The basics of embedded multitasking on a PIC: Part 1 - Reentrant multitasking

By Gamal Ali Labib
Embedded.com
(12/03/07, 12:30:00 AM EST)

High-end concepts typically aren't applied to an 8-bit MCU. But don't rule them out just yet.

Applying multitasking to an embedded system based on a PIC microcontroller may look appealing. It permits full use of the PIC resources by serving more than one task at a time rather than staying idle waiting for an external event to occur and reacting accordingly from within an application. However, multitasking can turn applications into memory-hungry monsters that barely fit in your PIC. Task reentrancy is one approach that tackles expanded memory tasks, especially in small to medium PICs.

In this, the first part one of a three part series about reentrant multitasking and how you can realize it with two working examples of a Microchip's PICmicro MCU, I begin with an introduction to multitasking. In part two and three, available on Embedded.com, I'll get more into the meat of multitasking on the PIC.

Embedded multitasking is not new. It goes back some 30 years. The Intel 8080, Zilog Z80, and Motorola 68000 were popular microcontrollers 30 years ago. Interestingly, they were very similar in processing capabilities to many of today's microcontrollers. Those ancient devices had multiple general-purpose and index registers that could perform arithmetic and logic operations, and supported vectored interrupts and I/O ports. They had data buses with widths up to 16 bits, and address buses up to 24-bits wide. They also had large and powerful instruction sets that gave way to efficient programming and ran at moderate speeds of around 10 MHz. What's changed over the years is that today, that same processor is packaged with RAM and EPROM. Adding analog-to-digital modules and EEPROM gives us today's "microcontrollers."

Thirty years ago, full-fledged real-time operating systems (RTOSs) existed and professional features like multitasking and multiprocessing were applied and implemented. This brings up an interesting point: where exactly does multitasking fit with this technology? With a little research, I found that products are available to handle multitasking with an RTOS. Developers must then question whether the RTOS is really required for their application, or near-real-time would suffice.

Back then, in the absence of multitasking, serial task execution was the only way to run embedded systems. Figure 1 shows two tasks running in a system. Task #1 executes until it concludes its operation, then passes control to Task #2. Task #2 runs and eventually terminates causing system operation to end.

Two tasks are running in a system.

![Figure 1](https://www.embedded.com)

By definition, multitasking allows an embedded system to share its resources between several programs running concurrently. The resources can be those of the processor itself (such as internal registers, memory, and arithmetic logic units) or the peripherals attached to the embedded system (such as display, printer, sensors, and actuators).

Figure 2 illustrates a simple form of multitasking. Here, Task #1 is the main task running in the system until some sort of interrupt or an event occurs at time $t_1$. This causes the system to put that task on hold, passing its control to another secondary Task #2, at time $t_2$, to serve the event and willingly return system control to the main task at time $t_3$. This kind of multitasking is based on cooperation between the running tasks, such that tasks are willing to give up system control at
some point throughout their execution. Note that there's no mention of operating-system coordination between tasks; task
delegation of system control is built in the task programs themselves. Nested subroutine calls and branching instructions are
a means of implementing simple multitasking.

A simple form of multitasking is shown.

![Diagram of multitasking](image)

**Figure 2**

Multitasking in an operating-system environment is a different story. System resource sharing is realized under the operating
system's control, rather than task delegation, by allocating those resources to a single task for a specific period of time
before putting them temporarily on hold and switching the resources to another task. The task allocated time and frequency
depend on "multitasking policy." Different policy schemes can be adopted, such as round-robin (equal task rights) and
priority-based (higher priority tasks use resources more frequently than lower priority ones).

**Cooperative vs. preemptive multitasking**

The part of the operating system responsible for task switching is called the **task scheduler**. The scheduler enforces the
multitasking policy whenever it regains system control from the user tasks. User tasks can be designed to either give up
system control to the scheduler voluntarily (scenario #1) or forcibly (scenario #2).

Scenario #1, called **cooperative multitasking**, is where a breakpoint is implanted in the task program to transfer system
control to the scheduler program. The scheduler can later resume task execution at that breakpoint. **Figure 3** illustrates this
scenario with three tasks. Task #1 returns control to the scheduler at time $t_1$. The scheduler decides that Task #2 gets the
control, which is passed on to it at time $t_2$. At time $t_3$, Task #2 returns control to the scheduler and Task #3 gets control at
$t_4$. Task #3 returns control to the scheduler at $t_5$; the scheduler then decides that Task #1 resumes its execution. Task #1
regains control at $t_6$. This scenario repeats until all tasks conclude their operation or the system shuts down.
Scenario #2, called preemptive multitasking, is where task execution is interrupted at any point in time by an internal or external system event. System control is then handed over to the scheduler to decide which task is to resume execution. Figure 4 illustrates this scenario with two tasks. Here, Task #1 runs until an interrupt occurs at time $t_1$, suspending the task. Control is passed on to the scheduler program at $t_2$, at which time the program decides to pass the control to Task #2 based on some policy and activates that task at $t_3$. Task #2 runs until $t_4$, when another interrupt occurs causing it to suspend and the control is passed on the scheduler again at $t_5$. The scheduler decides to resume Task #1 execution at $t_6$, which takes back control until another interrupt occurs. Frequent interrupts to running tasks enable the scheduler to manage system control switching between tasks.

Using preemptive multitasking, task execution is interrupted and system control is handed over to the scheduler to decide which task should resume execution.

Multitasking reentrant programs
The tasks employed for a multitasking embedded system can be replicate control applications or a combination of control, data crunching, and user interfacing. But we know that not all embedded systems have an abundance of memory that fits multiple task programs. In addition, many PICs have limited interrupts, work registers, and stack capabilities that keep them from competing against full-fledged processors. This may explain why it's hard to find operating systems that support multitasking.

One way around the PIC's limitations allows better manipulation of its resources. In computer jargon, it's called reentrant programming. Reentrant programs realize a simple form of multitasking while minimizing memory overhead. When you consider running a number $x$ of tasks of similar functionality, each of which requiring $y$ memory bytes to run, you may think at first glance that you would need $x \times y$ bytes of memory. However, with reentrancy, you would actually require only a fraction $\frac{?}{x}$ of memory over and above that of a single task, giving a total of $\left(1 + \frac{?}{x}\right) \times y$ bytes for the $x$ tasks.

Figure 5 illustrates the reentrant program concept as a special case of multitasking, which was demonstrated in Figure 4. Here, all tasks are similar in functionality and are represented by a single reentrant program. The scheduler switches the

As shown, the process gets more complicated when three tasks are implemented. This is called cooperative multitasking.
A reentrant program is a generic data-driven task. If you manage to separate the task data from the code and make the code register-dependent, you can end up with a generic task. Having done this, another question arises: how can user tasks invoke their reentrant program simultaneously? The answer is twofold:

- First, you must initialize the registers of the generic task with each task’s data.
- Second, you need some mechanism, such as interrupts, to switch the generic task between user tasks.

A user task gaining control of the generic task loads its processor state (its registers’ settings) and resumes its previously interrupted operation. But that only happens after it the user task saves its predecessor state. The task in control of the processor may set a pointer to its descendant task to take over control when the interrupt occurs.

### System limitations

My first encounter with microcontrollers was the Microchip's PIC18F452 MCU. It was a challenge to implement my version of multitasking on such a device. I was faced with limitations that required a workaround that costs memory and execution time. I didn’t encounter such limitations while working with those ancient microprocessor devices.

On one hand, most of the PIC data transfer and arithmetic instructions involve the W register. To facilitate indirect addressing on a PIC, the address must first be loaded into the FSR register. This makes a stack implementation inefficient, comprising many instructions for one indirect memory access. The programmer has to work around this by passing parameters in fixed memory locations.

The PIC’s small hardware return stack allows only minor nesting of calls, so the program structure is almost flat, relying on macros. Also, the PIC lacks support for automatically saving and restoring its state on interrupts. This makes interrupt processing extremely delicate. It’s up to the programmer to save the processor state on interrupt, but this entails a strict instruction sequence.

**Figure 6** shows how the memory in a PIC-based embedded system is allocated for a multitasking environment. Two cases are shown for the program memory: the memory map on the left shows a multiple of similar user tasks with replicated code.
The map on the right uses a reentrant code. Common program modules in both cases are the interrupt handler, the task initialization, and the task scheduler. Those modules are almost identical in both cases. The data memory holding the tasks' work areas is also similar but with minor differences. The software stack in the data memory solves the drawbacks of the PIC's hardware stack.

Gamal Ali Labib is an IT consultant in Cairo, Egypt. He specializes in IT security and IT turn-key projects management. He is also interested in parallel processing and VLSI. Dr. Labib has a B.Sc. and M.Sc. in computer engineering and electronics from Ain Shams University, Egypt, and a PhD in computer science from University of London, U.K. You can reach Dr. Labib at drgamallabib@yahoo.co.uk.

Discuss This Article

5 message(s). Last at: Dec 31, 2007 10:19:24 AM

I was with you right up to to the point where you characterised priority-based scheduling as "higher priority tasks use resources more frequently than lower priority ones". This may be the case on a time-sharing scheduler as used in general purpose desktop operating systems, but it is not usually a characteristic of an RTOS, where a task with higher-priority merely has the right to access a resource in preference to lower priority tasks. That does not mean that it executes more often, merely that it executes when it needs to rather than waiting for lower priority tasks to complete or yield. In most cases such tasks in fact run far less frequently. An RTOS is about deterministic behaviour and guaranteed execution latencies, not about how often a task runs.

It mat have been a throw-away point not critical to the overall article, but I have seen inexperienced developers misapply priorities based upon similar misconceptions, resulting in systems
that do not meet their real-time constraints.

To make my point clearer, consider the scheduler of Windows CE (a true RTOS) for example. You find that threads that have a higher priority are scheduled to run first. Threads that have a lower priority do not run (or get suspended) until all of the threads that have a higher priority have finished. Threads that have the same priority (whatever high or low) run in a round-robin fashion for a time slice, or quantum. This gives an indication that Windows CE has a time-sharing and priority-based scheduler. This also gives some indication to the time slices a task gets, to access system resources, relative to the lifetime of the task invocation.

I have to agree with cpns. Especially with respect to Liu's paper on Rate-Monotonic-Scheduling. Where shorter tasks have higher priorities. Conceptually you could imagine an interrupt handler which has a very high priority (e.g. emergency shutdown), but that doesn't necessarily mean it is running more often. The expected frequency of operation does not directly map to it's priority.

good article though..

I find the transition from discussing non-OS cooperative to OS cooperative confusing. Are there multiple defintions of cooperative? How "cooperative" is it if you insert a breakpoint into a task (without the task being aware of it)? The cooperative OS approaches I'm familiar with on the PIC (for example, Pumpkin's Salvo) use the concept of the programmer inserting Yield() calls into each task, which return control to the OS, which can then decide whether to switch to another task or give this task some more time.

Maybe this is what the author meant, but that's not how I interpret the term "breakpoint". (As I know the term from debugging, it is a stop to a task determined at run time, not at coding time. But Yield() is a stop to a task determined at coding time.)

All very good points. To add a few perspectives and opinions:

"Higher Priority" -- the understanding of that term is the difference between the vast sea of computer programmers and embedded designers. The very essence of the main task that is really doing the work is that it is normally close to the lowest priority. For example, in the field of medical imaging, one is, well, processing images. We have had experience with partners who take that to mean that once an image is being converted from pixels of a depth to a medically usable image, that task grabs almost the highest priority and slows down Ethernet and other comms, basic
sensing, etc. The point we all know is that the highest priority task has to have the shortest duration and the very measure of a real-time system is how short one can make these tasks. Our lower level products are built on Pics and the higher level on Windows / Windows CE under .Net.

"Lack of decent stack". Yup, one of the reasons the PIC can execute real tasks so quickly is that it has a grotty stack configuration. But real-time is not the only way to handle the allocation of storage. The HiTech compilers, for example, do a great job of looking at the whole program during compile time and making sure that an automatic variable in a calling function is not reused in a called function, not matter how deep the stack. As long as the compiler knows what the operational sequence is (they call it the "call graph") one can be guaranteed that the variable will not be used. Of course, reentrant or reused code cannot use this method. We normally run 8 time sliced tasks that do not bump into each other.

But I do like the article from the perspective that one can execute an RTOS that properly utilizes the multi-tasking abilities of the PIC.

Please login or register here to post a comment or to get an email when other comments are made on this article

**Track This Thread**
Sends you email alerts when someone comments on this article

**Discuss This Article**
5 message(s). Last at: Dec 31, 2007 10:19:24 AM

**cpns**  
*Senior Software Engineer*
commented on Dec 3, 2007 5:22:24 AM

I was with you right up to to the point where you characterised priority-based scheduling as "higher priority tasks use resources more frequently than lower priority ones". This may be the case on a time-sharing scheduler as used in general purpose desktop operating systems, but it is not usually a characteristic of an RTOS, where a task with higher-priority merely has the right to access a resource in preference to lower priority tasks. That does not mean that it executes more often, merely that it executes when it needs to rather than waiting for lower priority tasks to complete or yield. In most cases such tasks in fact run far less frequently. An RTOS is about deterministic behaviour and guaranteed execution latencies, not about how often a task runs.

It mat have been a throw-away point not critical to the overall article, but I have seen inexperienced developers misapply priorities based upon similar misconceptions, resulting in systems that do not meet their real-time constraints.

**glabib**  
*Author*
commented on Dec 3, 2007 6:53:46 PM

To make my point clearer, consider the scheduler of Windows CE (a true RTOS) for example. You find that threads that have a higher priority are scheduled to run first. Threads that have a lower priority do not run (or get suspended) until all of the threads
that have a higher priority have finished. Threads that have the same priority (whatever high or low) run in a round-robin fashion for a time slice, or quantum. This gives an indication that Windows CE has a time-sharing and priority-based scheduler. This also gives some indication to the time slices a task gets, to access system resources, relative to the lifetime of the task invocation.

GT Bradley  
*Principal Software Engineer*  
commented on Dec 6, 2007 11:36:08 AM

I have to agree with cpns. Especially with respect to Liu's paper on Rate-Monotonic Scheduling. Where shorter tasks have higher priorities. Conceptually you could imagine an interrupt handler which has a very high priority (e.g. emergency shutdown), but that doesn't necessarily mean it is running more often. The expected frequency of operation does not directly map to it's priority.

good article though..

Stefan Daystrom  
*Embedded Software Engineer*  
commented on Dec 6, 2007 4:03:12 PM

I find the transition from discussing non-OS cooperative to OS cooperative confusing. Are there multiple definitions of cooperative? How "cooperative" is it if you insert a breakpoint into a task (without the task being aware of it)? The cooperative OS approaches I'm familiar with on the PIC (for example, Pumpkin's Salvo) use the concept of the programmer inserting Yield() calls into each task, which return control to the OS, which can then decide whether to switch to another task or give this task some more time.

Maybe this is what the author meant, but that's not how I interpret the term "breakpoint". (As I know the term from debugging, it is a stop to a task determined at run time, not at coding time. But Yield() is a stop to a task determined at coding time.)

Tom Mariner  
*VP Software*  
commented on Dec 31, 2007 10:19:24 AM

All very good points. To add a few perspectives and opinions:

"Higher Priority" -- the understanding of that term is the difference between the vast sea of computer programmers and embedded designers. The very essence of the main task that is really doing the work is that it is normally close to the lowest priority. For example, in the field of medical imaging, one is, well, processing images. We have had experience with partners who take that to mean that once an image is being converted from pixels of a depth to a medically usable image, that task grabs almost the highest priority and slows down Ethernet and other comms, basic sensing, etc. The point we all know is that the highest priority task has to have the shortest duration and the very measure of a real-time system is how short one can make these tasks. Our lower level products are built on Pics and the higher level on Windows / Windows CE under .Net.

"Lack of decent stack". Yup, one of the reasons the PIC can execute real tasks so quickly is that it has a grotty stack configuration. But real-time is not the only way to handle the
allocation of storage. The HiTech compilers, for example, do a great job of looking at the whole program during compile time and making sure that an automatic variable in a calling function is not reused in a called function, not matter how deep the stack. As long as the compiler knows what the operational sequence is (they call it the "call graph") one can be guaranteed that the variable will not be used. Of course, reentrant or reused code cannot use this method. We normally run 8 time sliced tasks that do not bump into each other.

But I do like the article from the prespective that one can execute an RTOS that properly utilizes the multi-tasking abilities of the PIC.

Please login or register here to post a comment or to get an email when other comments are made on this article

Track This Thread
Sends you email alerts when someone comments on this article