µRTOS: Simple Multitasking with Microcontrollers

Professor Dr Dogan Ibrahim, lecturer at the Near East University in Cyprus, describes the design of a C-based, simple multitasking RTOS, using PIC microcontrollers

EMBEDDED SYSTEMS are usually microcontroller-based systems that represent a class of reliable and dependable dedicated computer systems designed for specific purposes. Microcontrollers are used in most electronic devices in an endless variety of ways. For example, it is estimated that there are more than 50 microcontrollers used in intelligent appliances in a modern average household in Europe. Some applications areas are in telephone systems, microwaves, washing machines, cookers, digital TVs, remote control units, Hi-Fi equipment, PCs, MP3 players, mobile phones and so on.

Some microcontroller-based embedded systems are required to respond to external events in the shortest possible time and such systems are often referred to as real-time embedded systems. It is important to understand that not all embedded systems are real-time and, also, not all real-time systems are embedded. For example, most of embedded automotive systems can be classified as real-time systems. Various specialized control functions in a vehicle, such as engine control, brake and clutch control are examples of real-time systems.

Most complex real-time systems require a number of tasks to be processed independently and this requires some form of scheduling and task control mechanisms. For example, consider an extremely simple real-time system which must flash an LED at required intervals and at the same time look for a key input from a keyboard. One solution would be to scan the keyboard in a loop at regular intervals while flashing the LED at the same time. Although this approach may work for a simple example, in most complex real-time systems, a real-time operating system (RTOS) or a multi-processing approach are usually employed. Multi-processing is beyond the scope of this article and this approach requires the use of more than one processor (e.g. more than one microcontroller) and is generally used in parallel computing applications where very high-speed processing, as well as multitasking, are required.

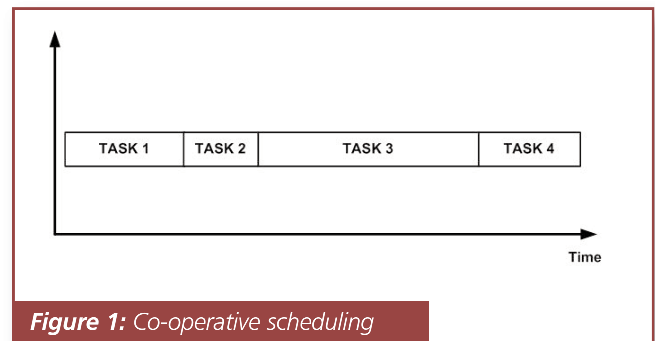

Figure 1: *Co-operative scheduling*

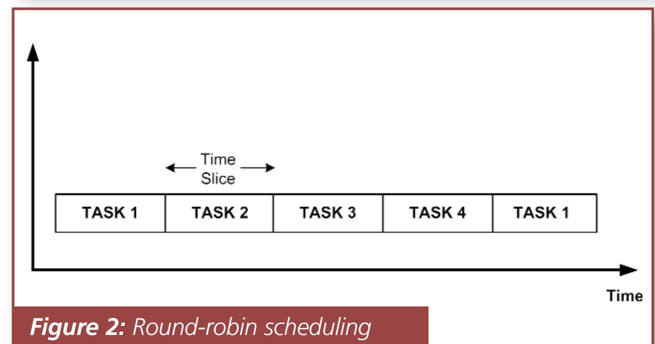

Figure 2: *Round-robin scheduling*

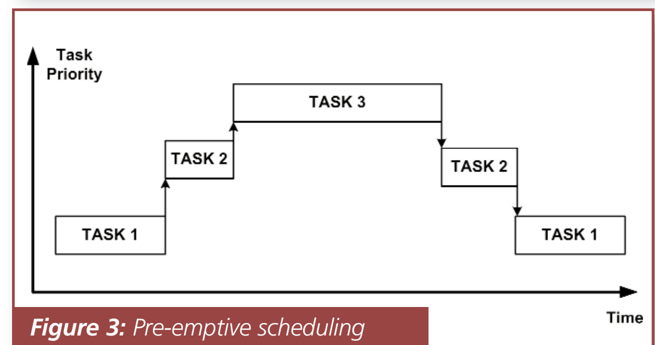

Figure 3: *Pre-emptive scheduling*

The Basic Principles of an RTOS

An RTOS is a program that manages system resources, schedules the execution of various tasks in the system and provides services for inter-task synchronization and messaging. There are many books and sources of reference that describe the operation and principles of various RTOS systems.

Every RTOS consists of a kernel that provides the low level functions, mainly the scheduling, creation of tasks and inter-task resource management. Most complex RTOSs also provide file-handling services, disk input-output operations, interrupt servicing, network management and user management.

A task is an independent thread of execution in a multitasking system, usually with its own local set of data. A multitasking system consists of a number of independent tasks, each running its own code and communicating with each other in order to have orderly access to shared resources. The simplest RTOS consists of a scheduler that determines the execution order of the tasks in the system. Each task has its own context consisting of the state of the CPU and associated registers. The scheduler switches from one task to another one by performing a context switching where the context of the running process is stored and context of the next process is loaded appropriately so that execution can continue properly with the next task. The time taken for the CPU to perform

context switching is known as the context switching time and is negligible compared to the actual execution times of the tasks.

Although there are many variations of scheduling algorithms in use, the three most commonly used algorithms are:

- Co-operative scheduling
- Round-robin scheduling
- Pre-emptive scheduling.

The type of scheduling algorithm to be used depends on the nature of the application and, in general, most applications use either one of the above algorithms, or a combination of them, or a modified version of these algorithms.

Co-operative scheduling (see **Figure 1**) is perhaps the simplest algorithm where tasks voluntarily give up the CPU usage when they have nothing useful to do or when they are waiting for some resources to become available. This algorithm has the disadvantage that certain tasks can use excessive CPU times, thus not allowing some other important tasks to run when needed. Co-operative scheduling is used in simple multitasking systems with no time critical applications. A variation of the pure co-operative scheduling is to prioritize the tasks and run the highest priority computable task when the CPU becomes available.

Round-robin scheduling (see **Figure 2**) allocates each task an equal share of the CPU time. Tasks are in a circular queue and when a task's allocated CPU time expires, the task is removed and placed at the end of the queue. This type of scheduling can not be satisfactory in many real-time applications where each task can have varying amount of CPU requirements depending on the complexity of processing involved. One variation of the pure round-robin scheduling is to provide a priority-based scheduling, where tasks with the same priority levels receive equal amounts of CPU time.

Pre-emptive scheduling is the most commonly used scheduling algorithm in real-time systems. Here, the tasks are prioritized and the task with the highest priority among all other tasks gets the CPU time (see **Figure 3**). If a task with a priority higher than the currently executing task becomes ready to run, the kernel saves the context of the current task and switches to the higher priority task by loading its context. Usually the highest priority task runs to completion or until it becomes non-computable, for example by waiting for a resource to become available. At this point the scheduler determines the task with the highest priority that can run and loads the context of this task. Although the pre-emptive scheduling is very powerful, care is needed, as an error in programming can place a high priority task in an endless loop and, thus, not release the CPU to other tasks. Some multitasking systems employ a combination of round-robin and pre-emptive scheduling. In such systems, time critical tasks are usually prioritized and run under pre-emptive scheduling, whereas the non-time critical tasks run under round-robin scheduling, sharing the left CPU time among themselves.

So far, we have said nothing about how various tasks work together in an orderly manner. In most applications, data and commands must flow between various tasks so that the tasks can co-operate and work together.

Figure 4: *Program listing of µRTOS*

```
/*          uRTOS PIC18 Multitasking          */

#pragma disablecontexsaving
#define MaxTsk 3
#define freq 8
#define Prescale 64
#define T 1000
#define Timervalue 256-(T*freq/(4*Prescale))
#define StopTask while(1){Swap=1; INTCON.F2=1;}
#define SwapTask {Swap=1; INTCON.F2=1;}

unsigned char TMR0 = Timervalue;
unsigned char Temp,Twreg,Tstatus,Tbsr,Swap = 0;
unsigned char Saved[MaxTsk][3];
unsigned char TaskNumber=0;
unsigned char TStack[MaxTsk][4];
unsigned int TCount[MaxTsk];
unsigned int TTime[MaxTsk];

void interrupt()
{
   TMR0L = TMR0;
   Twreg = WREG;    Tstatus = STATUS;    Tbsr = BSR;

   TCount[TaskNumber]++;
   if((Swap == 1) || (TCount[TaskNumber] >= TTime[TaskNumber]))
   {
     TCount[TaskNumber] = 0;
     if(Swap == 1)Swap=0;

     Saved[TaskNumber][0] = Twreg;    Saved[TaskNumber][1] = Tstatus;
     Saved[TaskNumber][2] = Tbsr;
//
// Save return address of current task
//
     TStack[TaskNumber][0] = TOSL;    TStack[TaskNumber][1] = TOSH;
     TStack[TaskNumber][2] = TOSU;
     asm POP
//
// Get next task, and save its return address on TOS
//
     TaskNumber++;
     if(TaskNumber > MaxTsk-1)TaskNumber=0;
     asm PUSH
     Temp = TStack[TaskNumber][0];    TOSL = Temp;
     Temp = TStack[TaskNumber][1];    TOSH = Temp;
     Temp = TStack[TaskNumber][2];    TOSU = Temp;
//
// Restore task registers and return from interrupt
//
     INTCON=0x20;
     Temp = Saved[TaskNumber][1];    BSR = Saved[TaskNumber][2];
     WREG = Saved[TaskNumber][0];    STATUS = Temp;
   }
   else
   {
     INTCON = 0x20;    WREG = Twreg;    STATUS=Tstatus;    BSR = Tbsr;
   }
   asm retfie 0
}


void SetUpTmrInt(void)
{
   T0CON = 0xC5;    TMR0L = TMR0;    INTCON = 0xA0;
}


void InitTask(unsigned char TaskNo, unsigned int t)
{
   TStack[TaskNo][0] = TOSL;    TStack[TaskNo][1] = TOSH;
   TStack[TaskNo][2] = TOSU;    TTime[TaskNo] = t;
   asm POP
}


void StartTasks(void)
{
   SetUpTmrInt();
   Temp = TStack[0][0];    TOSL = Temp;
   Temp = TStack[0][1];    TOSH = Temp;
   Temp = TStack[0][2];    TOSU = Temp;
   asm RETURN
}
```

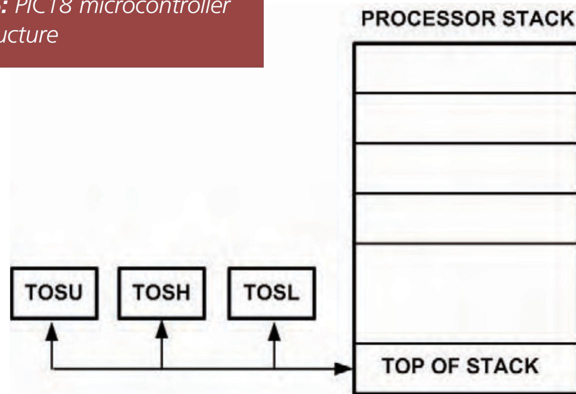

Figure 5: *PIC18 microcontroller stack structure*

One very simple way of doing this is through shared data held in RAM where every task can access. Modern RTOS systems, however, provide local task memories and inter-task-communication tools such as mailboxes and pipes so that data can be passed securely and reliably between tasks. In addition, tools such as event flags, semaphores and mutexes are usually provided for task, as well as inter-task synchronization purposes.

RTOS Systems for PIC Microcontrollers

There are several commercially available, shareware and open-source RTOS systems for the PIC microcontroller family. Brief details of some popular RTOS systems are given in this section.

Salvo (www.pumpkininc.com) is a low-cost, event-driven, priority-based, multitasking RTOS designed for microcontrollers with limited program and data memories. It can be used for many microcontrollers, including the 8051 family, ARM, Atmel AVR, M68HC11, MS430, PIC microcontroller family and others. Salvo is written in ANSI C and supports a large number of compilers, including Keil C51, Hi-Tech 8051, Hi-ech PICC-18, Microchip MPLAB C18 and many others. A demo version (Salvo Lite) is available for evaluation purposes. The SE and LE versions are for systems requiring smaller number of tasks with less features, while the Pro version is the top model aimed for professional applications. The Pro version supports unlimited number of tasks with priorities, event flags, semaphores, binary semaphores, message queues and many more features.

CCS is a C compiler developed and distributed by Custom Computer Services Inc (www.ccsinfo.com) for the PIC microcontrollers. There are several versions of the compiler depending on the type of microcontroller used in the target design. The PCW models of the compiler support built-in RTOS. The provided RTOS is co-operative and requires no interrupt features. The RTOS provides a number of functions to start and terminate a task, to send messages between tasks, to synchronize tasks using semaphores and so on. When a task is scheduled to run, control of the processor is given to that task. When the task is complete, or does not need the processor any more, control returns to a dispatch function, which gives control of the processor to the next scheduled task. Because the RTOS does not use interrupts and is not pre-emptive, the user must make sure that a task does not run forever.

CMX-Tiny+ (www.cmx.com) supports a large number of microcontrollers, including the PIC24 and dsPIC family. This is a pre-emptive RTOS, supporting a large number of features such as event flags, messages, cyclic timers, semaphores and so on. This RTOS can be configured to operate as a co-operative scheduler if required. Although CMX-Tiny+ is a highly sophisticated RTOS, it has the disadvantage that the cost is relatively high and it is not available for lower members of the PIC family.

PICos18 (www.picos18.com) is an open-source pre-emptive RTOS for the PIC18 microcontrollers, developed under the GPL license. The full documentation and the source code is provided free of charge for people wishing to use the product.

MicroC/OS-II (http://micrium.com) is a low-cost priority based pre-emptive RTOS which has been ported to many microcontrollers including the PIC microcontrollers. This RTOS is developed in ANSI C with full source-code and documentation provided, and is used over hundreds of real-time products all over the world. MicroC/OS-II is a highly sophisticated RTOS, providing semaphores, mailboxes, event flags, timers, memory management, queues and so on.

FreeRTOS (www.freertos.org) is an open-source royalty-free RTOS that can be downloaded and used in commercial applications. This RTOS has been ported to many microcontrollers, including the PIC family of microcontrollers. FreeRTOS is pre-emptive but can be configured for co-operative or hybrid operations. The software supports interrupts, queues, mailboxes, binary

```
//
// Flash the LED
//
void Task0(void)
{
    InitTask(0,1);

    while(1)
    {
      if(flag == 1)
        PORTB.F0 = ~PORTB.F0;
      else PORTB.F0 = 0;
    }
}


//
// Stop flashing
//
void Task1(void)
{
    static unsigned int i = 0;
    InitTask(1,1);

    while(1)
    {
      while(PORTB.F1);
      flag = 0;
    }
}


//
// Start flashing
//
void Task2(void)
{
    InitTask(2,1);

    while(1)
    {
      while(PORTB.F2);
      flag = 1;
    }
}


//
// Main program. Call each task and the start Task0
//
void main(void)
{
    TRISB=0x06;

    Task0();
    Task1();
    Task2();
    StartTasks();
}
```

Figure 6: *A simple µRTOS multitasking example*

semaphores, counting semaphores, mutexes and so on.

Finally, OSA-RTOS (http://picosa.narod.ru) is a freeware RTOS for PIC microcontrollers distributed under the BSD license. This RTOS is compatible with a large number of C compilers, including the Microchip C18, mikroC and CCS. The full source code and the documentation are available from the website. OSA is a co-operative multitasking RTOS, offering many features such as semaphores, events, data queues, mutexes, memory pools, system services and many more.

Development of the µRTOS

This section describes the development of a simple, yet effective RTOS for microcontrollers, called the µRTOS. Although µRTOS has been primarily developed for the PIC18 series of microcontrollers, it is written using the C language and, thus, it should be possible to modify and adapt it for other types of microcontrollers.

One of the classical approaches to RTOS design is to use Task Control Blocks (TCBs). A TCB is basically a structure that stores the essential information about a task, such as the return address, CPU registers, task state, local event flags, etc. µRTOS has been designed to have a simple architecture and thus is not based on TCBs.

µRTOS uses a round-robin type scheduler with the addition of allocating variable CPU time to individual tasks. Thus, for pure round-robin type applications all tasks can be configured to have the same CPU time allocations. Task durations, however, can be configured if desired so that tasks can be allocated different maximum CPU times.

Task scheduling in µRTOS is based on 1ms timer interrupts where context switching occurs within the timer interrupt service routine. Data flow between the tasks can be achieved using common variables declared at the beginning of the program. In addition, task synchronization tools are not provided in this simple RTOS.

Figure 4 shows the program listing of µRTOS. The code is based on the mikroC language from mikroElektronika (www.mikroe.com), which is currently one of the popular C language compilers for PIC microcontrollers.

Each task in µRTOS is organized as a C function, running forever in a loop. The first thing a task does is to call kernel function InitTask which saves the task return address in an array called TStack. In addition, the maximum allocated duration of each task (in ms) is also stored in array TTime. The program counter of a PIC18 microcontroller is 24-bits wide and is stored in three 8-bit stack registers TOSL, TOSH and TOSU after a procedure call or an interrupt (see **Figure 5**). These registers are accessed by µRTOS during the saving and restoring of task return addresses.

In an application, the main program initially calls all the tasks in turn so that their return addresses can be saved. Then function StartTasks is called. This function calls to SetUpTmrInt to configure timer TMR0 so that timer interrupts can be generated every milliseconds for the kernel. In addition, the return address of Task 0 is pushed onto the stack and a RETURN is executed so that task execution starts from Task 0. At the core of the kernel we have the timer interrupt service routine (ISR). The ISR determines the next task to run and performs the necessary context switching. The following operations are carried out within the ISR:

- Timer register TMR0 is reloaded for one millisecond interrupts;
- Current CPU registers W, STATUS and BSR are saved;
- If allocated duration of current task has not expired, then timer interrupts are re-enabled and ISR passes control back to the same task with no context changing;
- Otherwise, the return address of current task is saved in array TStack;
- Task number of the next task is determined and its return address is pushed onto processor stack;
- CPU registers W, STATUS and BSR of next task are restored;
- Timer interrupts are re-enabled and ISR passes CPU control to the next task.

Using µRTOS

Here is an example to show how µRTOS can be used in a simple multitasking application. An LEDs is connected to port pins RB0 of a PIC18F452 microcontroller, operated from a 8MHz crystal. Similarly, port pins RB1 and RB2 are connected to push-button switches such that logic 0 is applied to the corresponding microcontroller pin when a switch is pressed. Three tasks named TASK0, TASK1 and TASK2 are created (see **Figure 6**) with the following functions:

TASK 0: This task flashes the LED connected to port pin RB0 as long as a Flag is set.

TASK 1: This task clears the Flag when button connected to RB1 is pressed, thus stops the flashing.

TASK 2: This task sets the Flag when button connected to RB2 is pressed, thus re-starts the flashing.

Each task initially calls to function InitTask with its task number and maximum duration and then enters its endless loop. Task0 flashes the LED as long as the Flag variable is set. Task1 and Task2 both wait in a while loop until either switch RB1 or RB2 is pressed. Pressing RB1 clears the Flag and, thus, the flashing stops. Similarly, pressing RB2 sets the Flag and, as such, the flashing is re-started by Task0. Notice that a Macro called SwapTask can be used to stop execution of the current task and pass control to the next task, i.e. it can be used to force a context switching when a task has completed its execution or it has nothing else useful to do.

The main program initially calls to all the tasks so that their return addresses are saved and then calls to function StartTasks to start execution from Task0. Notice that variable Flag is used as a shared variable and thus it must be declared at the beginning of the program.

Simple and Effective RTOS

The development of a simple, yet effective RTOS system for the PIC18 series of microcontrollers has been described. Although there are several commercially available microcontroller RTOS systems, some are expensive, or use too many resources of the target microcontroller. The RTOS described in this paper (µRTOS) is simple and uses only a timer and very small RAM memory of the target microcontroller. It should be possible to use the µRTOS in many small real-time multitasking applications. The code given in Figure 4 is configured for three tasks, 8MHz clock frequency and 1ms scheduling time. These parameters can easily be changed to suit any other application. ■