7 Steps to Writing a Simple Cooperative Scheduler

Jacob Beningo - September 10, 2014

Real-Time Operating Systems (RTOS) have been extremely popular in recent years. Most engineers will select an RTOS very early on in the design cycle, sometimes even before requirements have been defined. One of the interesting things about RTOS is that for many MCU-based applications an RTOS is overkill. The magic of an RTOS really comes into its own when the application requires task preemption (temporarily suspending a task to switch to a higher priority task and later resuming) and has hard real-time requirements. There are many instances where a much simpler, cooperative scheduler will fit the requirements just as easily.

A cooperative scheduler still allows tasks to be scheduled through the use of a background periodic timer that creates a system tick just like in an RTOS. The difference is that rather than having priorities and preemption, the cooperative scheduler only executes tasks that occur at a time periodic interval. If two tasks are due to run at the same time, the task higher up in the task list runs first followed by the second and so on. The cooperative scheduler allows for soft real-time behavior but through the use of interrupts and other mechanisms can also meet hard real-time needs as well.

One of the great advantages of using a cooperative scheduler is that they are fairly simple and straightforward compared to an RTOS. Debugging an RTOS can be extremely complicated and is usually very painful. A cooperative scheduler on the other hand has very few pieces and is much easier to debug. In fact, a cooperative scheduler can be designed and implemented in just a few easy steps. They also use very little flash and RAM. The cooperative scheduler that is presented in this example can be downloaded from http://bit.ly/1oh8sV5

Step 1 - Define the Scheduler Requirements
Before sitting down and writing any code it is a great idea to get an understanding of exactly what it is that is going to be written. This often means going to the project requirements document and understanding what is required. For a cooperative scheduler, there are a few basic requirements that should be kept in mind:

- The scheduler shall use a single interrupt driven timer to keep track of system time
- Scheduler shall be written so that it can be reused from one project to the next
- The scheduler shall be capable of scheduling periodic and background tasks
- The scheduler shall be easily configured through the use of a configuration table

Step 2 - Create the Software Architecture
When developing reusable software it is critical that a good software architecture be implemented. When it comes to a scheduler that will be used on more than one type of hardware, an architecture that is layered can be the difference between code that is reused forever and code that is thrown away after the first project. Creating an architecture that consists of a driver layer for hardware
dependent code, an application layer for the guts of the scheduler and a configuration layer to configure the application works perfectly! An example of such an architecture can be seen below in Figure 1.

![Layered Architecture](image)

**Figure 1. Layered Architecture**

**Step 3 - Define the Components of a Task**

**Step 3 - Define the Components of a Task**
At a minimum there are three pieces of information that a scheduler needs in order to run periodic tasks properly; the time interval between successive runs of the task, the system tick of the last time the task was executed and the function that should be executed when the task comes due. With these pieces of information it is possible to define a C structure that can be used to define each task that the processor must execute. Figure 2 provides an example of what this structure would look like.

```c
/**
 * TaskType is used to define the parameters required in order to
 * configure a task and execute it.
 */
typedef struct
{
    uint16_t Interval;    //**< Defines how often a task will run */
    uint32_t LastTick;    //**< Stores the last tick task was run */
    void (*Func)(void);   //**< Function pointer to the task */
}TaskType;
```

**Figure 2. TaskType Structure for Task Scheduler**

The definition of this structure is pretty straightforward. The only potentially complex component to the definition is the use of a function pointer to call the task code. In this example all tasks return void and take no parameters. To learn more about function pointers you can visit [Embedded.com](https://www.embedded.com).

**Step 4 - A Task Configuration Table**
Once the TaskType structure has been defined, it is now possible to create an array of TaskType where each element of the array defines a task. For small applications this table will be relatively short. The Tasks array should be defined as a static variable and when possible as a const. This will help ensure that it isn’t possible for the task definition to change during execution of the program. An example of the Tasks array can be found in Figure 3.
In this example, things are kept simple and only the scope of the variable is set. This example consists of three different tasks. The first, is a background task. This task is executed when there are no other tasks that need to be executed. It is defined by setting the interval to 0, ensuring that each loop through the scheduler results in it being ran. This is a task where perhaps polling code (gasp!) might be placed.

The other two tasks in the table define periodic tasks of 10 and 100 milliseconds. The INTERVAL_xxMS definitions are #define statements that specify the number of system timer ticks necessary for the interval to be reached. For example, if the system tick is only 10 milliseconds, then INTERVAL_10MS would be 1. Then again if the system tick is 1 millisecond, then INTERVAL_10MS would instead be 10. These could also be defined in an enumeration.

Step 5 - The First Task Function
Before going too much further in the development of the scheduler it is a good idea at this point to make sure that the tasks defined in the task configuration table are defined. This will help prevent the compiler from getting angry about functions not being defined. The tasks themselves could all be stored in one module or as the author prefers in separate modules. The definition of the task functions is just like any other C function. An example of defining the Tsk_100ms task can be found in Figure 4.

```c
void Tsk_100ms(void)
{
    Wdt_Clear();
}
```

Figure 4. Defining a Task Function

Step 6 - A Few Supporting Configuration Functions

At this point nearly everything concerning the task configuration is setup and ready to go. The only thing missing is a couple of helper functions that the scheduler needs in order to traverse the configuration table. The first, *Tsk_GetConfig, is a function that returns a pointer to the Tasks[] configuration table. This will allow the scheduler to access the table structure. The second, Tsk_GetNumTasks, is a function that returns the number of tasks stored in the configuration table. These two functions exist because good programming practice of data encapsulation is used. This information is limited to the module scope and the scheduler requires these two helping functions to access the data. Their exact implementation can be seen in the example code mentioned at the beginning of this article.
Step 7 - The Birth of a Scheduler
Finally all of the pieces are in place to write the actual scheduler part of the program. The scheduling algorithm for a cooperative scheduler is usually written directly in the main function as can be seen in Figure 5. The scheduler is initialized by creating a pointer to the task configuration table. The number of tasks in the table is also retrieved.

```c
int main(void)
{
    static uint32_t tick = 0;          // System tick
    static TaskType *Task_ptr;         // Task pointer
    static uint8_t TaskIndex = 0;      // Task index
    const uint8_t NumTasks = Task_GetNumTasks(); // Number of tasks

    // System Initialization
    Task_ptr = Task_GetConfig();       // Get a pointer to the task configuration
    Sys_Init();                        // Initialize the system and all peripherals

    // The main while loop. This while loop will run the program forever
    while(1)
    {
        tick = Task_GetSystemTick();  // Get current system tick

        // Loop through all tasks. First, run all continuous tasks. Then,
        // if the number of ticks since the last time the task was run is
        // greater than or equal to the task interval, execute the task.
        for(TaskIndex = 0; TaskIndex < NumTasks; TaskIndex++)
        {
            if(Task_ptr[TaskIndex].Interval == 0)
            {
                // Run continuous tasks.
                (*Task_ptr[TaskIndex].Func)();
            }
            else if((tick - Task_ptr[TaskIndex].LastTick) >= Task_ptr[TaskIndex].Interval)
            {
                (*Task_ptr[TaskIndex].Func)();  // Execute Task
                Task_ptr[TaskIndex].LastTick = tick; // Save last tick the task was ran.
            }
        }
    }
    // End main
}
```

**Figure 5. Cooperative Scheduler Algorithm**

With these two pieces of information, the main loop will start by retrieving the current system tick. In a 32-bit system this is a trivial endeavor since the reading and writing of a 32 bit tick variable is atomic. Next, each task entry that exists in the task configuration table is looped through and examined. If the interval of the task is set to 0 (a constantly running background task), then it is executed. If on the other hand the interval is nonzero then some math is performed to determine if the difference between the last time the task ran and the current time is greater than or equal to the task interval. If it is then the task will be executed.

**Conclusions**
As can be seen the implementation of a cooperative scheduler is pretty simple and straight forward. There aren’t many moving pieces and once the scheduler is built, if done properly it can be reused from one project to the next. The only piece that needs to be changed is the system tick timer and then the task configuration table. While such a simple scheduler doesn’t come with all the bells and whistles of today’s RTOS, it is truly amazing how many applications this simple scheduler actually does apply to.

*Jacob Beningo is an embedded systems consultant and lecturer who specializes in the design of*
resource constrained and low energy devices. He works with companies to decrease costs and time to market while maintaining a quality and robust product. He is an avid tweeter, a tip and trick guru, a homebrew connoisseur and a fan of pineapple! Feel free to contact him at jacob@beningo.com or at his website www.beningo.com.

Related articles:

- Low-cost cooperative multitasking
- RTOS or bare metal - five decisive factors
- The basics of embedded multitasking on a PIC
- Comparing real-time scheduling on the Linux kernel and an RTOS
- Function pointers: Task Scheduling