Function pointers, Part 2: Task scheduling

Jacob Beningo - January 13, 2013

One of the most common applications for using function pointers is in task schedulers. Every embedded system contains some type of task scheduler whether it is a simple round robin loop or a state of the art real-time operating system (RTOS).

One of the simplest approaches to task scheduling with function pointers is a cooperative scheduler. A cooperative scheduler is a glorified round robin loop with the ability to monitor system time and execute tasks at specific time intervals. It executes a single task uninterrupted until it is completed before moving on to execute the next task. In other words, it does not support preemption which would allow a higher priority task to interrupt a lower priority task. A cooperative scheduler doesn’t allow priorities to be assigned to the task.

There are three basic parameters that the scheduler requires each task to have in order to function; task interval, last time executed and the function that should be executed. The task interval defines how often the task should run. This interval should be defined in system time ticks and associated with a variable that is incremented at a constant rate by a timer.

Caution should be taken when designing the system tick in order to ensure that the variable that is updated is written atomically. Timers are usually setup for the task with the shortest interval. This corresponds in most systems to system ticks of 1 or 10 milliseconds depending on the application and the resolution required. The LastTick variable contains the system tick from the last time the task was executed. The purpose of this is to allow the scheduler to calculate if it is time to run the task again. Finally, the scheduler needs to know what function to execute which it is provided through the use of a function pointer.

Listing 1 TaskType definition

The requirements of each task can be defined in a single, typedef structure named TaskType. The definition of TaskType can be found in code Listing 1. In this example, the function pointer is a void pointer that returns and takes nothing. Since the TaskType structure defines all of the information required for each task, an array of TaskType can be created that stores all of the system tasks. Code Listing 2 shows an example of an array named Tasks that contains definitions for each task within a
One of the important points to notice about the Tasks array is that it contains a task with an interval of zero named Tsk. Tsk is a continuous task, or idle task, and will be executed by the scheduler every time it checks to see if a task needs to be executed. The remaining tasks are timed tasks and will execute at the initialized interval. The masks INTERVAL_XXXXMS are #define statements that contain the number of system ticks that must occur before the interval is reached. Notice that the LastTick index for each element in Tasks is set to zero. This indicates that the tasks all ran at system tick 0 and will not run for the first time until their interval has expired!

For example, the 1000 milliseconds task will execute for the first time 1000 milliseconds after the system starts not at time 0 milliseconds and 1000 milliseconds. If for some reason the developer wanted to delay the 500 millisecond task from running for one second after the system boots, the LastTick could be changed from 0 to 1000 (assuming that the system tick is set for a 1 millisecond interval). Finally, Tsk_XXms is a function that is executed at the interval. By listing the functions in this way the address of the function is stored in the function pointer. An interesting point about assigning tasks in this way is that to add a new task all one needs to do is add a new entry to the table.

Since the scheduler will be reused, there needs to be a method for getting the number of tasks in the configuration table and also the address of the configuration table to the scheduler during initialization. Code listing 3 and 4 respectively show how to define functions that perform this purpose. This function would be called at the beginning of main during system initialization to initialize the scheduler.
With the ground work laid, the next critical component of the scheduler to examine is the main loop. The entire main loop can be found in Listing 5. Main starts out by defining some scheduler variables. These include a system tick variable, a pointer of TaskType to store the location of the task table, a TaskIndex variable that is used to move through the task table and then NumTasks which is used to define how many tasks are defined in the task table. Before the scheduler begins scheduling tasks it initializes the system through the Sys_Init function.
Once the setup of the scheduler is completed, the code enters the infinite while loop that continuously runs the scheduler code. The scheduler begins by getting the current system tick value. In the example code it is assumed that this is an atomic operation (i.e. being executed on a 32 bits machine). The scheduler then using a simple for loop designed to check each entry in the task table. The scheduler first checks to see if the task entry is a continuous task. If it is, then the function pointer for the task is executed.

If the task is not the continuous task, the difference between the current system time and the last time the task was executed is compared and then checked against the task interval. If the interval has been met or exceeded then the task is run. In order to execute the task, the scheduler dereferences the function pointer that is associated with the task. The LastTick entry in the table is then updated for that task and the next task is then checked.

The use of function pointers in a task scheduler greatly simplifies the implementation of the scheduler. Without the use of function pointers, a task scheduler would become an unmanageable mess of code that would be difficult to maintain or reuse. Function pointers allow a simple table to be used to contain all of the tasks and for loop to be used to check each task and decide whether it should be executed. This post has hardly scratched the surface of task scheduling but instead has
focused on an example application for function pointers.

*Jacob Beningo* is an embedded software consultant who currently works with clients in more than a dozen countries to dramatically transform their businesses by improving product quality, cost and time to market. Feel free to contact him at jacob@beningo.com, at his website www.beningo.com, and sign-up for his monthly *Embedded Bytes* Newsletter [here](#).

**Related articles:**

- [Function pointers: An introduction](#)
- [Function pointers, Part 3: State machines](#)