structures.

USING AN RTOS ON MICROCHIP'S PIC18C

The benefits outlined above are made possible for 8 bit microcontrollers by the development of single shot kernel technology (see sidebar: single shot execution), as exemplified by Realogy's Real-Time Architect, which includes SSX5, designed for deeply embedded applications, and OSEKernel, aimed at automotive applications. We will use Microchip's popular 8-bit design, the PIC18, as an example.

Probably the single hardest constraint for an RTOS to overcome on the PIC18C architecture is the size of the program stack: no more than 31 return addresses can be stored on this. The data stack is likewise limited: 256 bytes of data memory are available for use as function parameters and local variables. For an RTOS that uses one stack per task, there are two ways to deal with this constraint:

- When a context switch occurs, the entire program stack is copied into RAM. On the 512-byte RAM variant of the PIC18C, storing the stacks for 5 tasks would leave only 47 bytes for other RTOS use, and

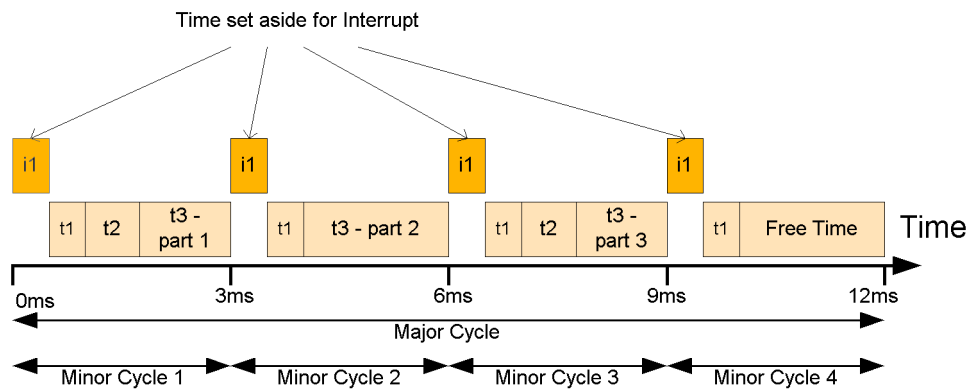


Figure 1. Cyclic executive with four tasks and an interrupt. Each major cycle consists of four minor cycles. Task t1 executes every minor cycle, whereas t2 occurs every other minor cycle. The execution of t3 needs to be split over three minor cycles. Although the interrupt (i1) can only occur once every hundred major cycles, it is necessary to assume that it could occur within any minor cycle (and therefore leaving time for it is necessary).

for applications.

- Each task in the system has a certain amount of the program stack reserved for it. This places very tight constraints upon the depth of the call tree that any task may have. A similar constraint to this applies for the parameter stack: a certain amount of RAM per task needs to be set aside for the use of every task.

SINGLE SHOT EXECUTION MODEL

The use of a single shot execution model (see sidebar) actually allows an RTOS to be based on a single stack, thus overcoming the above difficulties. Tasks only need space on the stack if they are actively running or have been preempted in mid execution (see figure 2). Tasks of the same priority cannot run concurrently, and so if several tasks have the same priority level, we know in advance that only one of these tasks will ever be present on the stack at any moment in time. However, to take advantage of the savings offered by the single stack approach, it is important to determine the overall stack requirements for the entire application in advance. Fortunately, Real-Time Architect provides support for this.

Note: on the Harvard Architecture of the PIC18C, the single stack approach actually requires two stacks (one for code and one for data).

Real-time embedded applications perform a fixed set of activities, so constraints can be placed on them off-line. This allows a build process to optimize the run-time components for the particular application. In the single stack model, the build process includes techniques to allow tasks to overlap their stack usage and so reduce RAM requirements yet further.

OFF-LINE CONFIGURATION

For example, the off-line configuration of the system can include the constraint that two or more of the tasks must never preempt each other (this is achieved by defining a non-preemption group in which these tasks are contained). This kind of constraint can be useful in a number of situations: possibly each of the tasks access a coprocessor - placing these tasks in a non-preemption group ensures that access to the

coprocessor is serialized. Another situation in which the non-preemption group can be used is when porting a legacy system from a cyclic executive (where tasks were not been designed with preemption in mind). Placing the legacy tasks in a non-preemption group avoids the possibility of unintended side-effects arising from preemption.

In the off-line configuration, the stack usage for each individual task can be recorded. Real-Time Architect can use this information (together with knowledge about non-preemption groups and other information such as disabling of interrupts and resource locking) to determine the overall worst-case stack usage.

The benefits obtained by using Real-Time Architect can be illustrated by considering the kernel overheads of SSX5 on the PIC18C for a typical benchmark application: for a 10 task system these are: less than of total 2% CPU utilization, 1.5K of ROM and 400 bytes of RAM (including data stack).

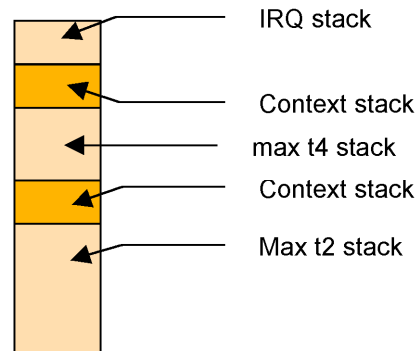


Figure 2. Single stack operation with SSX5 RTOS. This is what makes an RTOS on the PIC18 possible. Five tasks share one stack. Tasks t1, t2 and t3 are low priority tasks and thus will never be active concurrently. Tasks t4 and t5 have high priority and so may pre-empt the lower priority tasks, but will never be active concurrently. The return addresses of low priority interrupts are also stored on the program stack. The return address of a high priority interrupt is stored in a separate register.

TIMING ANALYSIS

In addition to the SSX5 or OSEKernel kernels, Real-Time Architect also provides support that includes timing analysis tools. This is a unique feature for Real-Time Architect, based on schedulability analysis, which extends DMA (deadline monotonic analysis). The most obvious feature of these tools is to allow the developer to guarantee that all deadlines will be met under all circumstances. This is clearly a powerful and extremely useful feature. However, the analysis of Real-Time Architect is not limited solely to showing this. The analysis also shows the amount of slack in the system as a whole, or in individual tasks, allowing for processors to be run at different speeds and still meeting deadlines, or adding functionality to specific tasks. Probably the most interesting feature for the PIC18C is the ability to automatically determine which tasks can be placed into non-preemption groups, thus reducing overall stack usage. Studies performed by Realogy over a wide range of systems show that, irrespective of the number of tasks in a system, systems rarely require more than 5 preemption levels.

The timing analysis can be applied through the development process, from design verification based on a system model and estimates of processing time through verification of implementation to investigating scope for change (headroom) and practicality of proposed enhancements or modifications.

CONCLUSION

In conclusion, the single-shot kernel technology offered by Real-Time Architect extends the "maximum performance for minimum silicon" philosophy of the PIC18C to the total system cost - both hardware and software. The tiny code footprint and cycle-optimizing scheduler of SSX5 provide the ideal complement for the resource-efficient PIC18C architecture. Furthermore, being able to define the configuration of the application ahead of deployment allows further optimization of memory usage. The ability of Real-Time Architect to support predictable (and verifiable) real-time operation is also a significant benefit for using the PIC18C in deeply-embedded hard real-time applications ■

Dr Andrew Coombes CEng is Product Development Manager (Automotive) at Realogy. He graduated with a BSc(Hons) in Computer Science. He received his DPhil in Computer Science at the University of York in the area of safety-critical systems in 1994. He has been involved in a variety of research and consulting work, primarily in the areas of automotive and aerospace sectors. He has worked at Realogy since 1999.

Single shot execution - SSX

A single shot execution model for an OS enables RAM usage to be smaller, by orders of magnitude than that used by conventional RTOSs. With conventional RTOS designs, the approach to real-time tasking is to bind each task to a non-terminating C function as shown in B1.

```
void task1(void) {
    for (;;) {
        do_task1_work();
        delay(100);
    }
}
```

Figure B1: Conventional RTOS execution model

A periodic task is bound to a C function called task1() and does some work periodically. The function never terminates: instead an RTOS "delay" call is used to suspend the task for an interval of time. The task is then marked as "suspended" and the RTOS runs other tasks. When the delay expires, the task becomes ready to run again, and the RTOS switches to running the task when it again becomes the highest priority task. The task then restarts execution at the end of the delay() call and performs more work before again delaying.

```
void task1(void) {
    do_task1_work();
}
```

Figure B2: Single-shot execution (SSX) model

Figure B2 shows the alternative single-shot execution approach. Here the task executes from the start of the function, does some work, then returns from the function. The RTOS itself invokes the task periodically at a rate specified by the developer in a configuration file, as in figure B3.

```
timeline {
    timebase ttime; default readonly;
    periodic f {
        task t1 every 5ms offset 0ms;
        task t2 every 10ms offset 0ms;
        task t3 every 20ms offset 5ms;
    }
}
```

Figure B3: A timeline defining the real-time behavior of three SSX5 tasks.