Software Specifications & Design Document

Assignment 2, EMBSYS110 , Spring 2015

Chad Bartlett, cbartlet@uw.edu

1. Introduction

1.1. Project Overview

The purpose of this project is implement a context switch within a timer ISR to a task defined in a thread queue. The application created for this project takes no inputs, but does output serial debug information. In addition, LEDs are toggled to indicate the state of the running application. Threads are executed forever using a preemptive, round-robin schedule. The first two threads manipulate the globally shared variable and the third thread monitors the shared variable for errors. If an error occurs, LEDs enter an error pattern state.

1.2. Project Scope

The project scope is limited to the assigned tasks listed in Reference 1.4.1.

1.3. Document Preview

Section 2 describes the system architecture and requirements. Section 3 explains the detailed design. Section 3.4 describes the test plan. Finally, Sections 3.9 and 3.7 elaborates on the serial output and LED illumination patterns.

1.4. References

1.4.1. **UW Tiny Kernel – Assignment 2, A2/CP110**

2. Architectural Design

2.1. Section Overview

The purpose of the section is to provide a high-level description of the application created for this project.

2.2. Block Diagram

The system is composed of the components listed in Figure 1.

Figure 1: System block diagram.

2.3. System Requirements

The system requirements are listed in the following Tables. These requirements are specifically called out in Reference 1.4.1 with the exception of Table 6: Testing Requirements. *The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.*

Table 1: Thread Requirements

SRS_THD_001	The thread queue *must* hold a minimum of four threads.
SRS_THD_002	One thread *must* be called *idle.*
SRS_THD_003	The idle thread *should* be the first thread in the thread queue.
SRS_THD_004	The threads *must* be executed forever using a round-robin schedule.
SRS_THD_005	Each thread *must* support data sharing.
SRS_THD_006	Each thread *must* contain an entry function. The entry function is the function called from the scheduler.
SRS_THD_007	Each thread entry function *should* have a unique name.
SRS_THD_008	The idle thread entry function *should* be called *idleEntry().*
SRS_THD_009	Each thread entry function *must* have the following signature: *void functionName(void* data)*
SRS_THD_010	A hardware timer *must* be implemented that interrupts every 2-milliseconds.
SRS_THD_011	A context switch to the next task in the READY state *must* occur within the hardware timer interrupt described in SRS_THD_011.
SRS_THD_012	All threads *must* operate in an infinite loop.

Table 2: Data Requirements

SRS_DAT_001	A structure, *thread_t*, *must* be defined with the following fields:
	{
	ENTRY // function pointer to thread entry
	DATA // an optional parameter to be passed to the Entry function
	STACK // a pointer to the thread stack
	STATE // thread state; Ex. INITIAL, ACTIVE, READY, BLOCKED
	NAME // a unique name for the thread
	}

SRS_DAT_002	A method *must* be defined to output the fields from the structure defined in SRS_DAT_001.

SRS_DAT_003	A structure, *globalData_t*, *must* be defined with the following fields:
	{
	GUARD // An access guard for this structure
	INC // A counter incremented within the guard in each thread
	T1 // The number of times thread 1 has been entered
	T2 // The number of times thread 1 has been entered
	}

SRS_DAT_004	A structure *should* be created that contains, for example, the ID of the currently running thread. This structure would be used by the kernel as opposed to other threads.

SRS_DAT_005	All structures and variables *should* be defined globally.

Table 3: Communication Requirements

SRS_COM_001	Serial output *must* be enabled to provide feedback during run-time.

Table 4: Run-time Requirements

SRS_RUN_001	An application running 3 threads *must* be created as described in Reference 1.4.1, under Section *Design Requirements*.
SRS_RUN_002	If an error occurs, the error *should* be recorded and reported.
SRS_RUN_003	Serial output *should* be generated throughout the board initialization code to indicate progress.
SRS_RUN_004	A hardware timer *should* be implemented to provide a system tick for the application running at 500 Hz.
SRS_RUN_005	All thread state changes *should* be printed.
SRS_RUN_006	At the completion of the 'idle' thread, the state of the shared variable described in SRS_DAT_003 *should* be printed.
SRS_RUN_007	The CPU % utilization time taken for handling the timer interrupt and carrying out a context switch *must* be determined. See SRS_THD_011 and SRS_THD_012.

Table 5: Documentation Requirements

SRS_DOC_001	All source code *should* be documented using Doxygen with the following keywords where appropriate: • File comment block: @file, @brief, @author, @date • Function comment block: @brief, @param, @return • Inline code using: '/**< *description* */' or '//!< *description*'

Table 6: Testing Requirements

SRS_TST_001	All APIs *must* be verified for conformity to the specifications.
SRS_TST_002	The state of the shared variable defined in SRS_DAT_003 *must* be monitored during run-time to verify the counter variable remains < 3.
SRS_TST_003	The system tick frequency defined in SRS_RUN_004 *must* be verified using an external tool (e.g. oscilloscope, DMM, logic analyzer, etc).
SRS_TST_004	The board initialization execution time *must* be measured and recorded.
SRS_TST_005	The execution time of all threads *must* be measured and recorded.

3. Detailed Design

3.1. Section Overview

This section describes the build environment, scheduler, test plan, thread queue, error handling, serial output and LED illumination patterns. Refer to Appendix Section 3.10

for class diagrams of this application.

3.2. Build Environment

A virtual machine (VM) has been created using VirtualBox for use as the build environment for this project. The VM guest OS is Lubuntu 14.04. The target application is built using the GNU ARM Embedded Toolchain. A STMicroelectronics STM32VLDiscovery discovery kit with the STM32F100RB MCU is used as the target hardware. To download and debug the application binaries, the Texane STM32 STLink driver is utilized. The project is under version control using Git with a repository on GitHub. The source code can be viewed on GitHub here or the repository cloned from the following address:

- https://github.com/bartsblues/EmbSys110_cbartlett.git

The project directory structure is as follows.

```
App/    -
        |- OS.cpp
        |- OS.h
        |- Threads.cpp
        |- Threads.h
        |- TimingMeasurements.h
        |- main.cpp
        |- mini_cpp.cpp
        |- no_heap.cpp
Bin/    -
        |- Assignment2_cbartlett.bin
Bsp/    -
        |- Bsp.cpp
        |- Bsp.h
        |- Interrupts.cpp
        |- startup_stm32f10x.cpp
        |- stm32f100.ld
        |- stm32f10x_conf.h
Doc/    -
        |- Assignment2_Design_cbartlett.doc
        |- Assignment2_Design_cbartlett.pdf
Lib/    -
        |- CriticalSection.cpp
        |- CriticalSection.h
        |- DUart.cpp
        |- DUart.h
        |- IDUart.cpp
        |- IDUart.h
Tests/  -
        |- stubs/   -
                    |- StubBsp.cpp
                    |- StubDUart.cpp
        |- Makefile
        |- RunAllTests.cpp
        |- TestCriticalSectionAPI.cpp
        |- TestIUart.cpp
        |- TestTemplate.cpp
./      -
        |- .ddd_gdb
        |- .nemiver_gdb
        |- README
        |- Makefile
        |- Makefile.Includes
        |- Makefile.Common
```

3.3. Scheduler

The scheduler executes a preemptive, round-robin pattern. Execution begins with the *idle* thread. Three additional threads are implemented to manipulate and monitor a shared variable. All threads have the same priority. Threads can be in one of four states: READY, ACTIVE, BLOCKED or DELAYED. Context switching is implemented in alignment with the Cortex-M processor series OS support features—namely, the shadowed stack pointer, SysTick timer and PendSV exception.

A depiction of the thread states is shown in Figure 2. If a thread is currently running it will be in the ACTIVE state. From here it can enter the other three states. The BLOCKED state is entered if the thread is waiting for access to an area of memory guarded by a critical section. Threads are able to delay their execution by calling the OS::TimeDelayMs() function. A transition can be made from any of these states to the READY state. The scheduler selects the next READY state from the thread queue to activate in a round-robin pattern.

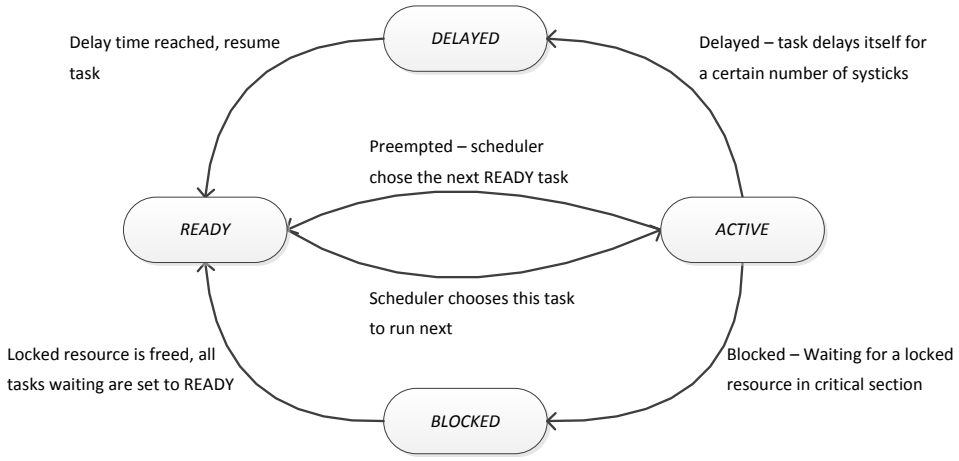

Figure 2 - The thread states of the OS.

Threads execute within the Thread mode of the Cortex-M. In thread mode, the PSP (Process Stack Pointer) is used as the selected stack pointer (SP) for stack operations. This has several benefits including improved system reliability by limiting potential stack corruption to the individual thread stacks, controlling the maximum stack usage for each thread and the ability to utilize the Memory Protection Unit (MPU) for the defined stack regions of the thread stacks. In addition, it makes performing context switches very straightforward.

All Cortex-M processors have a small integrated timer called the SysTick timer. The timer in this application is configured to generate an interrupt every 2-milliseconds. Within the interrupt the OS performs a number of actions. These include the following:

- Determine if any delayed tasks are ready to run
- Find the next task in the thread queue that is in a READY state
- Trigger a PendSV exception to perform a context switch if necessary

Context switching is carried-out within the PendSV (Pendable Service Call) exception handler.

A context switch is only performed if it has been determined that there is a thread ready to execute. The context switch consists of the following actions:

- Push the current task's context (i.e. registers R4-R11) using the PSP
- Update the current task's SP
- Get the SP from the next task to execute and set the PSP
- If the previous task was ACTIVE, set it to READY
- Restore the next task's context

The PendSV has a programmable priority level and is triggered by writing to the Interrupt Control and State Register (ICSR). The PendSV can be triggered from a higher priority exception and executed when the higher-priority handler completes. This is important in the case where an interrupt request (IRQ) occurs before the SysTick exception. By design, the Cortex-M does not allow a return to thread mode when there is a pending interrupt. The problem is solved by configuring the PendSV exception to have the lowest exception priority such that it is guaranteed to execute after all other ISRs.

3.4. Test Plan

The testing framework CppUTest is utilized to create tests that exercise the APIs of this application. These tests are executed automatically within the VM in the guest environment after a successful build for the target.

In addition, the three threads form a run-time test of the round-robin scheduler. The first two threads manipulate a globally shared variable and the third thread monitors the shared variable for errors. If an error occurs, a message is output through the serial-port and the LEDs enter an error pattern state. See Section 3.7. Additionally, each thread controls an LED. Thread 1 toggles the blue LED, thread 2 toggles the green LED and thread 3 sets both LEDs. The timing is such that both LEDs toggle twice for a relatively short period and once for a longer period.

3.5. Thread Queue

The thread queue is a statically allocated array consisting of pointers to 4 *thread_t* objects described in Table 2. The idle thread is contained at the first index, followed by *thread1*, *thread2* and *thread3*. See the Thread diagram in Figure 7 of Section 8.

3.6. Error Handling

Assert statements are used to maintain preconditions within the application. If an assertion fails the file and line number where the failure occurred are printed to serial output and the application enters a *while(1)* loop. The application must be manually reset to continue. All other run-time errors are printed to the serial output. See Section 3.7 as well.

3.7. LED Patterns

During normal operation, the LEDs blink as described in Section 3.4. If the third thread detects an error with the shared data, the both LED flashes at a sporadic rate. If an assertion has occurred both LEDs are steady on until the system is manually reset.

3.8. Namespaces

Namespaces are used to identify functionally related code into logical groupings. The namespaces of the application are Bsp, OS and Thread.

3.9. Serial Output

A minimal UART driver has been created to provide serial output to the application. Three public functions allow for printing either strings or numeric values (in the form of unsigned integer decimal or hexadecimal output). The connection uses a bit rate of 115200 and 8-N-1. Sample Output is shown below. See the class diagram in Figure 4 of Section 8.

Appendices

3.10. Object Diagrams

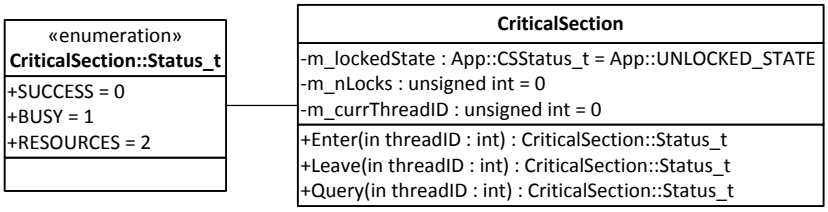

Figure 3 - Critical Section Class Diagram

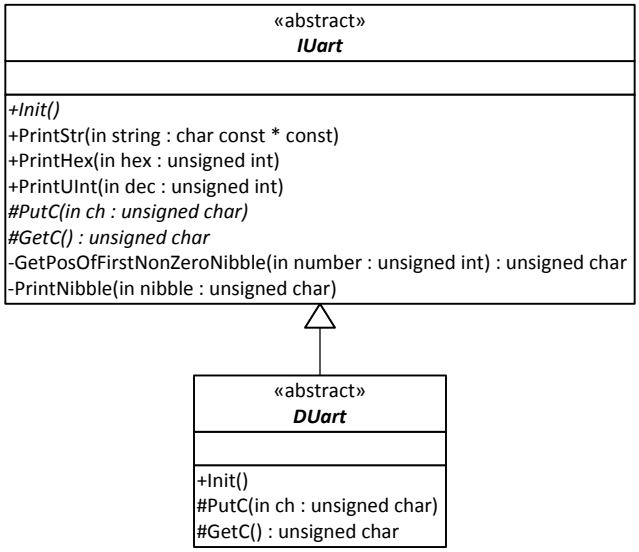

Figure 4- DUart Class Diagram

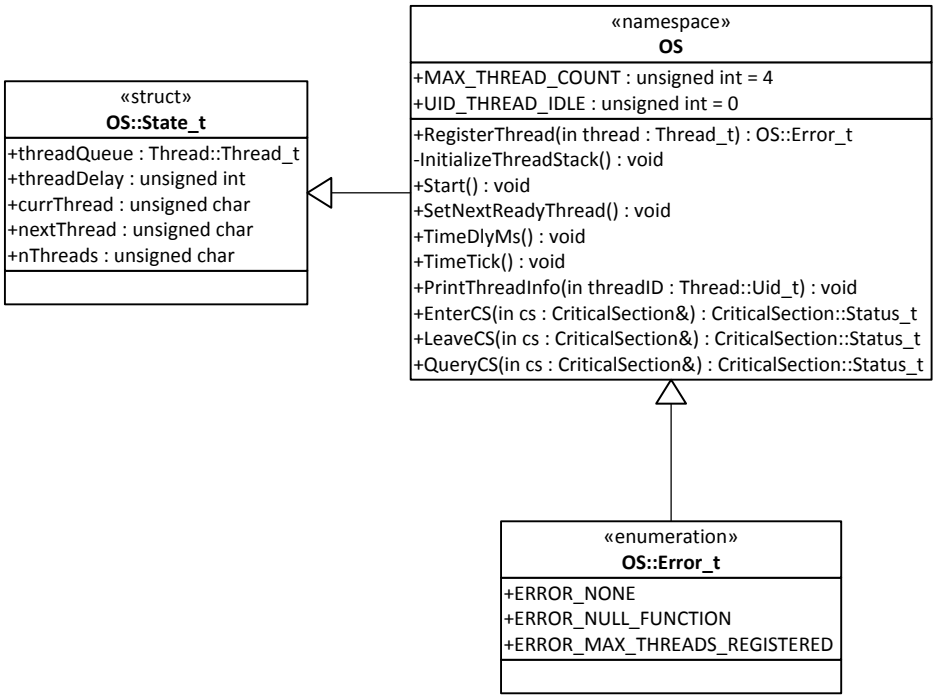

Figure 5- OS namespace Diagram

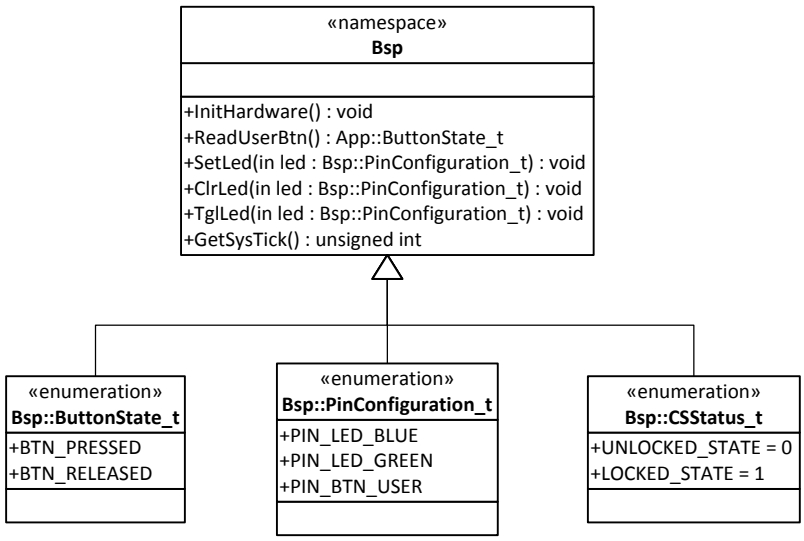

Figure 6 - Bsp namespace Diagram

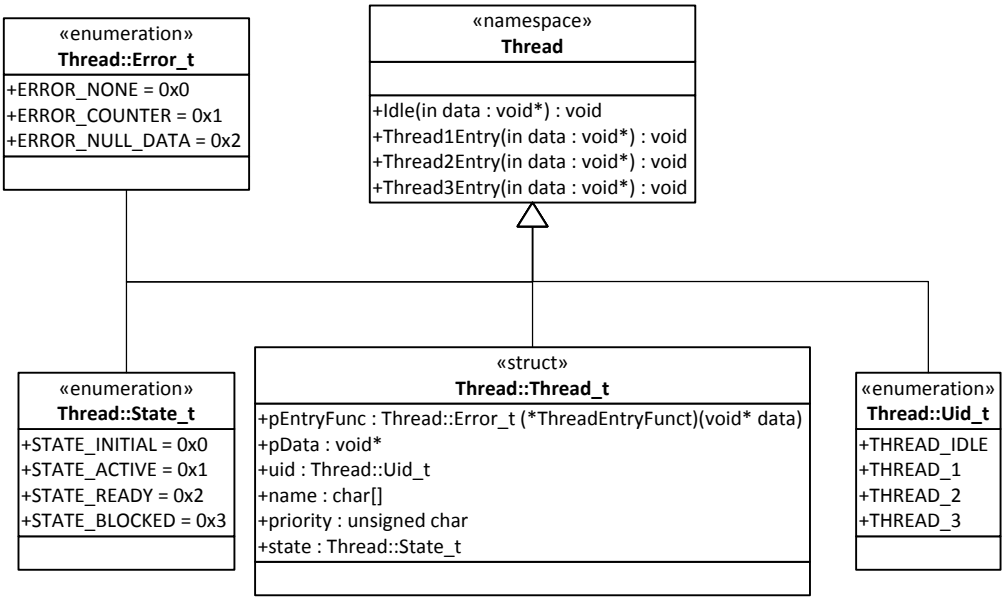

Figure 7- Thread namespace Diagram