



Department of Electrical and Computer Engineering

## Lab Manual

# ECE 455: REAL TIME COMPUTER SYSTEMS DESIGN PROJECT

# Acknowledgement

This lab manual and the projects it contains is the result of the hard work of numerous individuals. Without their effort, time, and ideas over the years, it would not have been possible to provide ECE 455 students with the best possible learning experience.

The Department of Electrical and Computer Engineering would like to express its gratitude to these people who have contributed to the design projects in a major way: A. Jooya, S. Khoshbakht, N. Agarwal, S. Campbell, N.J. Dimopoulos, J. Dorocicz, K. Jones, T. Kroeker, E. Laxdal, K.F. Li, L. Palmer, N. Rebenich, T. Torres-Bonet, B. Zakeri, and D. Zhao.

In addition, the Department acknowledges the generous financial support from the Wighton Engineering Product Development Fund and the Curriculum Development Fund from the UVic Design Engineering Office.

# Table of Contents

<a href="#">Lab Overview</a> .....	1
<a href="#">Project 0: Introduction to TrueSTUDIO and FreeRTOS</a> .....	4
<a href="#">Project 1: Traffic Light System</a> .....	11
<a href="#">Project 2: Deadline-Driven Scheduler</a> .....	16

# Lab Overview

## 1 Introduction

The purpose of the ECE 455 lab is to introduce students to hardware and software for real time computer system design.

Three projects are completed throughout the semester across eight lab sessions:

- Project 0: Introduction to TrueSTUDIO and FreeRTOS
- Project 1: Traffic Light System
- Project 2: Deadline-Driven Scheduler

Project 0 is a foundational project meant to familiarize students with the lab environment and is completed in the first lab session. Projects 1 and 2 are relatively complex real time computer system design projects which are to be completed over three or four lab sessions each.

## 2 Lab Assessment

The lab will be assessed with the following distribution:

Project 0	5%
Project 1	45%
Project 2	50%

Project 0 is marked based on a project demonstration and is recorded as either Pass (5%) or Fail (0%). Projects 1 and 2 are marked based on a project demonstration as well as a project report. The complete mark breakdown is shown in Table 1, and the evaluation criteria is discussed in the following sections.

*Table 1: Mark breakdown for Project 1 and Project 2.*

	Project 0	Project 1	Project 2
<b>Demonstration</b>	5%	15%	20%
<b>Report</b>			
Format and Content	-	10%	10%
Design Document	-	10%	10%
Software Design and Implementation	-	10%	10%
<b>Total</b>	5%	45%	50%

### 2.1 Project Demonstration

All three projects must be demonstrated to the teaching assistant (TA) before the end of the lab session specified in the Lab Schedule. If a student fails to give a demonstration for a particular project, a mark of zero will be assigned for that project and a failing grade will be given for the course. Note that a project can be demonstrated even if it is not functioning completely or correctly. As part of the demonstration, you may be asked to:

- Demonstrate how your solution meets the project's requirements.
- Answer technical questions about which features of FreeRTOS are used and why.
- Discuss limitations of your system (e.g. why such limitations exist, how to overcome them).

Full marks will be awarded if the system functions correctly and meets all specified requirements. The number of features successfully implemented and whether any design flaws exist will be considered in the evaluation. System functionality can be demonstrated by performing suitable tests.

## 2.2 Project Report

A project report is required for each of Project 1 and Project 2; no report is required for Project 0. **Submit the report in PDF format to your TA in the Brightspace website no later than one week after completing the demonstration.** Note that a report will not be accepted if a demonstration has not been given.

### 2.2.1 Format and Content

The project report is a formal technical engineering document and should be written such that a person familiar with the topic can understand and recreate your design solution. The report should include the following:

- Introduction
- Design Solution
- Discussion
- Limitations and Possible Improvements
- Summary
- Appendix (with source code)

The Design Solution section is the main body of the report and should include details about the approach used for the design as well as technical details about the implementation. It should also include diagrams, flow charts, pseudocode, C code snippets, etc., and should make reference to specific FreeRTOS features used such as tasks, queues, timers, etc.

### 2.2.2 Design Document

A one-page design document must be created before writing any code. The design document may consist of simple diagrams or text which specifies how software and hardware components will behave and interact. Note that the design document is not intended to be a rigid specification and the final design is expected to differ from it somewhat. **Include your design document in the report** and discuss any differences that exist between it and the final design.

### 2.2.3 Software Design and Implementation

All source code must be included in the Appendix section of the report for evaluation by the TA. Source code must be written in C and must use good coding style. Meaningful variable and function names should be used (e.g., use `counter` instead of `ct`), sufficient comments and documentation should be included, and the design should be modular and easily extendable. Poor coding practices should also be avoided (e.g., avoid `GOTOs`). Discuss any known bugs, the test methods used, and any problems encountered while implementing specific features.

### 3 Lab Schedule

The following lab schedule will be followed:

*Table 2: Lab schedule.*

Lab Session	Lab Focus	Deliverable Due (Before End of Lab)
1	Project 0	Project 0 Demo
2	Project 1	
3	Project 1	
4	Project 1	Project 1 Demo
5	Project 2	Project 1 Report
6	Project 2	
7	Project 2	
8	Project 2	Project 2 Demo
		Project 2 Report

### 4 Additional Information

- Projects must be completed individually.
- No late submissions for Project 0 will be accepted.
- A grade penalty of 10% per day will be applied to late reports for Project 1 or Project 2. Deliverables that are more than three days (72hrs) late will receive a grade of zero and a failing grade for the course will be given.

### 5 Lab website

The lab website is [labs.engr.uvic.ca](https://labs.engr.uvic.ca). You can find ECE455 under “Electrical and Computer Engineering Specialized Equipment Labs”. Each system you connect to has a unique ID. The monitor application (figure 7) shows where you can find the ID

# Project 0: Introduction to TrueSTUDIO and FreeRTOS

## 1 Objective

The objective of this project is to introduce the Atollic TrueSTUDIO integrated development environment (IDE) for real time application development with FreeRTOS and the STM32F4 Discovery Board. TrueSTUDIO will be used to create, compile, build and debug a simple real time project and some basic concepts of real time operating systems will be explored.

## 2 Development Environment

The following hardware and software are used in the lab for developing real time applications:

- **STM32F4 Discovery Board**  
The STM32F4 is a high-performance microcontroller and the discovery board is used to leverage its capabilities and facilitate development.
- **FreeRTOS**  
FreeRTOS is a real time operating system kernel that provides methods for real time applications such as tasks, mutexes, and software timers. It is written in C.
- **Atollic TrueSTUDIO for STM32**  
TrueSTUDIO is the software development platform that is used to build, run, and debug FreeRTOS applications for the STM32F4 discovery board.

The computers in the lab run Windows and should have Atollic TrueSTUDIO for STM32 already installed. Note that TrueSTUDIO is built on Eclipse and therefore its interface and menus may already be familiar.

## 3 Readings

Prior to starting this project, browse through the following material:

- <https://www.freertos.org/>
- [Mastering the FreeRTOS Real Time Kernel – a Hands On Tutorial Guide](#)
- [FreeRTOS V10.0.0 Reference Manual](#)

Read the following sections of *Mastering the FreeRTOS Real Time Kernel – a Hands On Tutorial Guide*:

- Preface
- Task Management (Sections 3.1 to 3.7)
- Queue Management (Sections 4.1 to 4.3)

## 4 Procedure

There is a single template project that you will be using as the starting point for each lab experiment.

1. Log on to one of the computers in the lab.
2. Download the project template from the Brightspace website (Lab1\_Code.zip) and make sure to copy it to **C:\Users\YOURNAME\Desktop** folder.
3. Run Atollic TrueSTUDIO for STM32 using the  icon shortcut in the ECE455 folder on the desktop. Make note of the workspace path<sup>1</sup> and click OK.
4. Click on the **File** menu, select **Open Projects from File System**, then select **Archive...** on the top

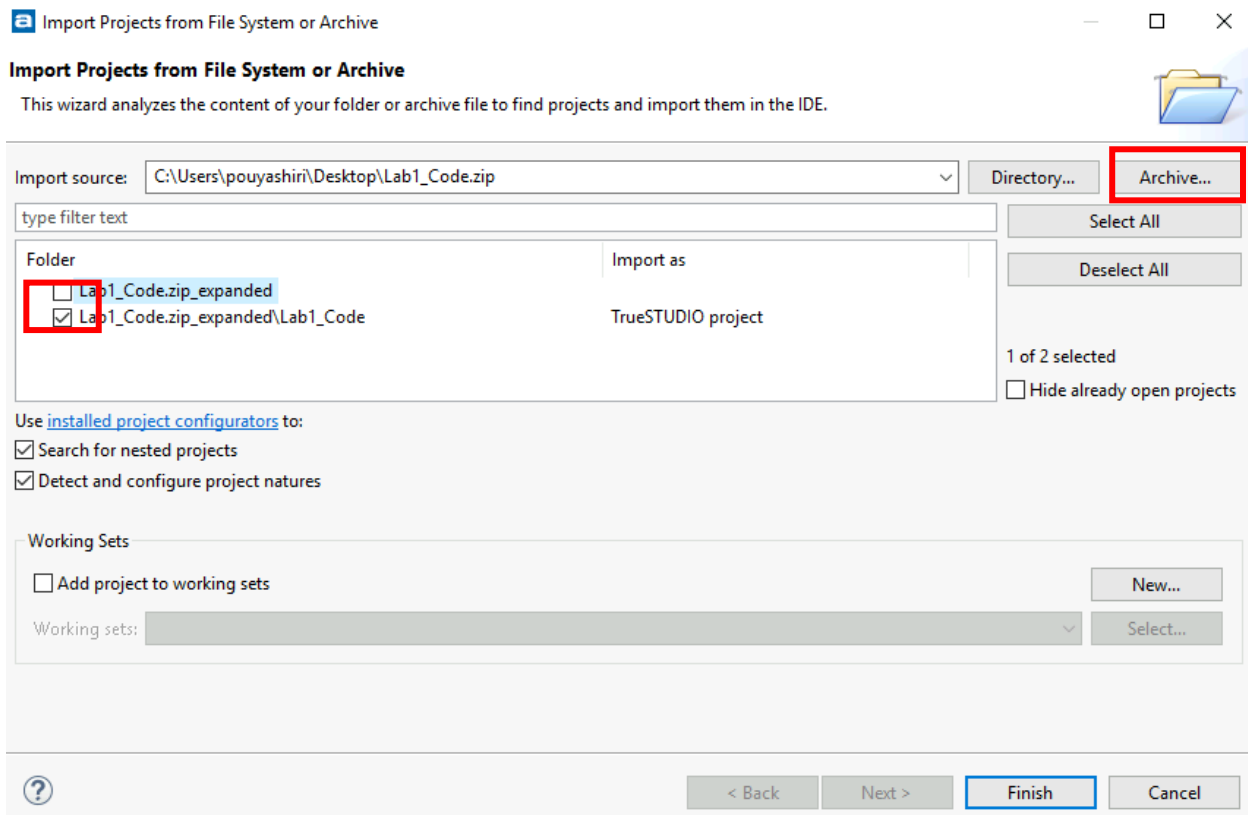


Figure 1 Importing a sample FreeRTOS project from the template

right corner to browse to **C:\Users\YOURNAME\Desktop** to import the project.

5. Check the folder with “TrueSTUDIO project” in the “Import As” column. Click on “Finish” to import the project.
6. Build the project and make sure that the build process is successful.

<sup>1</sup>Always set the workspace path to be on the “C” drive. At the end of each lab session, it is recommended to save a backup of the project folder to your personal “M” network drive (which is accessible from any lab computer). However, a backup project folder should always be copied back to the “C” drive before continuing development since setting the workspace path to a folder on the “M” drive can create issues.



7. Examine the project code in **main.c**. Notice the different tasks communicating with each other and the delays and timeouts used in each task. Ask your TA if you have any questions.
8. To set up the debugger, click on the **Run** menu and select **Debug Configurations**. On the left panel, under **Embedded C/C++ Application**, there is a debug configuration with the same name as the project's. Set the configuration as shown in Figure 3.

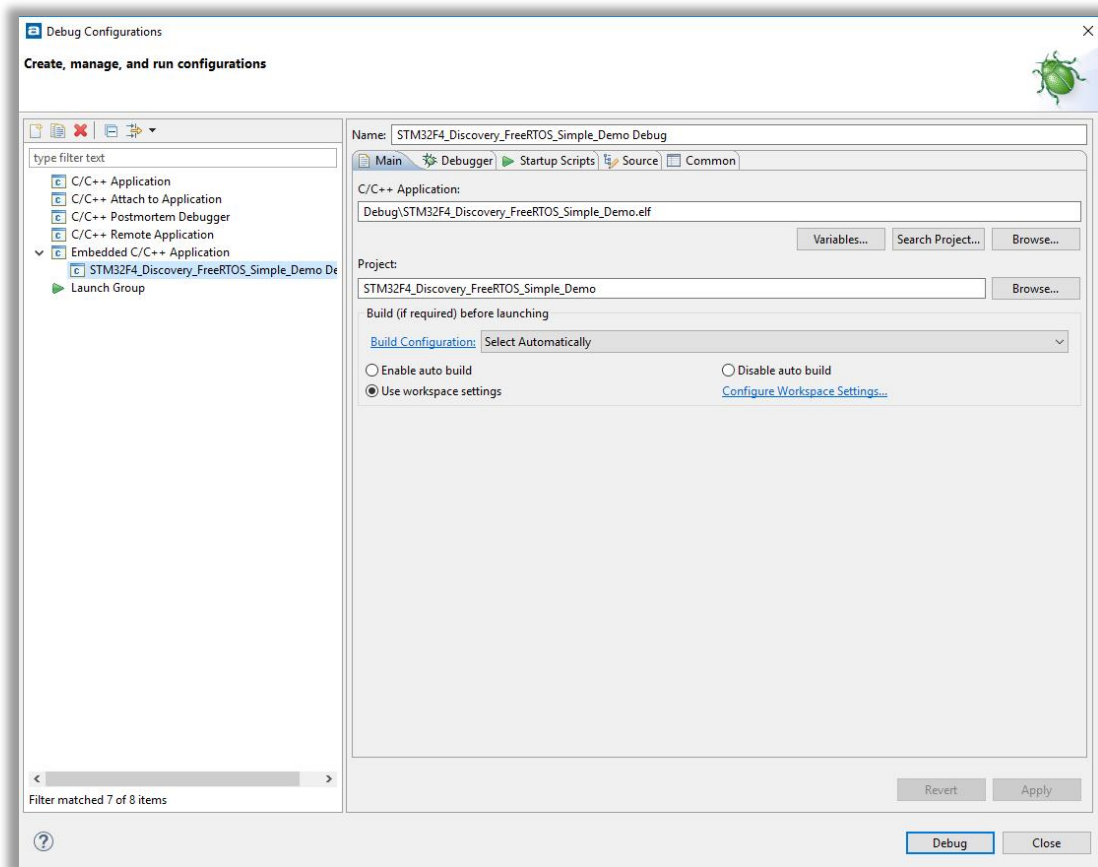


Figure 3: Debug configuration window.

9. Select the Debugger tab and check **Enable** under **Serial Wire Viewer (SWV)** as shown in Figure 4.

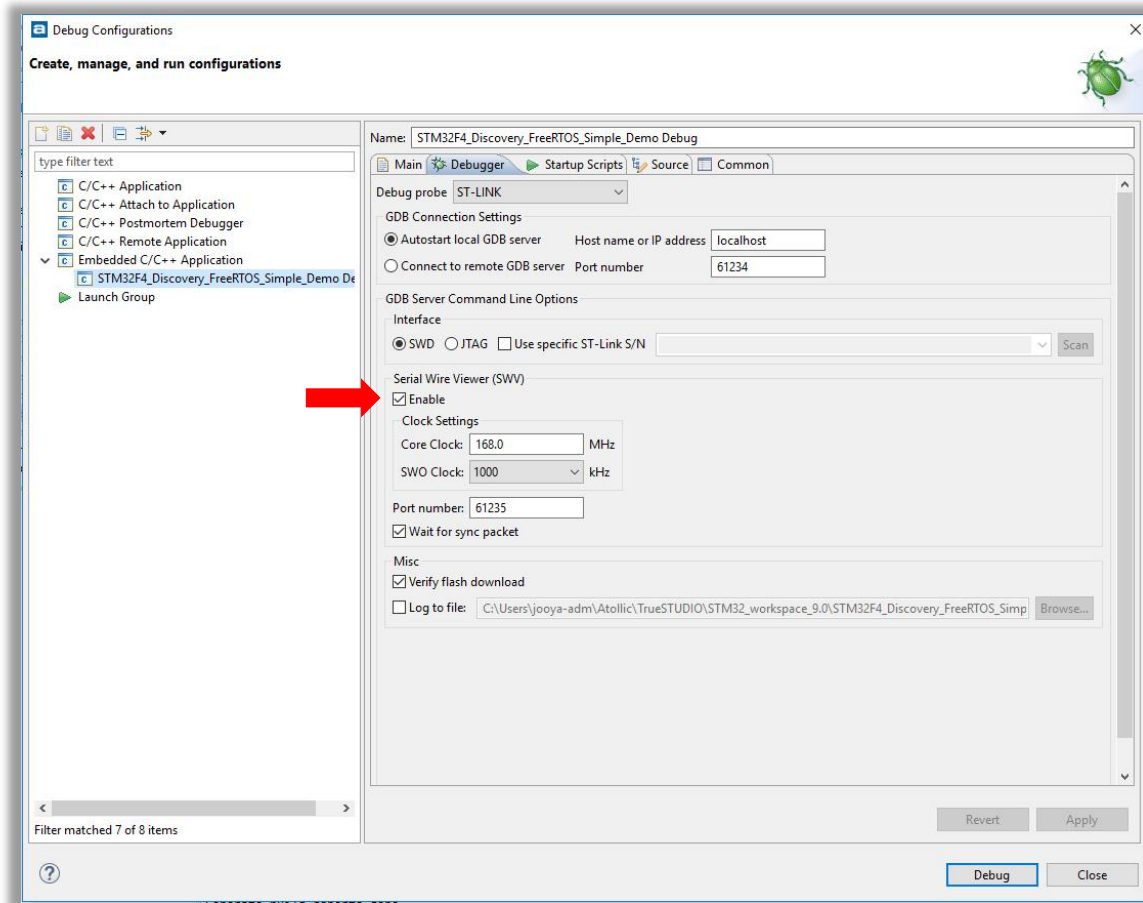


Figure 4: Enabling the Serial Wire Viewer.

10. Check the other tabs and make yourself familiar with the debugger options (do not change other settings, however). Click **Debug** and observe that TrueSTUDIO is now in debug mode.
11. From the **View** menu, enable the following options:
- Select **FreeRTOS** and enable **FreeRTOS Queues**.
  - Select **FreeRTOS** and enable **FreeRTOS Task List**.
  - Select **SWV** and enable **SWV Console**.

You can check the status of queues and tasks using the views you enabled. You can use this for debugging purposes. You may use the **printf** command to print to the SWV Console. Note that the printing operation includes an overhead, so try to minimize the information that you would like to print. The debug interface should now be similar to the one shown in Figure 5.

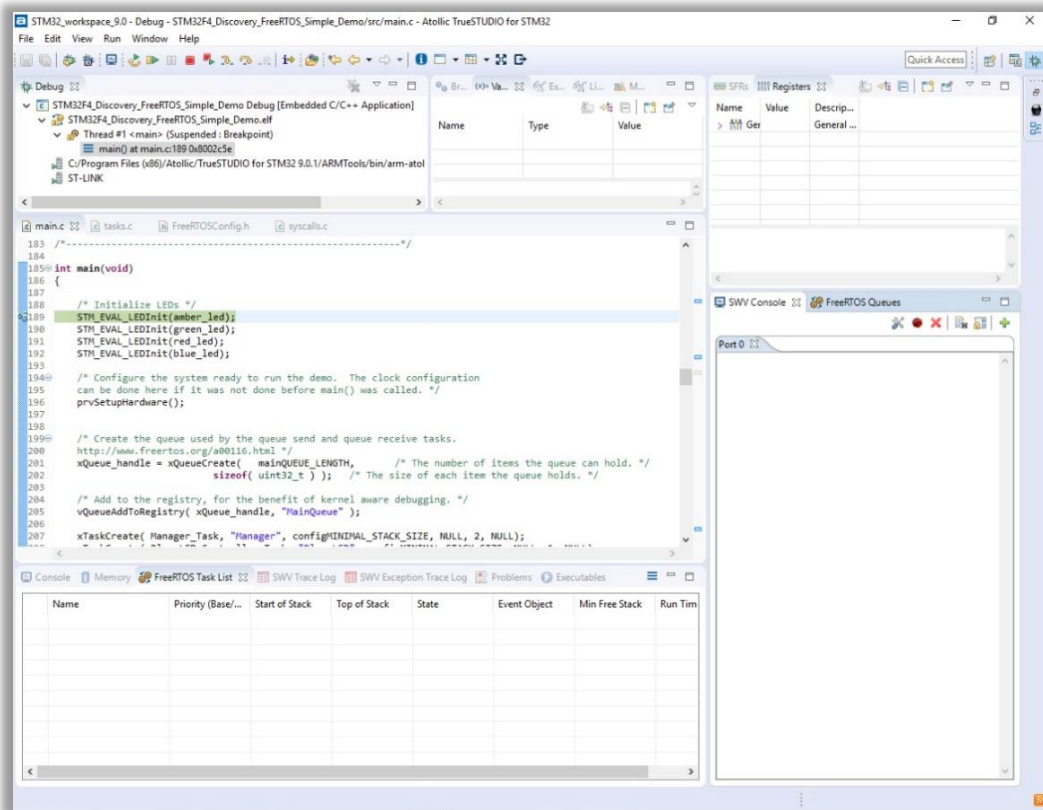



Figure 5: Debugger view with SWV console and FreeRTOS queue and task list enabled.

12. Click on the tool icon  in the **SWV Console**.
13. Check the **Port0** box under **ITM Stimulus Ports** as shown in Figure 6. Click **OK** to close the window.

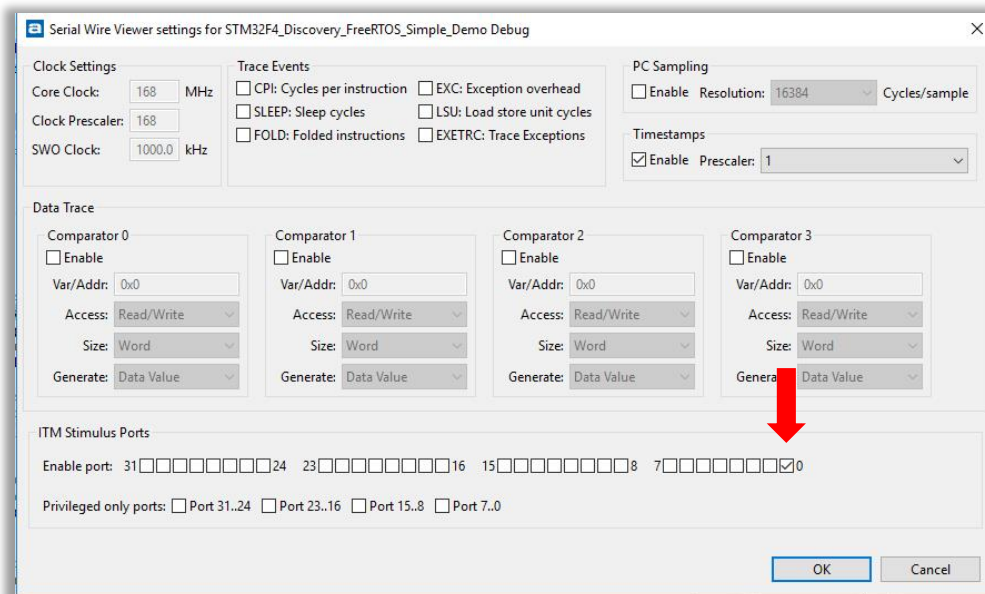



Figure 6: Enabling Port0 of ITM.

14. Set breakpoints in different tasks before and after the print statements of queue access instructions, then click on the red button  on the **SWV Console** panel and run the debugger.
15. Step through the code and observe the print messages on the **SWV Console**, the number of elements in the **FreeRTOS Queues** panel, and how the program switches between tasks in the **FreeRTOS Task List** panel.
16. Validate your understanding of the system's behaviour by experimenting with different timing values for the different task and queue functions.

## 5 Deliverables

1. Demonstrate the running project to the TA.
2. Explain the code and the output in the SWV Console.

# Project 1: Traffic Light System

## 1 Objective

The objective of this project is to design and implement a Traffic Light System (TLS) using middleware, and FreeRTOS features such as tasks, queues, and software timers.

## 2 Description

The TLS simulates vehicle traffic on a one-way, one-lane road with a simplified intersection that has a single traffic light. The system has three main components:

1. **The traffic flow adjustment potentiometer**

The potentiometer is used to dynamically adjust the traffic flow rate (i.e. the number of cars created per second) at run-time.

2. **LEDs representing cars**

The LEDs are used to represent the position of cars at different points in time. If an LED is on, it means a car is on the road at that LED's position; if an LED is off, it means there is no car at that LED's position. When a car is on the road, it can "move" by shifting over one position to the right; this is achieved by toggling the appropriate LEDs on or off. More than one car can exist on the road at a time.

3. **Traffic light**

A traffic light with three LEDs (one green, one yellow, one red) is used to control the traffic at the intersection.

The main components of the TLS in the user interface that will be used, are visualized in Figure 7.

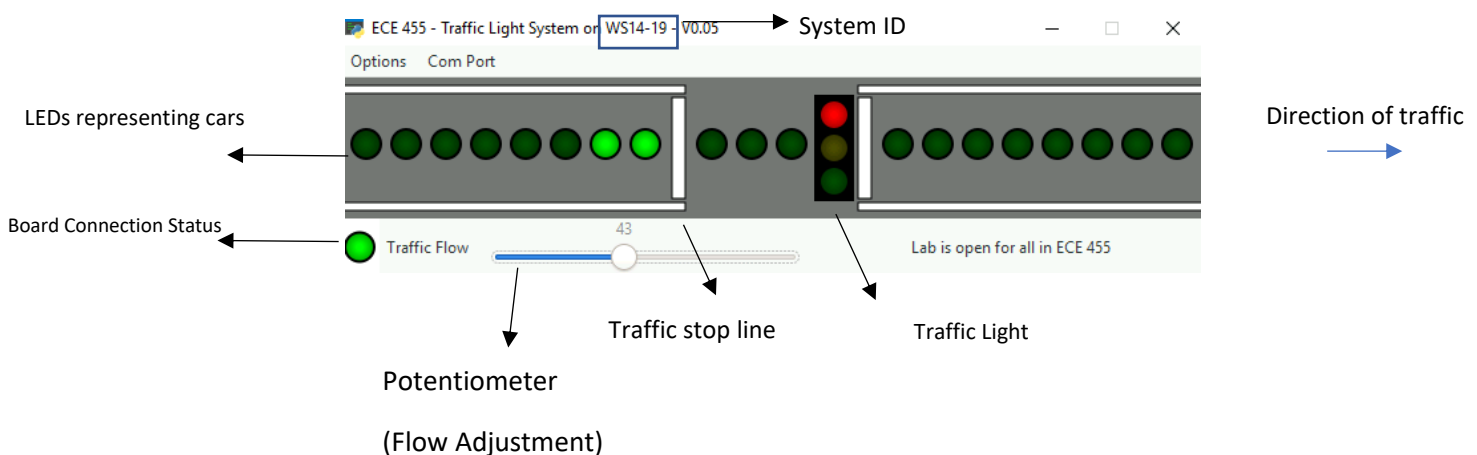


Figure 7: The traffic light system User Interface.

A video of the TLS (Traffic\_Light\_System\_Demo.mp4) is available on CourseSpaces which demonstrates a fully functional project.

## 2.1 Hardware

The hardware is prepared and connected on the server-side for each computer. There are several pins available to work with the online board. Figure 1 shows a list of these pins. There are 3 pins for the traffic lights, a single pin for the potentiometer input and 3 pins for the shift register (to load the traffic state).

There are three daisy-chained shift registers to act as a single serial-to-parallel converter (SPC). SPCs are needed to reduce the number of electrical connections with the discovery board because it has too few general-purpose input/output (GPIO) pins to map every LED directly to it. Here, 3 shift registers are stacked together to increase the number of outputs using a single input.

ECE 455 Traffic Light Signals

STM32F0 Port Pin	Signal	Notes
PC0	Red Light	Traffic Light
PC1	Amber Light	Traffic Light
PC2	Green Light	Traffic Light
		Traffic flow shift register
PC8	Shift Register Reset	Active Low, minimum 1us period
PC7	Shift Register Clock	Falling edge trigger
PC6	Shift Register Data	After clock falling edge data hold time of a minimum of 1 us
PC3	Potentiometer input	0 – 3 Volt Input

Figure 2. Available pins on the server

## 2.2 Middleware

Middleware is typically software that defines the interactions between application software and the device drivers. It is an abstraction layer generally used on embedded devices which have multiple applications in order to provide flexibility, security, portability, connectivity, intercommunication, and/or interoperability mechanisms between them<sup>2</sup>. The layers of an embedded system are modelled in Figure 8.

<sup>2</sup> Tammy Noergaard, "Guide to Embedded Systems Architecture - Part 1: Defining middleware." Online: [https://www.eetimes.com/document.asp?doc\\_id=1276764](https://www.eetimes.com/document.asp?doc_id=1276764)



Figure 8: Embedded system model<sup>2</sup>.

In Project 0, middleware developed by ST was included in the sample project code that was used. The source files can be found in **Project Explorer -> Libraries -> STM32F4xx\_StdPeriph\_Driver -> src**, and the source file for the analog-to-digital converter (ADC) is **stm32f4xx\_adc.c**. Note that not all functions in this file are required by every program.

For the TLS project, middleware must be written to configure the STM32's GPIO pins and the ADC to allow application code to interact with hardware components. This may require enabling peripheral clocks, defining `InitTypeDef` structs for initializing both GPIO pins and the ADC, and writing a function to read the current value of the ADC. Note that the TLS middleware can make use of existing Application-Programming Interfaces (APIs) and therefore it is not necessary to work directly with registers.

## 2.3 Application Software

Application code must be written using FreeRTOS tasks, queues, and timers to manage system resources and simplify hardware and software interactions. Although it may be possible to write the TLS software with a single-threaded application, this is not acceptable for this project.

The following tasks are recommended for the TLS:

1. **Traffic Flow Adjustment Task**

The traffic flow that enters the intersection is set by a potentiometer. This task reads the value of the potentiometer at an appropriate interval and sends its value to other tasks. A low potentiometer resistance corresponds to light traffic and a high resistance corresponds to heavy traffic.

2. **Traffic Generator Task**

This task randomly generates new traffic with a rate that is proportional to the potentiometer's value; this value is received from the Traffic Flow Adjustment Task. The generated traffic is then sent to another task so that it can be displayed on the road. Note that traffic can be represented in many ways (e.g., a single integer, an array, or a struct).

3. **Traffic Light State Task**

This task controls the timing of the traffic lights and outputs its current state (i.e. green state, yellow state, or red state). The timing of the lights is affected by the load of the traffic which is received from the Traffic Flow Adjustment Task. An increase in traffic should increase the duration of the green light and decrease the duration of the red light.

#### 4. *System Display Task*

This task controls all LEDs in the system and is responsible for visualizing all vehicle traffic and the traffic lights. It receives information from the Traffic Generator Task as well as the Traffic Light State Task and controls the system's LEDs accordingly. This task also refreshes the car LEDs at a certain interval to emulate the flow of the traffic.

Tasks other than those described above may be used, but any differences must be highlighted and justified to the TA and discussed in the report.

### 3 Requirements

#### 3.1 Functional Requirements

- A. The TLS must have a similar construction to that shown in Figure 7.
- B. Traffic must be generated randomly and at a rate that is directly proportional to the resistance of the potentiometer.
- C. All cars must “move” at a constant speed of approximately 1-3 LEDs per second (unless stopped at a red light – speed is zero in this case).
- D. Cars can only proceed through the intersection when the light is green and must stop at the traffic stop line if the light is yellow or red.
- E. The duration of the green traffic light must be directly proportional to the traffic flow rate.
- F. The duration of the red traffic light must be inversely proportional to the traffic flow rate.
- G. The duration of the yellow traffic light must be constant.
- H. When the traffic flow is at its maximum setting:
  - There should be no gaps between new cars appearing on the road (i.e. bumper-to-bumper traffic).
  - The traffic light must stay green approximately twice as long as it stays red.
- I. When the traffic flow is at its minimum setting:
  - Traffic should be created with a gap of approximately 5 or 6 LEDs between cars.
  - The traffic light must stay red approximately twice as long as it stays green.

#### 3.2 Technical Requirements

- A. Tasks must be used to control code execution.
- B. Queues must be used for inter-task communications. Global variables may NOT be used for this purpose.
- C. Software timers must be used to control the state of the traffic lights.

### 4 Recommended Steps

The following steps are recommended for completing this project:

1. Create a design document which considers what tasks, queues, functions, etc. will be needed and how these components will interact. **Include a copy of the design document in your report.**
2. Build a circuit and write code to light and “move” all car LEDs.
3. Build a circuit and write code to interface with the potentiometer.
4. Build a circuit and write code to operate the traffic light.
5. Write code to fully communicate all necessary information between tasks.
6. Refine the system to meet all requirements.



## 5 Notes

- This project is extremely difficult if not impossible to ‘hack’ together. Therefore, be sure to create a design *before* writing any code since failing to do so will likely increase the time required to finish the project.
- You should expect to spend at least 20 hours designing, implementing, and testing the project.

## 6 Deliverables

1. Demonstrate your project to the TA.
2. Submit a project report. Include the following:
  - System overview diagram which shows the relationship between tasks, and queues.
  - Description and discussion of middleware, including:
    - A list of all GPIO pins used and their function.
    - Implementation details.
  - Description and discussion of software.
  - A flow chart of the traffic generating algorithm.
  - A flow chart of the system display algorithm.

# Project 2: Deadline-Driven Scheduler

## 1 Objective

The objective of this project is to design a custom Deadline-Driven Scheduler (DDS) to dynamically manage tasks which have hard execution deadlines.

## 2 Description

The DDS uses Earliest Deadline First (EDF) scheduling to maximize processor utilization. Since FreeRTOS is not sufficiently configurable to natively perform deadline-driven scheduling, the DDS must be built on top of the existing FreeRTOS task scheduler. In order to differentiate between the different tasks managed by the two schedulers, a task managed by the DDS will be referred to as a Deadline-Driven Task (DD-Task) whereas a task managed by FreeRTOS will be referred to as a FreeRTOS Task (F-Task). A DD-Task is not a “real” task but is simply a data structure which holds the handle of a corresponding user-defined F-Task along with the necessary information for EDF scheduling.

The scheduling mechanism to be implemented by the DDS is based on dynamically changing the priorities of user-defined F-Tasks from an actively-managed list of periodically- or aperiodically-generated DD-Tasks. If a DD-Task needs to be scheduled and its deadline is the earliest, the DDS will set the priority of the F-Task referenced by the DD-task to ‘high’ and set the priorities of all other F-Tasks referenced by the other DD-tasks in the list to ‘low’. Effectively, this will cause the native FreeRTOS scheduler to only execute a single user-defined F-Task (i.e. the one with the earliest DD-Task deadline) since this F-task will now have a higher priority for execution over other user-defined F-tasks.

To test the functionality of the DDS, auxiliary F-Tasks will need to be written. These include User-Defined Tasks, a Deadline-Driven Task Generator, and a Monitor Task. Note that the auxiliary F-tasks solely exist to test the system and are independent of the DDS. The F-Tasks which will need to be implemented for this project are:

1. **Deadline-Driven Scheduler**  
Implements the EDF algorithm and controls the priorities of user-defined F-tasks from an actively-managed list of DD-Tasks.
2. **User-Defined Tasks**  
Contains the actual deadline-sensitive application code written by the user.
3. **Deadline-Driven Task Generator**  
Periodically creates DD-Tasks that need to be scheduled by the DD Scheduler.
4. **Monitor Task**  
F-Task to extract information from the DDS and report scheduling information.

### 2.1 Deadline-Driven Scheduler

#### 2.1.1 DD-Tasks

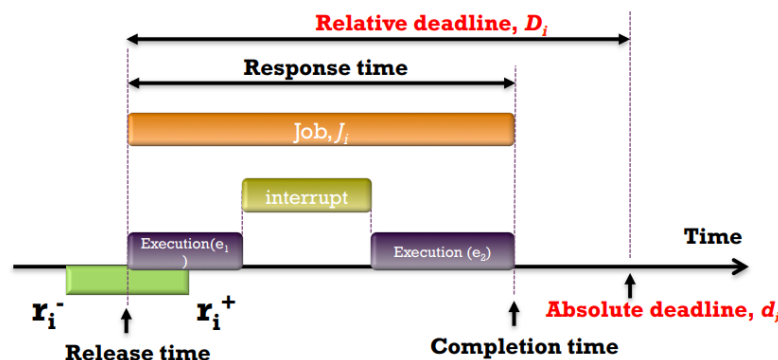
Internally, the DDS uses a DD-Task data structure shown in Figure 9. This structure contains the task handle of the corresponding User-Defined Task along with relevant information for performing EDF calculations. A DD-Task’s type is also stored since it may be periodic or aperiodic. A task identification number is given to allow the DDS to easily identify the DD-Task when performing its various functions.

```
enum task_type {PERIODIC,APERIODIC};

struct dd_task {
    TaskHandle_t t_handle;
    task_type type;
    uint32_t task_id;
    uint32_t release_time;
    uint32_t absolute_deadline;
    uint32_t completion_time;
}
```

Figure 9: DD-Task structure.

The values stored for `release_time`, `absolute_deadline`, and `completion_time` should be consistent with the real time definitions shown in Figure 10. Note that the deadline should be treated as 'hard'. Consider extending `dd_task` to incorporate additional information (e.g. list of interrupt times) as it may be useful for debugging or for implementing the Monitor Task. Note that the actual execution time of the DD-Task is defined by the application code (which executes within the referenced F-Task) and is therefore not known to the DDS.

Figure 10: Real time task definitions<sup>3</sup>.

### 2.1.2 DD-Task Lists

The DDS organizes DD-Tasks into three lists:

1. **Active Task List**  
A list of DD-Tasks which the DDS currently needs to schedule.
2. **Completed Task List**  
A list of DD-Tasks which have completed execution before their deadlines.
3. **Overdue Task List**  
A list of DD-Tasks which have missed their deadlines.

The Active Task List will need to be sorted by deadline every time a DD-Task is added or removed from the list and an appropriate data structure and a sorting algorithm will need to be selected and implemented for this purpose. The following singly-linked list structure is recommended:

<sup>3</sup> ECE 455 Lecture Notes.

```

struct dd_task_list {
    dd_task task;
    struct dd_task_list *next_task;
}

```

Figure 11: Recommended structure for DD-Task lists.

DD-Tasks which successfully complete their execution before their deadline must be removed from the Active Task List and added to the Completed Task List. DD-Tasks which do not meet their deadlines must be removed from the Active Task List and added to the Overdue Task List.

Note that the Completed Task List is primarily used for debugging, testing, and reporting purposes as it contains a useful summary of the system's events. In practice, however, the use of a Completed Task List is not recommended since the list uses valuable memory space and does not contain any relevant information for scheduling active tasks.

### 2.1.3 Core Functionality

The DDS is an F-Task with the highest priority and is normally suspended unless a call is made to one of its interface functions:

1. **release\_dd\_task**  
This function receives all of the information necessary to create a new `dd_task` struct (excluding the release time and completion time). The struct is packaged as a message and sent to a queue for the DDS to receive.
2. **complete\_dd\_task**  
This function receives the ID of the DD-Task which has completed its execution. The ID is packaged as a message and sent to a queue for the DDS to receive.
3. **get\_active\_dd\_task\_list**  
This function sends a message to a queue requesting the Active Task List from the DDS. Once a response is received from the DDS, the function returns the list.
4. **get\_completed\_dd\_task\_list**  
This function sends a message to a queue requesting the Completed Task List from the DDS. Once a response is received from the DDS, the function returns the list.
5. **get\_overdue\_dd\_task\_list**  
This function sends a message to a queue requesting the Overdue Task List from the DDS. Once a response is received from the DDS, the function returns the list.

Note that the DD-Task lists can be returned either by reference or by value. Analyze the options and justify the design choice.

The recommended declarations for these functions, respectively, are:

1. `void create_dd_task( TaskHandle_t t_handle,  
task_type type,  
uint32_t task_id,  
uint32_t absolute_deadline,  
);`
2. `void delete_dd_task(uint32_t task_id);`
3. `**dd_task_list get_active_dd_task_list(void);`

4. `**dd_task_list get_complete_dd_task_list(void);`
5. `**dd_task_list get_overdue_dd_task_list(void);`

Since the DDS is independent of the auxiliary tasks, auxiliary tasks should only access the DDS using the interface functions. Furthermore, auxiliary tasks should not have access to any internal data structures used by the DDS.

Each function sends a message to a queue and then resumes the DDS; since the DDS has the highest priority, FreeRTOS will now execute the DDS's code without interruption. The DDS should receive the message from the queue, determine what type of message it is, and then act accordingly based on the following:

1. **Message from `release_dd_task`**  
The DDS should assign a release time to the new task, add the DD-Task to the Active Task List, sort the list by deadline, then set the priorities of the User-Defined Tasks accordingly.
2. **Message from `complete_dd_task`**  
The DDS should assign a completion time to the newly-completed DD-Task, remove the DD-task from the Active Task List and add it to the Completed Task List. The DDS should also sort the Active Task List by deadline and then set the priorities of the User-Defined Tasks accordingly.
3. **Message from `get_active_dd_task_list`**  
The DDS should send the Active Task List to a queue.
4. **Message from `get_completed_dd_task_list`**  
The DDS should send the Completed Task List to a queue.
5. **Message from `get_overdue_dd_task_list`**  
The DDS should send the Overdue Task List to a queue.

Consideration must be given to exactly when DDS function calls are made and how the system will behave if a task has missed its deadline. For example, a software timer may be used internally by the DDS to issue a callback when a deadline has been exceeded and then move the late DD-Task to the Overdue Task List (depending on the implementation, this may also require an additional message type for the DDS to consider internally). With periodic DD-Tasks, however, this may be redundant since the DDS is already invoked when a task is created and a check for overdue tasks could be performed at this time. In specific circumstances, such as when multiple DD-Tasks need to be created at the exact same time, task creation events may experience a delay (i.e. "jitter") which could result in a delay to the detection of an overdue DD-Task. Therefore, consideration must also be given to the order and timing of DD-Task release, completion, and overdue events.

## 2.2 User-Defined Tasks

A User-Defined Task is an auxiliary F-Task that contains the actual deadline-sensitive application code which will be executed at run-time. Its code is fully self-contained and must not rely on any communication with other F-Tasks to complete its execution.

A User-Defined Task would normally perform something useful such as a calculation; for this project, however, it may simply execute an empty loop for the duration of its execution time. It is recommended to make use of the LEDs on the discovery board during development in order to provide a visual indication of which User-Defined Task is currently executing. Note that a User-Defined Task must call `complete_dd_task` once it has finished its execution.

## 2.3 Deadline-Driven Task Generator

The DD-Task Generator is an auxiliary F-Task (or a series of F-Tasks) responsible for periodically generating DD-Tasks. It is normally suspended and is resumed whenever a software timer callback is triggered. The timers should be configured to expire based on a particular DD-Task's time period. As part of its execution, the DD-Task Generator prepares all of the necessary information for creating specific instances of DD-Tasks and then calls `release_dd_task`.

To implement this functionality, a single generator could be used to create all DD-Tasks (Figure 12a); alternatively, each DD-Task could have its own dedicated generator (Figure 12b). Other implementations may also be suitable. Assess the options and justify the design in the report.

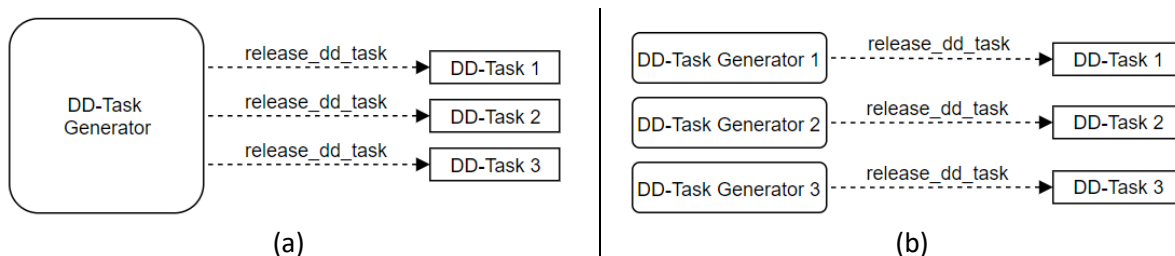


Figure 12: Possible implementations for generating DD-Tasks.

Note that the F-Task handles stored inside each DD-Task may either be created once (when the application is initialized) and subsequently re-used by the DD-Task Generator(s), or the F-Task handles may be continuously created and deleted every time a DD-Task is released and completed. Note that if the latter option is selected, FreeRTOS will need to be configured to use `heap_4.c` instead of `heap_1.c` for improved memory management. Once again, assess the options and justify the design in the report.

## 2.4 Monitor Task

The Monitor Task is an auxiliary F-Task responsible for reporting the following system information:

1. Number of active DD-Tasks
2. Number of completed DD-Tasks
3. Number of overdue DD-Tasks

The Monitor Task collects information from the DDS using the `get_active_dd_task_list`, `get_complete_dd_task_list`, and `get_overdue_dd_task_list` functions. It must then report the number of tasks in each list to the user (e.g. print text to the console). The Monitor Task must be allowed to execute even if there are active or overdue tasks so that it can continue to collect information and report system information. The Monitor Task introduces overhead into the system and therefore must be designed to minimize the amount of time it spends performing its function.

For an additional challenge, consider measuring and reporting processor utilization and system overhead. To implement this functionality, it may be necessary to check the status and the availability of the CPU using FreeRTOS APIs. Alternatively, it may be possible to calculate these values by extending the `dd_task` structure with additional timing fields (e.g. add an array of `interrupt` start and stop times) and using the DDS to record these times. With more information about each DD-Task, the Monitor Task may be able to perform simple calculations to determine processor utilization.

### 3 System Evaluation

The functionality of the DDS must be tested using each of the three test benches in Table 3. Each test bench must be executed for at least one full hyper-period. Note that the test benches are only specific examples and do not necessarily provide complete test coverage of the system.

Table 3: DDS Test Benches.

Task	Test Bench #1		Test Bench #2		Test Bench #3	
	Execution Time (ms)	Period (ms)	Execution Time (ms)	Period (ms)	Execution Time (ms)	Period (ms)
$t_1$	95	500	95	250	100	500
$t_2$	150	500	150	500	200	500
$t_3$	250	750	250	750	200	500

Set up the system code to execute Test Bench #1 and set a breakpoint at the start of the DDS F-Task. Begin the test and use the debugger to record the DD-Task release and completion times for all DD-Tasks for one hyper-period (1500ms) and record them in Table 4. Compare the results with expected values and indicate if any task has missed its deadline. **Include a copy of this table in your report.**

Table 4: DDS events for Test Bench #1.

Event #	Event	Measured Time (ms)	Expected Time (ms)
1	Task 1 released	?	0
2	Task 2 released	?	0
3	Task 3 released	?	0
4	Task 1 complete	?	95
5	Task 2 complete	?	245
6	Task 3 complete	?	495
⋮	⋮	⋮	⋮
17	Task 1 released	?	1500



Execute Test Bench #2 and record the output from the Monitor Task in Table 5 after one hyper-period (1500ms). Compare the results with expected values and indicate if any task has missed its deadline. **Include a copy of this table in your report.**

Table 5: Monitor Task output for Task Bench #2 after one hyper-period (1500ms).

	Measured	Expected
Number of active DD-Tasks	?	?
Number of completed DD-Tasks	?	?
Number of overdue DD-Tasks	?	?

For reference, the scheduling diagram for the first 500ms of Test Bench #2 is shown in Figure 13.

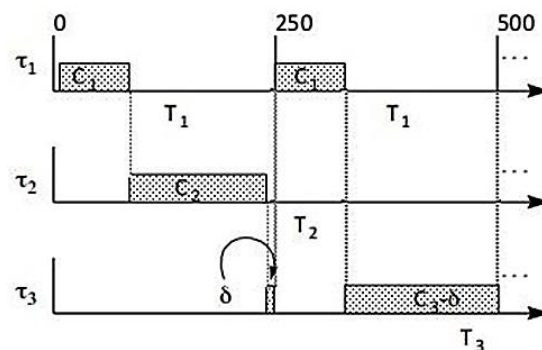


Figure 13: Scheduling diagram for Test Bench #2.

## 4 Requirements

### 4.1 Functional Requirements

- The DDS must correctly execute the three Test Benches using EDF scheduling.
- The Monitor Task must regularly report the number of active, completed, and overdue DD-Tasks.

### 4.2 Technical Requirements

- Auxiliary tasks must only interface with the DD Scheduler's via its four main functions.
- Tasks must be used to control code execution.
- Queues must be used for inter-task communications. Global variables may NOT be used for this purpose.
- Software timers must be used for generating time-based events.
- Task Notifications may not be used.
- The system must be designed to accommodate aperiodic tasks. However, aperiodic tasks do not need to be tested or demonstrated.

## 5 Recommended Steps

The following steps are recommended for completing this project:

1. Create a design document which considers what tasks, queues, functions, timers, etc. will be needed and how these components will interact. **Include a copy of the design document in your report.**
2. Write a simplified DD-Task Generator and `create_dd_task` function to send messages to the DDS.
3. Implement the DD-Task Lists and their support functions.
4. Complete the DD-Task Generator and `create_dd_task`.
5. Implement `delete_dd_task`.
6. Refine the system and subject it to the test benches.
7. Implement the Monitor Task.
8. Refine the system to meet all requirements.

## 6 Notes

- This project is extremely difficult if not impossible to ‘hack’ together. Therefore, be sure to create a design *before* writing any code since failing to do so will likely increase the time required to finish the project.
- The program I/O (i.e., `printf`) should be kept to a minimum as this increases overhead.
- Consider demonstrating the project with a simple toy application that requires real time functionality. The LEDs and the pushbutton on the discovery board may be used for this purpose. Bonus marks may be awarded for a novel real time DDS application.

## 7 Deliverables

1. Demonstrate your project to the TA.
2. Submit a project report. Include the following:
  - System overview diagram which shows the interactions between F-Tasks, queues, and DDS functions.
  - List of all F-Tasks and their priority levels.
  - Implementation details for each of the main DDS functions.
  - Flow chart of how the Deadline-Task Generator works.
  - Flow chart of the DD-Task sorting algorithm.
  - Flow chart of how the DD Scheduler works.
  - Table of DDS events generated for Task Bench #1.
  - Table of Monitor Task outputs for Task Bench #2.
  - Discussion of DDS scheduler performance.

## 8 Alternative Project

Students have the option to complete an alternative project of their choosing instead of completing the DDS project. However, this option must first be discussed with both the instructor and TA before proceeding. If additional hardware will be used to interface with the discovery board, the TA will need to approve the design to ensure there is sufficient over-voltage and over-current protection in place to avoid damage to the board as this will affect all users of the lab.

Note that alternative projects in the past have produced mixed results. Although they can sometimes be rewarding for students, there is typically a large time commitment involved, and unexpected problems can occur. Consequently, it is highly recommended that students first complete the DDS project and ensure that all of its requirements are met. If there is time left over, the complexity of the DDS can be increased by adding more tasks or by having it perform more useful calculations. The design can also be more rigorously tested and the system can be optimized to minimize overhead and improve confidence in its reliability. Since the STM32F4 discovery board is inexpensive and the IDE software is free, this project can also be extended at any time in the future.