

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. In Section 4 the expected serial output and LED patterns are listed. Finally, Section 5 lays out the test plan.

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.

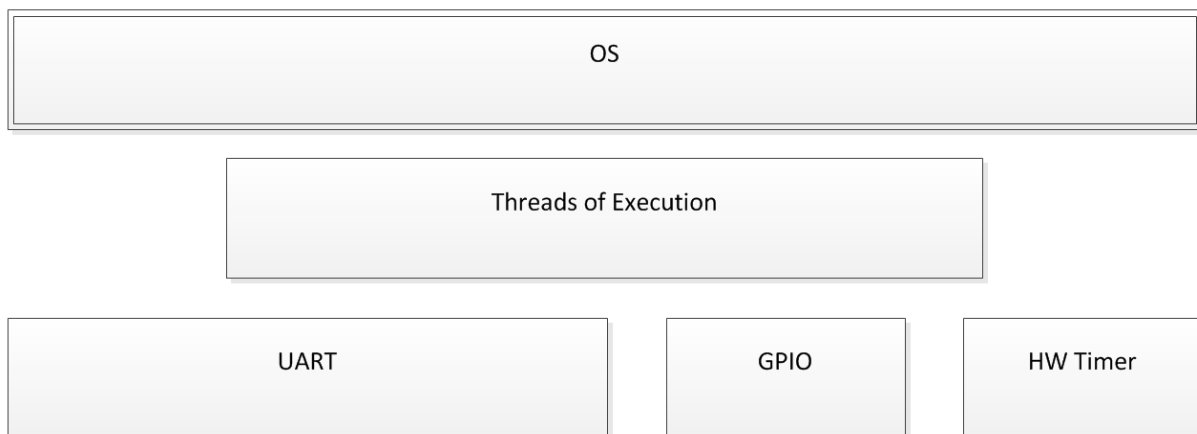


Figure 1: System block diagram.

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 <i>must</i> hold a minimum of four threads.
SRS_THD_002	One thread <i>must</i> be called <i>idle</i> .
SRS_THD_003	The idle thread <i>must</i> be the first thread in the thread queue.
SRS_THD_004	The threads <i>must</i> be executed forever using a round-robin schedule.
SRS_THD_005	Each thread <i>must</i> support data sharing.
SRS_THD_006	Each thread <i>must</i> respond to events generated by other threads.
SRS_THD_007	Each thread <i>must</i> contain an entry function. The entry function is the function called from the scheduler.
SRS_THD_008	Each thread entry function <i>should</i> have a unique name.
SRS_THD_009	The idle thread entry function <i>should</i> be called <i>idleEntry()</i> .
SRS_THD_010	Each thread entry function <i>must</i> have the following signature: <i>uint32_t functionName(void* data)</i> (Where a return code of 0 means success otherwise an error code.)

Table 2: Data Requirements

SRS_DAT_001	A structure, <i>threadstr</i> , <i>must</i> 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 <i>must</i> be defined to output the fields from the structure defined in SRS_DAT_001.
SRS_DAT_003	A structure, <i>globalDataStr</i> , <i>must</i> 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 <i>should</i> 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 <i>must</i> be defined globally.

Table 3: Communication Requirements

SRS_COM_001	Serial output <i>must</i> be enabled to provide feedback during run-time.
-------------	---

Table 4: Run-time Requirements

SRS_RUN_001	An application running 3 threads <i>must</i> be created as described in Reference 1.4.1, under Section <i>Design Requirements</i> .
SRS_RUN_002	If an error occurs, the error <i>should</i> be recorded and reported.
SRS_RUN_003	Serial output <i>should</i> be generated throughout the board initialization code to indicate progress.
SRS_RUN_004	A hardware timer <i>should</i> be implemented to provide a system tick for the application running at 100 Hz.
SRS_RUN_005	All thread state changes <i>must</i> be printed.
SRS_RUN_006	At the completion of the 'idle' thread, the state of the shared variable described in SRS_DAT_003 <i>must</i> be printed.

Table 5: Documentation Requirements

SRS_DOC_001	All source code <i>should</i> be documented using Doxygen with the following keywords where appropriate: <ul style="list-style-type: none"> • File comment block: @file, @brief, @author, @date • Function comment block: @brief, @param, @return • Inline code using: <code>/**< description */</code> or <code>/*!< description</code>
-------------	---

Table 6: Testing Requirements

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

3. Detailed Design

3.1. Section Overview

This section describes the build environment, scheduler, test plan, thread queue, error handling and serial output.

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/      -
          |- main.cpp
Bin/      -
          |- Assignment1_cbartlett.hex
          |- Assignment1_cbartlett.bin
Bsp/      -
          |- Bsp.cpp
          |- Bsp.h
          |- Interrupts.cpp
          |- startup_stm32f10x.cpp
          |- stm32f100.ld
          |- stm32f10x_conf.h
Lib/      -
          |- CriticalSection.cpp
          |- CriticalSection.h
          |- DUart.cpp
          |- DUart.h
Tests/    -
          |- stubs/      -
                        |- StubBsp.cpp
          |- Makefile
          |- RunAllTests.cpp
          |- TestCriticalSectionAPI.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 three states: 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.

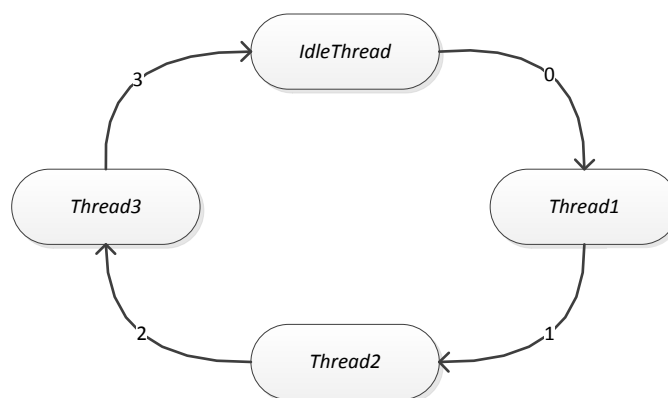


Figure 2 - Round-robin thread execution pattern.

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.

3.5. Thread Queue

The scheduler executes a non-preemptive round-robin pattern.

3.6. Error Handling

The scheduler executes a non-preemptive round-robin pattern.

3.7. Serial Output

The scheduler executes a non-preemptive round-robin pattern.

3.8. Component *n* Detail (include a sub-section for each component)

A structured description usually works. For example, if your components are classes you may wish to include the following subsections

3.8.1. Description

3.8.2. Data Members (include type, visibility, and description)

3.8.3. Methods (include English or pseudocode descriptions for each one)

4. **User Interface Design**

4.1. Section Overview

Provide a summary of the contents of this section. (1 – 2 paragraphs)

4.2. Interface Design Rules

Describe and justify the conventions and standards used to design your interface. You may be able to re-use some of the material prepared for CS 480 documents in this section. (1 – 2 paragraphs)

4.3. GUI Components

Note the GUI components or API's provided in the development environment that you plan on using. (1 paragraph + table)

4.4. Detailed Description

Provide a detailed description of the user interface including screen images. You may prefer to reference an appendix containing the screen snapshots. (1 – 4 pages)

5. **Test Plan**

5.1. Section Overview

5.2. Test Cases

5.3. Test Requirement Matrix

6. **Conclusion**

Provide an ending to this document with a mention of implementation and testing strategies resulting from this design (1 –2 paragraphs)

7. **Appendices** (a list of possibilities)

7.1. Database Entity-Relationship Diagram

7.2. Architectural Design Block Diagram(s)

7.3. Class Diagram(s)

7.4. Class Sequence Diagram(s)

7.5. User Interface Screen Snapshots