Run To Completion (RTC) OS

Jon Norenberg, August 2016

Overview

A number of embedded software systems run on “bare metal”, meaning that there is no operating system present because either there is not enough processing power or memory. The hardware (microcontroller) was selected to supply just enough of the software requirements, but nothing extra since hardware costs money.

These types of systems usually have a forever loop in main() that continually checks the inputs (switches, ports, or sensors) and processes the data and generates some type of output (stepper motor, leds, or sound). At the other end of the software spectrum, is the full blown operating system that allows the Software Engineer to divide the total software work into specific modules or tasks. This allows the designer to focus on one particular task at a time and not have to worry how it affects other parts of the system. I am generalizing here, but you get the picture.

The RTC OS is minimalistic, like the “bare metal” system, but allows the designer to break things down into tasks, that only run when there is data (events/messages) available. Tasks can send events and messages to each other. Another source of events and messages are Interrupt Service Routines (ISRs).

Run to completion means that when a task is called, it gets to run until it has completed its work. This is sometimes known as a cooperative system, where the task cannot be preempted by another task, only an ISR can interrupt it. A byproduct of this type of system is that semaphores and mutexes are not needed in the system. An ISR can interrupt a task, but the ISR will just send an event and then return so that the majority of the ISR work will be done in a task.

OS Configuration

The designer determines the configuration of the system by creating a board file for their system and adjusting a few parameters that are appropriate for their system. Some of these parameters are the number of tasks in the system, number of future events, and number of messages in the system. Changing these, change the memory requirements of the OS as you might expect. You can even turn off messages if they are not needed in your system, which decreases the code size and RAM footprint. There are instances where the OS needs to make sure it does not get interrupted, so there are defines for the critical section that need to be filled in. ENTER_CRITICAL_SECTION and EXIT_CRITICAL_SECTION. Also there is CRITICAL_SECTION_VARIABLE, in case your system needs to store information such as the interrupt level.

Other types need to be defined in the board file for your hardware. These define the size of some variables used in the system.

Tasks

Tasks are called by the OS when there is either and event or message for the task. The task handler is a simple C routine that determines what data has arrived for it and then acts on it. To make the system more responsive the task should only handle the highest priority data and then return to the OS. It should not try to handle all its outstanding data otherwise a higher priority task might be kept from running. Yes, tasks do have priority, which allows the designer to setup the most responsive system for their scenario. As mentioned earlier, if an interrupt was generated because of incoming data on the uart, then you might have a uart receive task as the highest priority, so that the uart ISR can send an event to the task.

```
osEvents_t TaskOne( osEvents_t eventFlags, osMsgCount_t msgCount, osTaskParam_t
taskParam )

{

    osMsg_t newMessage;

    // if you want messages to have a higher priority check them before events

    if ( msgCount )

    {  // just handle one message then return, for higher priority tasks

       if ( OS_ERR_NONE == osGetMessage( &newMessage ))

       {      // somebody sent us a message

                    return eventFlags;

       }

    }

    if ( eventFlags & T1_EVT_TEST1 )

    {   // this event would be the highest priority for this task

        // since it is handled first


        // do something


        // then return the events that have NOT been handled

        return eventFlags & ~T1_EVT_TEST1;

    }


    if ( eventFlags & T1_EVT_TEST2 )

    {

        // do something

        // then return the events that have NOT been handled

        return eventFlags & ~T1_EVT_TEST2;

    }
    // maybe there are some events that we do not care about
    // this will throw those events away
    return 0;
}
```

Events

Events are 1 bit each and can be assigned any meaning in the system and are defined by the designer. Each task can handle up to 32 different events and the priority of the events are defined by the designer. Events can be sent immediately or in the future. Also events can be sent to yourself, so if you want to blink a led you would just continually send a future event with a 500 ms delay.

Messages

Messages are a little heavier than events and are usually pointers to some memory structures that you want to pass between tasks. All messages are sent immediately, there is no such thing as a future message (version 2.0?). Keep in mind that since tasks run to completion, the designer does not have to worry about one task being interrupted before it finishes working on the data pointed to by the message.

System Tick

In order to support future events (or delayed events), the OS must be called every tick. The designer can determine the value of the tick, whether it is 1, 10, or 100 ms, whatever makes sense for the system.

System Sleep

If the system has no work to do, it will call a routine that can sleep the system if needed. This routine is registered at init time. Most microcontrollers have a STOP instruction that will halt the system until an interrupt occurs.

Startup

In main() the designer needs to init the hardware along with the OS and then create the tasks.

```
int main( void )
{
    hwInit();       // defined by the system designer
    osInit();       // set os variables

    // register tasks so that they can receive events
    osRegisterTaskEventHandler( TaskOne, TASK1_ID, 0 );
    osRegisterTaskEventHandler( TaskTwo, TASK2_ID, 0 );
    // register the sleep routine
    osRegisterSystemSleepHandler( SystemSleepHandler );
    // send the T2_EVT_TEST1 to task 2 every 5000 ticks
    osSendEvent( TASK2_ID, T2_EVT_TEST1, 5000, RELOAD_DELAY );

    startTickInterrupt();

    // start the system running, it will never return from this routine
    osRun();

    return 0;
}
```

Getting Started

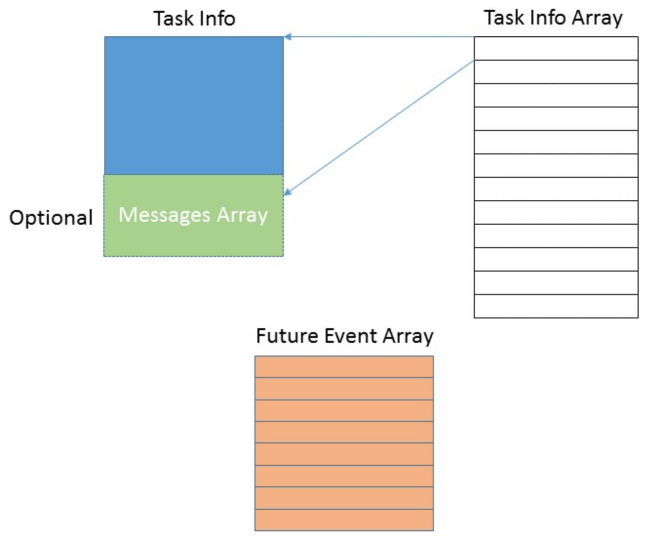
To get started, a directory should be created in the board directory. Copy a board file from another board directory as a starting point for your system. Edit the boards.h file and add your board information along with the compiler flag that indicates your board.

Look at the file that contains main() in the other board directories to give you some hints on setting up your main().

Implementation

Each task has some info pertaining to it in a C structure. An array is used to contain all the tasks and can be iterated through by indexing into the array. Since the iteration starts at 0 then the task info at element 0 is the highest priority because it is queried first about its events and messages. If no data is found, then the OS moves to the next task info element. If it walks through the entire array then it is possible that the system can sleep. The future event list must be checked also before sleeping.

Routines that start with os_ are internal to the file and are not expected to be called by the "application" code: tasks, ISRs, …



To understand how the system works, start with the osRun() routine. Basically, it searches for a task that has data and then executes its task handler.