Simple non preemptive task switching for 8 bit PIC microcontrollers

Orlin Tomov

Abstract: *The paper describes an approach for a simple non-preemptive/cooperative multitasking on 8 bit PIC microcontrollers, using XC8 compiler.*

Key words: *Computer Systems, Embedded systems, Real time execution, Multitasking.*

INTRODUCTION

Very often, in the embedded systems, there are time critical applications. Moreover, many algorithms are time-based, including tasks, that require delays from a couple of milliseconds to hours. Some are very critical about the time, and some are not so strict. The techniques for software delay are not an option here, because of the requirement for real-time execution. The classical usage of timers is not very handful also, due to the presence of many activities, needed to be performed at various intervals of time on one hand, and the limited number of hardware timers on another. A problem are also the big intervals of a couple of hours, that may require external clock signal. Here we have to use the support of an operating system, or any other approach.

COMPARISON OF SOME POPULAR OS AND APPROACHES

There are many operating systems and libraries, supporting multitasking for 8 bit embedded devices. Some of them are open-source, some are commercial. There are some ports for the 8 bit PIC microcontrollers, like FreeRTOS [1], SalvoRTOS [3], PICThread [4], CalvinRTOS [5], Pic'Xe [6], PicOS18 [7] and many others. In the following table are shown the most important characteristics for each of them.

Table 1

OS/Library	Compiler/ Assembler	Commercial	Cooperative/ Preemptive	Time-based task switching	Additional code size / Overhead
FreeRTOS	C18, wizC	No	Preemptive	Supported	Big
SalvoRTOS	HI-TECH PICC, IAR, C18	Yes	Cooperative	Supported	Relatively Big
PICThread	Any C compiler	No	Cooperative	No	Very small
CalvinRTOS	Assembly / C	No	Cooperative	Supported	Very small
Pic'Xe	Assembly	No	Cooperative	Supported	Very small
PicOS18	C18	No	Preemptive	Supported	Average

FreeRTOS is a very powerful operating system that support many features, as an addition to multitasking, like TCP/IP stack, filesystem, etc. This is one of the very few systems, that support pre-emprive task switching. The main problem here is the overhead (which is a problem for devices with small amount of flash and RAM memory), and the lack of support for the latest compiler from Microchip.

SalvoRTOS is a very functional non-preemptive system, but it is also lacking the XC8 support. Besides it is a commercial OS, which limits its usage and support.

PicThread is a very interesting approach for implementation of non-preemptive multitasking with just three simple macros - THREAD_START(); THREAD_BREAK;

THREAD_END(). All threads are defined within an infinite loop. The first and the last macro are defining the beginning and the end of each process, while THREAD_BREAK defines the places in the thread, where it can be interrupted. This solution uses only 2+n additional bytes of RAM, where n is the number of threads. The code overhead is just 7-8 instructions for THREAD_START(); 5-6 instructions for THREAD_BREAK; and 6 instructions for THREAD_END() [3]. It is an extremely universal idea, that could be ported to any compiler for just a couple of minutes. It is very handy, but when we are talking about real time, and especially for timing applications, critical to tens of microseconds, this approach fails.

PIC'XE is also an interesting solution, but it is written in assembly language, which is a problem for its integration or porting to a modern compiler.

PicOS18 is an open source OS. It supports pre-emptive task switching, but the price for this is a bigger source-code overhead. It is written for the C18 compiler.

From the overview above, it is obvious, that none of the selected items support the latest compiler XC, from Microchip (except PicThread, that could be ported), and there is no simple solution, to allow easy implementation of time-based real time applications.

A SUGGESTION

In order to provide timing delays from milliseconds to a couple of hours, without the need of an external clock signal, we need a a precise system timer. To do this without the help of the hardware is impossible. So we need a time tick from one of the device timers and to use its overflow interrupt. Practically, we can use Timer0 and set it up to fire an interrupt after 1ms. In the ISR we can re-initialize T0 and increment a 24 bit unsigned integer, which will be our system timer. Thus we can achieve overflow every 4hours and 39 mins. So our process can use relatively precise delay from 1ms to 4h39m.

Adding a delay to a process should be as follows:

1. A process checks the current status of the system timer;
2. Add the needed delay to the timer and store the result in a variable , specific for the current process;
3. Give the CPU resources to the next process.

To make this functional, we need to define the following array of structures:

```
struct multithread_timing{
  bit enabled;
  unsigned short long deadline;
  void (*thread)(void)
  };
multithread_timing threads[10];
```

Where *deadline* is the time, when the process should continue, calling a function, pointed by the *(*thread)(void)*. The time in *deadline* is according to the system timer. The flag enabled indicates whether the *deadline* is a valid time or not.

In the ISR, except the increment of the system timer, we need a cycle code, that checks every structure of the array *threads* and inspects its deadline (fig. 1). If it is reached, it calls the function, defined by the *(*thread)(void)* pointer.

Several requirements should be considered, implementing this approach. XC8 can not use the classical algorithm for calling functions, because of the limited depth of the hardware stack. Instead if this it supports a lookup table [7] to store the return address of every function. Thus, the hardware stack is used only for the return addresses from interrupts. So this option has to be enabled, otherwise the stack can overflow and the program will have unpredictable behaviour.

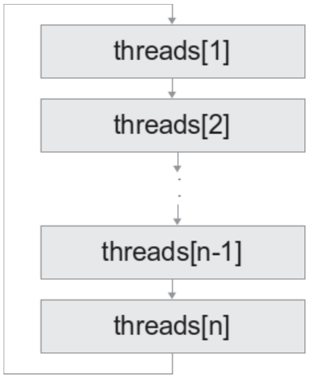

Fig. 1.

Another consideration is to find the balance between the length of the minimal time slice (the time at which T0 overflows) on one hand, and the maximum length for execution of a function, before it free the CPU resources.

If we have a huge functions, not finishing within the time slice, that may cause a fast exhausting of the hardware stack and the system will fail because of the loss of its return addresses. So if the algorithm does not require timing or task switching within 1ms, it is not a good idea to lower this time. As a conclusion we may say – longer time slice, bigger function code, more freedom to the programmer, less accuracy with the time measurement. And vice-versa – smaller time slices require smaller function code, so the programmer should be very careful about the execution time of each function not to exceed the time slice, but he will reach a better time-response of the system.

CONCLUSIONS AND FUTURE WORK

The suggested approach is simple and useful, but requires time for the programmer, to examine the code for the scheduler (ISR) and to design his application to "fit" in the suggested format. A further development of this project should deliver a library, containing a delay macro, allowing multiple tasks to be executed.

REFERENCES

[1] http://www.freertos.org/
[2] http://www.pumpkininc.com/
[3] http://www.romanblack.com/PICthread.htm
[4] http://fse.bc.ca/Calvin.html
[5] http://picxe.sourceforge.net/
[6]http://softelec.pagesperso-orange.fr/Projects/RTOS/PICOS18/Projects_RTOS_PICOS18_us.htm
[7] http://www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en559017

About the author:

Assist.Prof. Orlin Tomov, PhD, Department of Computer Systems, University of Rousse, e-mail: OTomov@ecs.ru.acad.bg

The paper has been reviewed.