Lab 5 – RTOS

Real-Time Operating System.

Introduction

An RTOS is often used in a "hard" real-time system – i.e. a system in which threads have to perform not only correctly but also in a timely fashion. You are to use a simple multithreaded, pre-emptive, real-time operating system (RTOS) that implements thread priority. The RTOS also provides services to threads for communication, synchronization and coordination.

One of the problems of the current Lab 4 implementation is the use of callbacks within ISRs. The callback functions could be complex functions that take a significant amount of time complete. Since they are executed inside an ISR, interrupts of the same or lower priority will not be serviced while they are executing. We can get around this problem by putting the same functionality as the callback into a background thread. The thread will wait for a semaphore to be signalled by an ISR before it carries out its task, and then return to the waiting state. Since threads can be interrupted, the interrupt latency of the whole system is reduced and we are more likely to achieve hard real-time constraints.

Objectives

1. To use an RTOS to implement a hard real-time system.

Equipment

- 1 TWR-K70F120M-KIT – UTS
- 1 USB cable – UTS
- NXP Kinetis Design Studio

Safety

This is a Category A laboratory experiment. Please adhere to the Category A safety guidelines (issued separately).

Cat. A lab

L5.2

Software Requirements

1. A simple RTOS is to be used to implement the same functionality as Lab 4.
2. The main application is to be architecturally structured around the use of an RTOS. In particular, multiple threads should be used for the various background tasks.
3. Function modules need to be rewritten so that they can be incorporated into the overall software architecture that uses an RTOS. In particular, you will need to carefully consider the use of interrupts, semaphores, critical sections and priority inversion.
4. Git must be used for version control. Note that version control will be assessed in Lab 5 based on the development of the software from Lab 1 through to Lab 5.

Hints

1. Think about the overall architecture in a multi-threaded environment – lots of small tasks work cooperatively to achieve the overall functionality.
2. Write modules that "block" when they can't proceed. A "blocking" function means that the thread is suspended if it can't carry out its intended function. For example, a call to `FIFO_Get` should suspend a thread if the buffer is empty. This can be achieved by "waiting" on a semaphore that keeps track of the number of items in the FIFO buffer.
3. We haven't used interrupt priorities yet, so all interrupts have "priority 0". This means that interrupts cannot be interrupted, except by a HardFault exception (which shouldn't occur). In other words, interrupts are "off" during an ISR. ISRs therefore need to be quick to reduce the latency of the system (the time it takes for the system to respond to an event).
4. It makes no sense for ISRs to call functions which "block", since an ISR is not a thread – it has no thread control block (TCB), and no private stack. It cannot be suspended, scheduled, etc. by an operating system. It is therefore a mistake to call `OS_SemaphoreWait` inside an ISR. You also need to watch out for implicit blocking functions, e.g. a call to a blocking implementation of `FIFO_Get` inside an ISR is also a mistake.
5. ISRs are allowed to signal, so calls to `OS_SemaphoreSignal` are valid. Indeed, this is the way that ISRs communicate with other threads, (which may be in a suspended state and should become ready to run after the ISR has handled the hardware event). For example, a UART ISR triggered from an RDRF event should signal a waiting thread whose only job is to process the arrival of characters. Similarly, a callback function can be replaced with a waiting thread which is signalled by a semaphore in an ISR.
6. Be careful about the identification of critical sections of code. They can now be protected in two ways: 1) pairs of `EnterCritical` and `ExitCritical` function calls; and 2) a binary semaphore (a mutex).

7. There are many areas where a mutex makes sense – putting a packet (so no other thread can interfere with the placement of 5 bytes into the transmit queue), I2C or SPI reads and writes (so no other thread can disturb the current I2C or SPI transaction), etc. Be aware though that use of a mutex semaphore requires the operating system to be running, i.e. calls to `OS_SemaphoreWait` and `OS_SemaphoreSignal` will enable interrupts and reschedule threads if necessary. Therefore, you can't use mutex semaphores in hardware initialisation code which runs before the OS starts. In these cases, you will need to create a thread of high priority that does the initialization work, then deletes itself.

8. Be aware of the "priority inversion" problem of using mutex semaphores to control access to a critical section, e.g. the internal FIFO array. What would happen if a low-priority thread acquires a mutex semaphore and an ISR occurs which makes a higher priority thread run which also requires access to the resource? The higher priority thread would be "blocked" which may violate a hard real-time constraint of the system.

9. If you find that your code crashes inside `OS.c`, it may be that:

   - you have inadvertently modified the vector table (e.g. through using Processor Expert). Make sure that the vector table has the following:

     ```
     (tIsrFunc)&Cpu_Interrupt,        /* 0x0D Reserved13 */
     (tIsrFunc)&OS_ContextSwitchISR,  /* 0x0E PendableSrvReq */
     (tIsrFunc)&OS_SysTickISR,        /* 0x0F SysTick */
     (tIsrFunc)&Cpu_Interrupt,        /* 0x10 DMA0_DMA16 */
     ```

   - you may be inadvertently waiting on a semaphore within an ISR. You should check carefully that any functions you are calling, both explicitly and implicitly, from within the ISR are not "blocking functions".

   - you may have a stack overflow – in this case simply increase the size of the thread's stack.

10. Doxygen will not compile due to the declaration style of the thread stacks.

Marking

1. **The software to be assessed must reside in the remote git repository before the start of your timetabled activity on the date specified in the Timetable in the Learning Guide.**
2. **Please create a "tag" called "Lab5Submission" (no spaces allowed) to the particular commit that you want marked. Markers will then create a branch from this tag called "Lab5Marking" in order to assess it.**
3. **Software marking will be carried out in the laboratory, in the format of a code review.**
4. **Refer to the document "Software Style Guide" for more details of some of the assessment criteria.**

L5.6

Assessment Criteria

Your lab will be assessed according to the following criteria:

Item	Detail	Evaluation	Mark
Opening comments / function descriptions	File headers and function descriptions are correct.	**E G A P N**	**/0.5**
Naming conventions / code structure	Names and code structure conform to the Software Style Guide.	**E G A P N**	**/0.5**
Version Control	History of development from Lab 1 through to Lab 5. Appropriate and relevant comments for code changes.	**E G A P N**	**/2**
RTOS Framework	Software designed to use an RTOS. Appropriate software architecture. Use of threads. Thread priorities.	**E G A P N**	**/2**
RTOS Function Integration	Module functions re-designed to utilise an RTOS. Appropriate identification of critical sections, use of semaphores, interrupts, etc.	**E G A P N**	**/3**
		TOTAL	**/8**

When we evaluate an assessment item, we will use the following criteria:

Evaluation	Mark (%)	Description
Excellent	100	**All relevant material is presented in a logical manner showing clear understanding, and sound reasoning.** For software – correct coding style, correct software architecture including: modularity; functions; parameters; and types, very efficient implementation (code and time) and/or novel (and correct) code.
Good	75	**Nearly all relevant material is presented with good organisation and understanding.** For software – mostly correct coding style, mostly correct software architecture including: modularity; functions; parameters; and types, reasonably efficient implementation (code and time).
Acceptable	50	**Most relevant material is presented with acceptable organisation and understanding.** For software – inconsistent coding style, reasonable software architecture (but could show improvement in modularity, use of functions, parameters, or types), some code may be prone to errors under certain operating conditions (e.g. input parameters) or usage, occasional inefficient or incorrect code.
Poor	25	**Little relevant material is presented and/or poor organisation or understanding.** For software – Conceptual difficulty of the underlying concepts, numerous coding style errors, functionality missing, poor software architecture, inappropriate or incorrect use of functions, parameters or types. Very inefficient and / or incorrect code.
No attempt	0	**No attempt.** For software – missing modules and/or functionality.

Oral Defence

During the assessment of your work you will be asked questions based on material which you have learnt in the subject and then used to implement the assessment task. You are expected to know exactly how your implementation works and be able to justify the design choices which you have made. If you fail to answer the questions with appropriate substance then you will be awarded **zero** for that component.