Software Specifications & Design Document

Assignment 1, EMBSYS110 , Spring 2015
Chad Bartlett, cbartlet@uw.edu

1. Introduction

1.1. Project Overview

The purpose of this project is to implement a thread queue with three serialized threads (including one additional idle thread) and a shared variable. 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 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, a message is output through the serial-port and the 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 3.7 and 3.8 elaborates on the serial output and LED illumination patterns.

1.4. References

1.4.1. **UW Tiny Kernel – Assignment 1, A1/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.

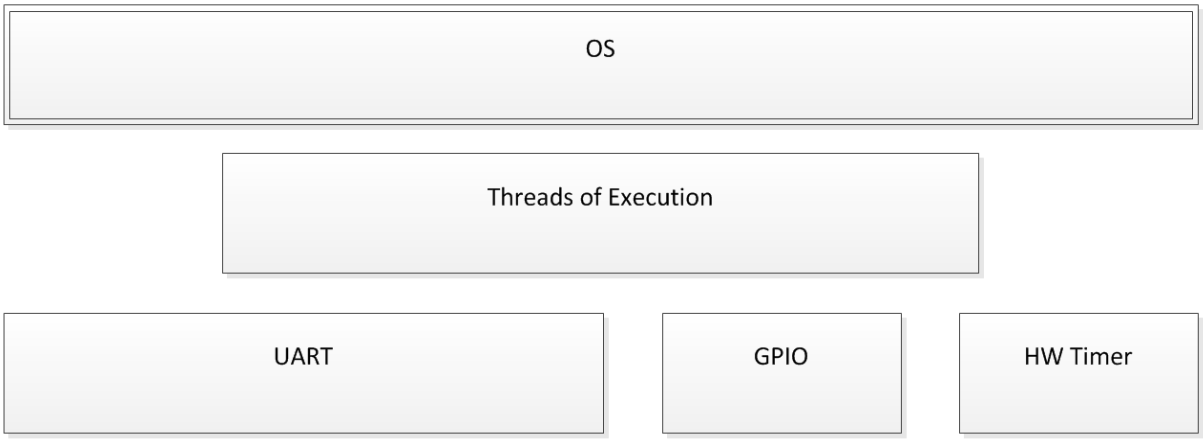

Figure 1: System block diagram.

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* respond to events generated by other threads.
SRS_THD_007	Each thread *must* contain an entry function. The entry function is the function called from the scheduler.
SRS_THD_008	Each thread entry function *should* have a unique name.
SRS_THD_009	The idle thread entry function *should* be called *idleEntry().*
SRS_THD_010	Each thread entry function *must* have the following signature: *uint32_t functionName(void* data)* (Where a return code of 0 means success otherwise an error code.)

Table 2: Data Requirements

SRS_DAT_001	A structure, *thread_t, must* be defined with the following fields:
	{
	UID // unique identification number
	NAME // fixed 12-character string
	PRIORITY // thread priority level
	STATE // thread state; Ex. INITIAL, ACTIVE, READY, BLOCKED
	PTR // function pointer to thread entry
	}

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 1000 Hz.
SRS_RUN_005	All thread state changes *must* be printed.
SRS_RUN_006	At the completion of the 'idle' thread, the state of the shared variable described in SRS_DAT_003 *must* be printed.

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 4.1 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. 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
        |- main.cpp
        |- mini_cpp.cpp
        |- no_heap.cpp
Bin/    -
        |- Assignment1_cbartlett.hex
        |- Assignment1_cbartlett.bin
Bsp/    -
        |- Bsp.cpp
        |- Bsp.h
        |- Interrupts.cpp
        |- startup_stm32f10x.cpp
        |- stm32f100.ld
        |- stm32f10x_conf.h
Doc/    -
        |- Assignment1_Design_cbartlett.doc
        |- Assignment1_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
        |- TestOS.cpp
        |- TestTemplate.cpp
./      -
        |- .ddd_gdb
        |- .nemiver_gdb
        |- README
        |- Makefile
        |- Makefile.Includes
        |- Makefile.Common
```

3.3. Scheduler

The scheduler executes a non-preemptive, round-robin pattern. Execution begins with the *idle* thread. Three additional threads execute sequentially. Once the last thread in the thread queue completes its tasks, the queue index wraps around to once again execute the *idle* thread, and so on. A visualization of this execution pattern is shown in Figure 2. Threads can be in one of four states: INITIAL, READY, ACTIVE and BLOCKED. However, in this implementation each thread runs to completion and therefore never enters the BLOCKED state. Also, all threads share a single stack. All OS procedures are handled with the OS class. See the class diagrams in Figure 3, Figure 5 and Figure 6 of Section 7.

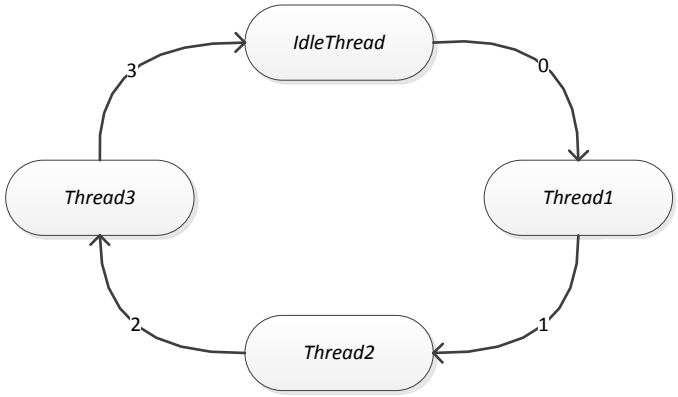

Figure 2 - Round-robin thread execution pattern.

3.4. Test Plan

The testing framework CppUTest is utilized to create tests that exercise the APIs of this application. These tests are executed 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 3.8.

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 7.

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 3.8 as well.

3.7. 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 7.

```
     > Chad Bartlett's Assignment 1.

     > Hardware initialization took: 5-milliseconds
(01) Name: Idle
(02) State: 0x2
     Name: Producer
     State: 0x2
     Name: Consumer
     State: 0x2
     Name: Monitor
     State: 0x2
(03) Inc: 0
(04) nT1: 0
(05) nT2: 0
(06) > Idle exec time: 10001-ms
     Name: Producer
     State: 0x2
(07) > Thread1 exec time: 1003-ms
     Name: Producer
(08) State: 0x1
     Name: Consumer
     State: 0x2
(09) > Thread2 exec time: 1002-ms
     Name: Consumer
     State: 0x1
     Name: Monitor
     State: 0x2
(10) > Thread3 exec time: 1002-ms
     Name: Monitor
     State: 0x1
(11) Inc: 0
     nT1: 1
     nT2: 1
     > Idle exec time: 10001-ms
     Name: Idle
     State: 0x1
```

(01) The name of the current thread
(02) The state of the current thread. State 0x2 corresponds to STATE_READY.
(03) Shared counter = 0. (Printed at the end of the Idle thread.)
(04) Times thread 1 has executed = 0. Shared counter = 0. (Printed at the end of the Idle thread.)
(05) Times thread 2 has executed = 0. Shared counter = 0. (Printed at the end of the Idle thread.)
(06) Execution time of the Idle thread.
(07) Execution time of the Thread1.
(08) State 0x1 corresponds to STATE_ACTIVE.
(09) Execution time of the Thread2.
(10) Execution time of the Thread3.
(11) State of shared data printed at the end of Idle.

3.8. LED Patterns

During normal operation, the blue LED flashes at a rate of 10 Hz while the green LED is cleared upon entry to the *Idle* thread and set upon exit. If the third thread detects an error with the shared data, the green LED flashes at a rate of 4 Hz. If an assertion has occurred both LEDs are steady on.

4. Appendices

4.1. Object Diagrams

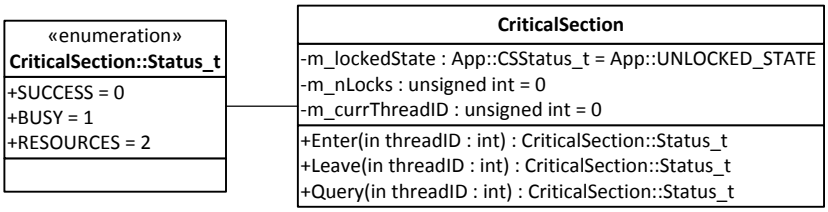

Figure 3 - Critical Section Class Diagram

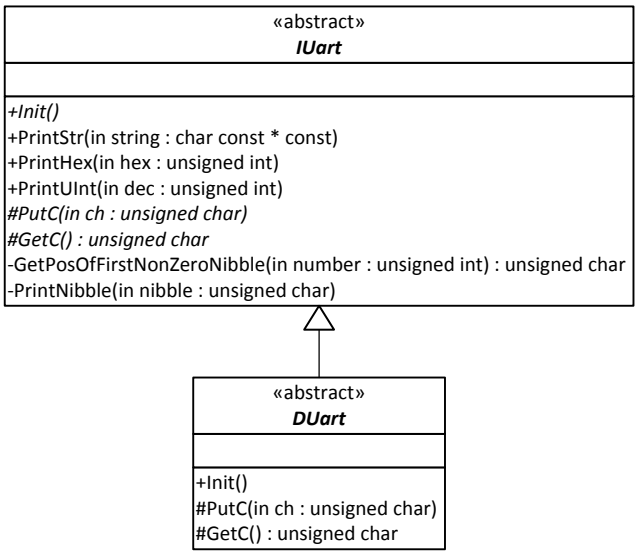

Figure 4- DUart Class Diagram

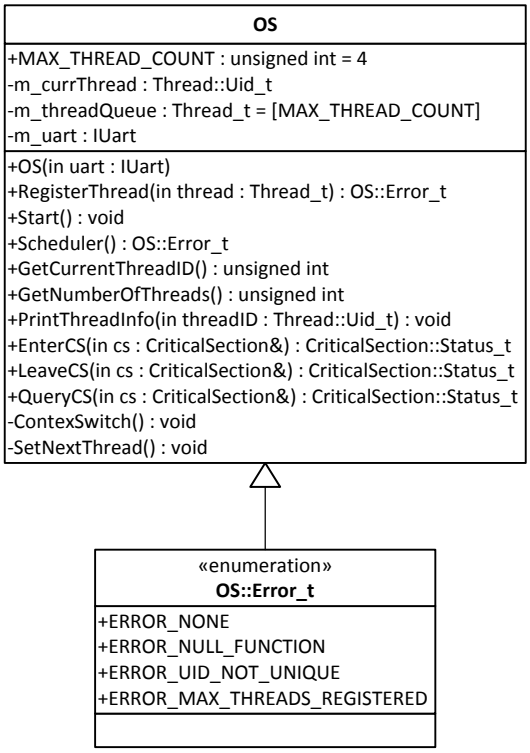

Figure 5- OS Class Diagram

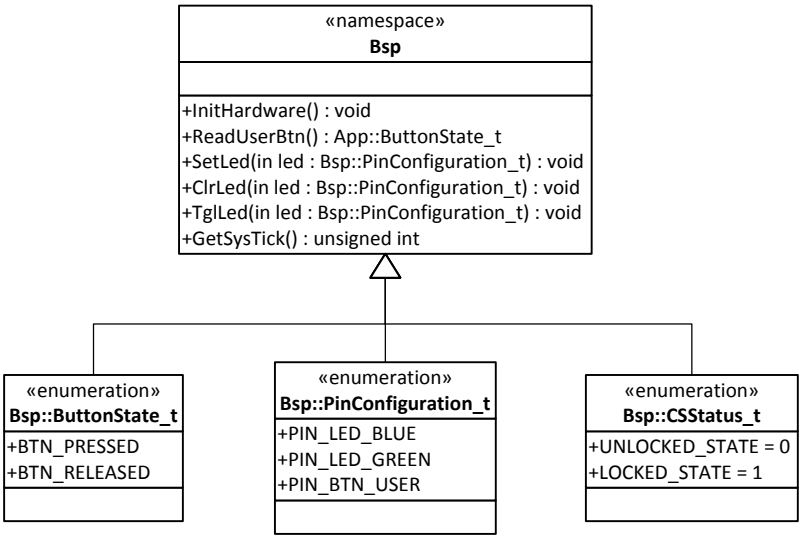

Figure 6 - Bsp namespace Diagram

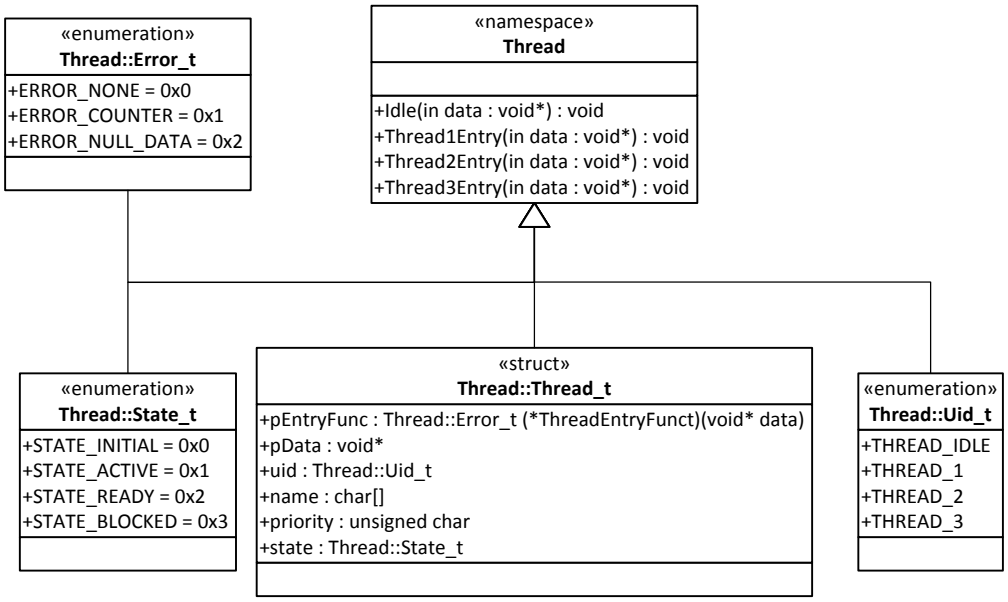

Figure 7- Thread namespace Diagram