MREN 178 - Lab 2

Stacks, Queues and Recursion

Jano Fu

Matthew Pan, PhD

A. Introduction

A.1 Objectives

The goal of this lab is to build upon what you've learned in Lab 1 with regards to coding data structures and algorithms for the Arduino. We will be using the same 1602 LCD Keypad Shield from the previous lab. Additionally, in this and future labs, we're going to throw a new sensor into the mix that will generate data.

In this lab, we will be using a DS3231 I2C Real-Time Clock (RTC) module from Adafruit (see here for more information: https://learn.adafruit.com/adafruit-ds3231-precision-rtc-breakout). The RTC is essentially a timekeeper; it uses a 32 kHz quartz timing crystal - a material that vibrates at a predictable and constant 32768 Hz frequency when electrically stimulated that allows us to keep precise track time. This DS3231 RTC module is very awesome in a few ways:

- The DS3231 includes the crystal inside the integrated circuit (IC) package to make It more robust while also having low power draw (important when it's being used for applications where your sensor is running off a battery).
- There is a temperature sensor within the IC that allows the chip to compensate for differences in oscillation frequency due to temperature (you can also access temperature data using a function in the DS3231 Arduino library if you wanted to). Thus, this sensor is incredibly *accurate* and *precise*. It is rated to a maximum time drift of only two minutes per year.

If you're experienced with the Arduino, you might know that the Arduino has its own oscillator near the USB port (metal pill-like case with 'T16.000' etched on it). So you might be thinking, why do we use an RTC module if the Arduino also keeps track of time? The answer is fairly simple: when the Arduino loses power, the onboard oscillator stops vibrating as well. Thus, the Arduino doesn't keep track of time per se – it keeps track of *time periods* and *time since last reset*. An RTC is different in that it uses a coin cell battery (included with the kit) to always power the oscillator – giving you a stable time signal even if your Arduino dies. This means if you set the current date and time on the module using an Arduino sketch, it'll keep track of the current date and time for years. This is the same method that your laptop or desktop computer uses to maintain a sense of time after it powers off.

For this lab, we will be generating time data using the clock and storing them in stack and queue data structures.

Figure 1. DS3231 Real Time Clock module

A.2 New Topics Covered

- DS3231 Precision RTC Module
- Solderless Breadboarding (the workhorse for electronics prototyping!)
- Stacks
- Queues

A.3 Pre-Requisite Knowledge

This lab assumes that you have completed the pre-lab for Lab 2 and have reviewed the different implementations of stacks and queues in C, as well as solderless breadboarding. Much of the experience you gained working with the Arduino and the 1602 LCD Keypad Shield in lab 1will be used in this lab as well.

A.4 Equipment

Most everything that you need to complete this lab is included.

From your MREN 178 lab kit, you will need:

- Arduino UNO R3 board
- 1602 LCD Keypad Shield
- DS3231 Precision RTC Module
- CR1220 3V Button Cell Battery (insert into the slot of the DS3231 – ensure the positive, flat, non chamfered edge of the battery is inserted away from the board)
- Solderless Breadboard
- Female-Male Jumper Wires
- Male-Male Jumper Wires
- Male USB-A to Male USB-B Cable

From the OnQ MREN 178 course site, you will need:

- Lab 2 Instructions (this document)

- Pre-Lab 2 Instructions and code (for reference)
- A zip file containing source code for the lab.

On your computer, you'll need:

- The Arduino IDE
- Internet Access

B. Lab Setup

B.1 Connecting the LCD Keypad Shield and RTC Module to the Arduino

In Lab 1, you were able to mount the LCD Keypad Shield directly on the Arduino. Because we need access to some of the Arduino's pins for connecting the DS3231 RTC module, we will not be able to mount the shield directly on the board. Instead, we'll be breaking out the solderless breadboard and jumper wires to help make some connections. It would be best if you read the tutorial from SparkFun and/or Adafruit in the prelab on using the breadboard and jumper wires.

All we need to do is connect the necessary pins of the LCD keypad shield and DS3231 RTC module to the Arduino. The RTC clock in our lab only requires connections for SCL, SDA, Vin, and GND. The pins we need to connect are shown in Table 1.

Table 1. Pin assignments for module assembly.

Arduino Pin	Module	Module Pin
8	LCD Keypad Shield	LCD RESET
9	LCD Keypad Shield	LCD Enable
7	LCD Keypad Shield	LCD D7
6	LCD Keypad Shield	LCD D6
5	LCD Keypad Shield	LCD D5
4	LCD Keypad Shield	LCD D4
A0	LCD Keypad Shield	Buttons
SCL	DS3231 RTC module	SCL
SDA	DS3231 RTC module	SDA
5V	DS3231 RTC module & LCD Keypad Shield	Vin (RTC) & 5V (LCD)
GND	DS3231 RTC module & LCD Keypad Shield	GND (RTC) & GND (LCD)

Unfortunately, your LCD Keypad Shield will likely not have labels for each of the pins. Therefore, we've included a wiring diagram below. Ensure the jumpers are attached to the correct pins! Double check connections before plugging in the Arduino – have a TA check your wiring if your LCD Keypad Shield doesn't light up or the RTC module doesn't send data. Note

that we have the 5V and GND connected to both modules – we use the breadboard's power rails to support 5V (red rail) and ground (blue rail) connections.

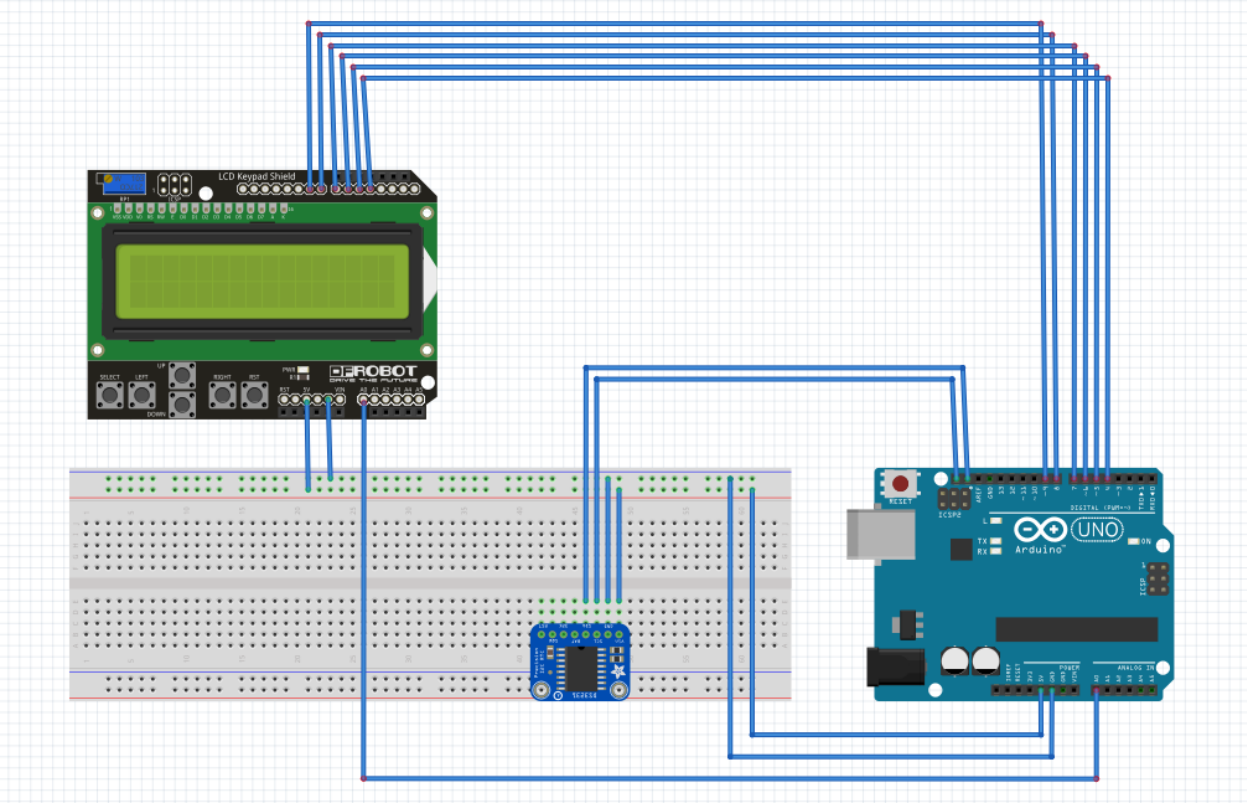
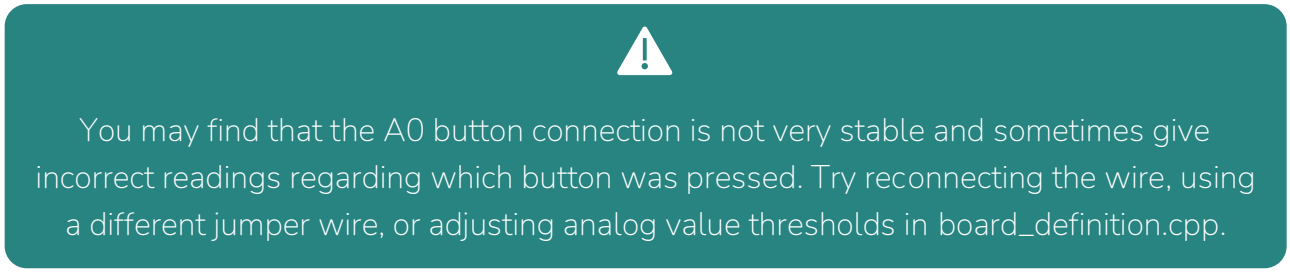


Figure 2. Arduino wiring schematic for Lab 2. This schematic connects the LCD Keypad Shield and DS3231 RTC module.

The solderless breadboard provided has power rails that run only half the length of the board (up to about the 30th row on the board). Ensure that your 5V and ground wires for the shield and RTC module connect to the same rails that the Arduino is connected to.

Be careful when inserting and removing pins of printed circuit boards and integrated circuits (ICs) such as the RTC module or shield into and out of the breadboard. Be gentle, otherwise you may break some pins.



You may find that the A0 button connection is not very stable and sometimes give incorrect readings regarding which button was pressed. Try reconnecting the wire, using a different jumper wire, or adjusting analog value thresholds in board_definition.cpp.

So you might be thinking after making all the wiring connections: "Oh boy. My setup looks as fragile and hacked together as that time dad built a deck by himself because he didn't want to spend money getting a pro to design and build it." The truth of the matter is, this is the reality of prototyping. It's going to look fragile and like a mess of wires; even after spending half an hour trying to connect everything. My setup is shown in Figure 3 – just to give you an idea. Since I worked with a brand-new kit, I kept some of my jumper wires contained in ribbons to make the wiring more manageable.

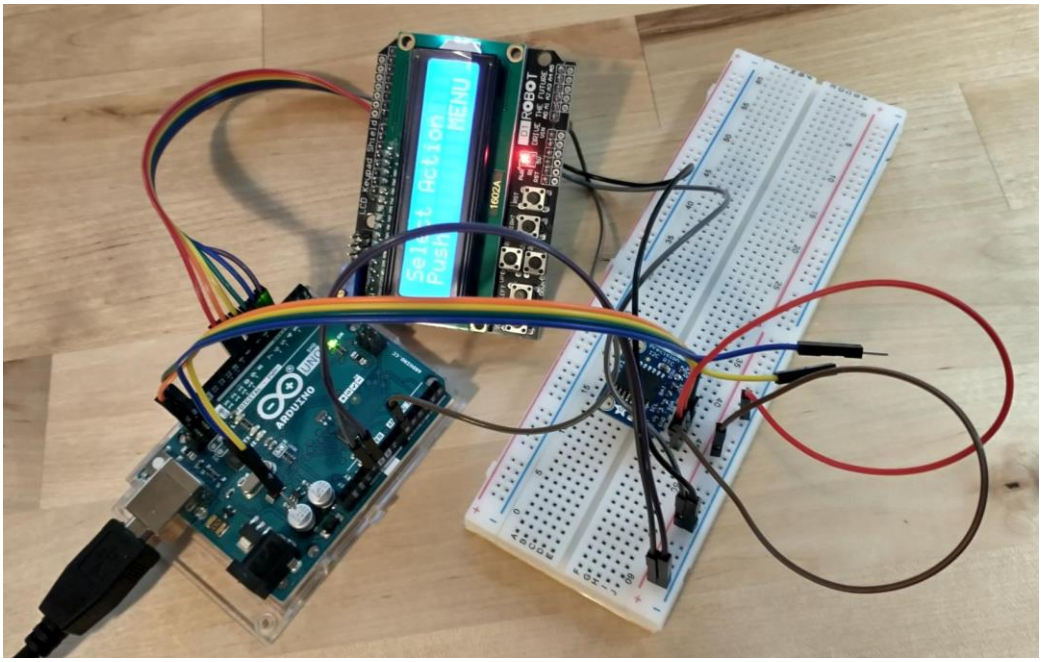


Figure 3. The bane of every hobby electrical project – messy wires and boards that don't sit flush on the table.

There is a possibility of mounting the shield on the breadboard and connecting jumper wires directly to the RTC module; this will help with wiring robustness.

For more information regarding the specifications of the DS3231 module, review the DS3231 datasheet found in the lab materials on OnQ. Although technically dense, it does offer everything you need to know about using the module. Skim over it to get an idea of what

information is included. Datasheets for components are supplied by vendors and will help you assess which components to get for an electronics project and how they work. You'll be using them quite frequently in your career. Being able to extract pertinent data from them is an excellent skill to develop.

B.2 Downloading and Including the DS3231 Library

Installation

The DS3231 module requires its own library for receiving commands. Downloading this library is straightforward. Navigate to library manager on the Arduino IDE using the shortcut Ctrl+Shift+I (eye) or go to 'Tools > Manage Libraries...' as shown in Figure 4.

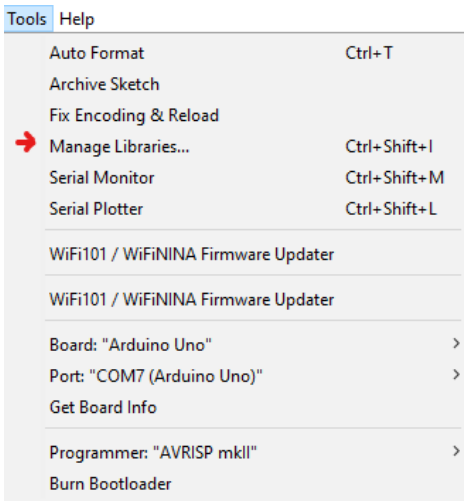


Figure 4. Manage Libraries selection in Arduino IDE menu.

In the library manager, type 'DS3231' in the search bar. Select the library' DS3231' by Andrew Wickert and install. Once installed, the library item should display 'INSTALLED', as shown in Figure 5.

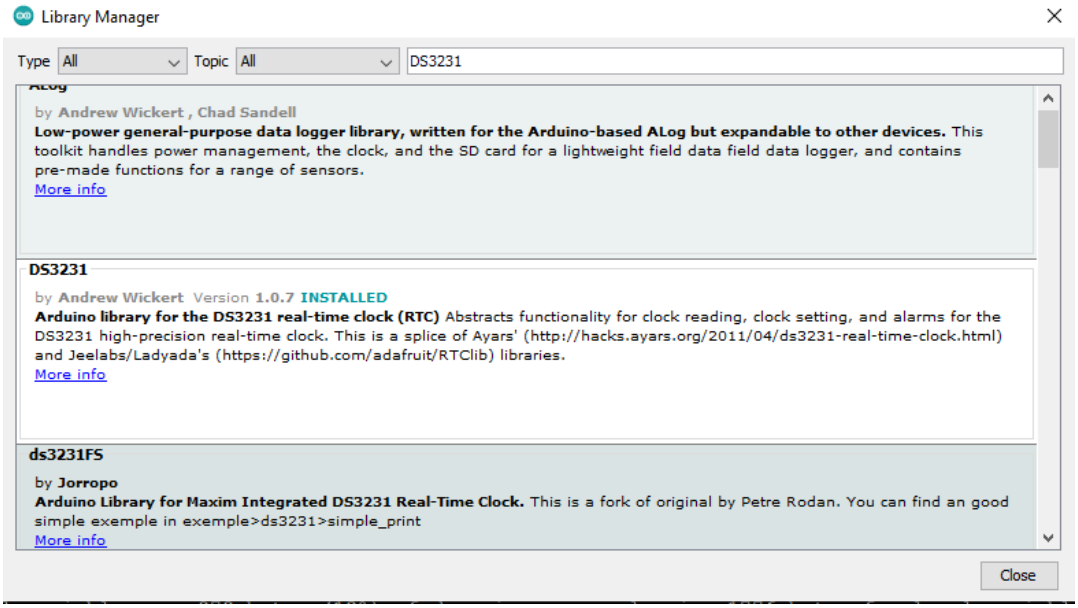


Figure 5. DS3231 in Arduino Library Manager.

Using the Library

To be able to use and command functionality of the DS3231, we need to include the DS3231.h file, which gives us access to functions of the DS3231 library we just downloaded. Examining the lab code, you'll see this has already been included for you in stack.h and queue.h code.

```
#include <DS3231.h>
```

The DS3231 library provides a lot of functionality for setting and getting dates and times from the RTC module. It 'abstracts away' the communication protocol used to transfer data from the module to the Arduino (Inter-Integrated Circuit protocol or simply I^2C – one of 2) so that you don't need to know the nitty-gritty of how that protocol works. You just simply need to call C++ class methods to get that data.

And that's the complicating part: unfortunately, the DS3231 library is written in C++ (as much of the Arduino ecosystem is based in C++) so it uses a C++ 'class', which you are likely not familiar with. I'll give you a brief primer on classes in case you want to modify the code to collect a different type a data: A class is like a much more advanced C struct. Remember how a struct has its own fields/variables that you can access via the dot ('.') or arrow ('->') operators ? Well, a class is like that but it also has its own functions that can be called using the dot and arrow operators. These 'functions' that belong to the class are referred to as 'methods'. So, for example, let's say we define an object (that's what we call instances of classes) of the class DS3231 called clock using this line of code:

```
DS3231 rtc_clock;
```

The definition of the DS3231 class is defined in DS3231.h in our case and implemented in DS3231.cpp – you can find both in the Arduino library folder now that you've downloaded the DS3231 library. So, the variable **`rtc_clock`** is now declared an 'instance' of the DS3231 class.

Now with rtc_clock, you can access its methods (functions). For example. we can get the year stored on the DS3231 module using:

```
rtc_clock.getYear();
```

In fact, the library that you've included allows you to get a whole bunch more data using the DS3231 class methods, as shown in

Table 2.

Table 2. List of DS3231 class methods that allow you to retrieve data from the DS3231 RTC module.

DS3231 Methods	Description	Return Type
`rtc_clock.getYear()`	Returns the year (two-digit value)	int
`rtc_clock.getMonth(false)`	Returns the month as an. The only parameter this method takes is a **`bool`** (Boolean) which is whether a century has passed since the year 2000 (fun fact: it hasn't).	int
`rtc_clock.getDate()`	Returns day of month	int
`rtc_clock.getDoW()`	Returns day of week	int
`rtc_clock.getHour(bool *h12Flag, bool *pmFlag)`	Returns the hour. Depending on the clock's settings, The value pointed to by h12Flag will be changed by the method to true if the hour is represented in a 12 hour format. Otherwise, hour is represented in 24 hour format. If the value pointed to by pmFlag is true (set by the method), it's the pm. Otherwise, it's am.	int
`rtc_clock.getMinute()`	Returns the current minute of the hour	int
`rtc_clock.getSecond()`	Returns the second of the minute	int
`rtc_clock.getTemperature()`	Returns the current temperature detected by the module in Celsius	int
`clock.oscillatorCheck()`	Tells you whether the time is (likely to be) valid (i.e., whether the button cell battery was removed).	bool

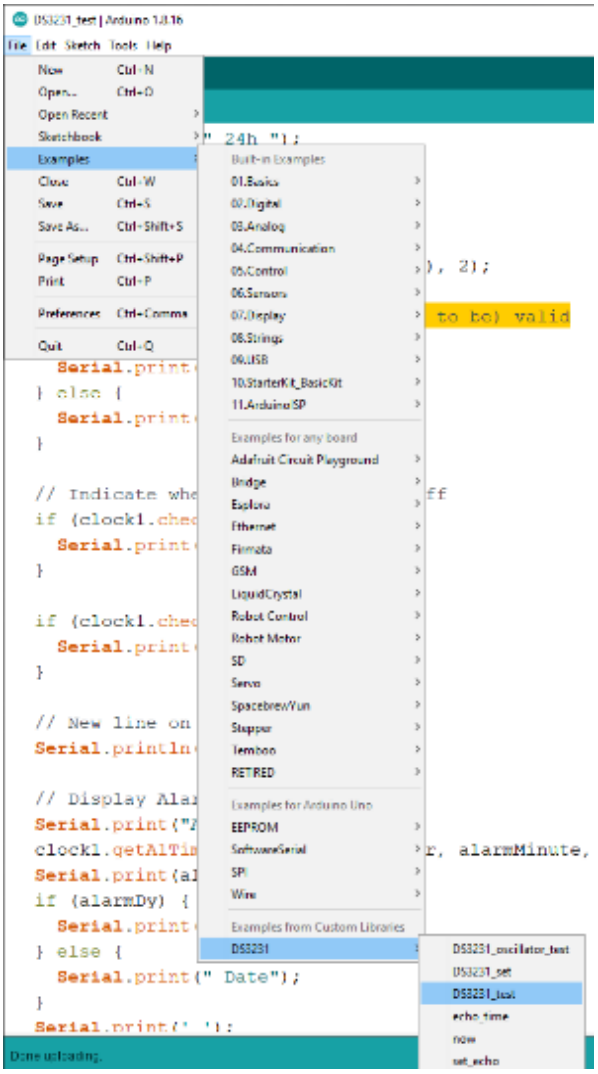


Figure 6. Accessing DS3231 examples from the examples menu in Arduino IDE.

The DS3231 library includes some examples that can be accessed from the 'examples' menu (Figure 6). Try examining and running DS3231_test. When viewing results in serial monitor, remember to set the Baud rate (the serial communication rate) to the rate defined in the sketch, otherwise, you'll get gibberish.

If you find that you are getting an angry " 'DS3231 clock' redeclared as a different kind of symbol" error when verifying the sketch, you'll need to change the name of the clock variable defined in the sketch to something different, like rtc_clock or clock1. There is a naming conflict with an Arduino function of the same name in newer versions of Arduino.h.

Exercises (NOT FOR MARKS)

When you first insert the battery and use the RTC clock module, the output of the test sketch will likely give you the incorrect date and time (just after midnight on January 1st 2000. You'll need to correct this using the example DS3231_set sketch located in the examples menu.

- Read the code to figure out how to set the time using the serial monitor. You'll only need to do this once as long as the button cell battery is inserted and doesn't run down completely.
- Figure out what the Unix epoch is - specifically what date and time it is and in which time zone. Why is the Unix epoch important?
- Based on the DS3231 examples and documentation from the DS3231 library (https://github.com/NorthernWidget/DS3231), write a function that converts the

date and time from the DS3231 RTC module and provides the number of seconds since the UNIX epoch. Check your results using https://www.epoch101.com/.

C. Lab Exercises

C.1 Stacks

Description

In this part of the lab, we will be implementing a Stack data structure and associated algorithms using arrays on the Arduino. Instead of working with one stack, the code here is meant to handle multiple array-based stacks. These stacks, in turn, be stored in an array of type Stack. In other words, you'll have an array, where each element in that array will contain a Stack data structure, which in turn has an array of stack items.

As data, the Arduino will accept character input from the serial interface at 9600 baud. Up to ten characters read over serial will be pushed a character-at-a-time in order onto a stack of your choosing. For example, if I selected that data should go into stack 2 and entered in the text 'abcde' in the serial monitor, 'a' will be pushed, then 'b', etc. Characters over the 10 characters will be pushed the next time you select the 'Push' menu option from the LCD keypad interface. Each character entry into the stack will be timestamped.

Sometimes, you may get spurious data sent over serial and pushed into the stack due to naïve implementations of the buffer in the stack code. This is normal. The Arduino serial comm generally isn't the most stable form of communication and may be prone to errors.

The code for this part of the lab is found in the folder 'lab2_1'. There will be a few source code files:

- *lab_2_1.ino* – initializes the LCD keypad shield, serial and I^2C communications, DS3231 clock, and an array structure to hold the stacks. The loop function handles the button/LCD text interface displayed on the LCD keypad shield.
- *board_definition.h* – contains enums and function prototypes to drive the button-input, text-based menu interface on the LCD keypad shield.

- *board_definition.cpp* – implements the function prototypes found in board_definition.h. Some functions (e.g., PushInterface(), PopInterface(), Print()) interacts with the array of stacks using functions you'll be writing.
- *stack.h* – has preprocessor definitions of how many stacks will be in the array of stacks (NO_OF_STACKS), and the maximum stack array length (MAX_STACK_LEN). It defines a struct for an item that goes in the stack (containing an int timestamp. and a char), as well as another struct that defines a Stack (includes a top and stack_of_items array). Finally, there are function prototypes that define interactions with the stack.
- *stack.cpp* - will contain implementations of functions defined in *Stack.h*. Some will be filled out, others will not. Functions that will need to be completed include: Push(), Pop(), IsStackFull(), IsStackEmpty(), PopAll(), and GetTime(), FUNCTION PARAMETERS AND RETURN TYPES MUST NOT BE MODIFIED.

You should look over all files to ensure that you understand how the program functions.

Deliverables

stack.cpp

- Completed Push(), Pop(), IsStackFull(), IsStackEmpty() and PopAll() functions that allow for proper execution of the program as described above.
- GetTime() should represent a representation of a timestamp generated from the DS3231 module that can be returned in int form. Examples could include: epoch, hhmmss, seconds elapsed since today etc.
- Headline comment section need to be filled in with your name and student ID, your partner's name, as well as any additional contributors to your code (including websites and resources you used to assist in the lab exercise).

C.2 Queues

Description

In this part of the lab, we will be implementing a linked-list-based queue data structure and associated algorithms on the Arduino. Like the stacks exercise, you'll be working with multiple queues that will be stored in an array (of Queue structures). Unlike the stacks exercise above, you won't be collecting data over serial. Instead, the item struct type will only include one data value: an integer timestamp that records the time that the item was enqueued in the queue selected.

The code for this part of the lab is found in the folder 'lab2_2'. There will be a few source code files in a similar setup compared to the stacks program code:

- *lab_2_2.ino* – initializes the LCD keypad shield, serial and I^2C communications, DS3231 clock, and an array structure to hold the queues. The loop function handles the button/LCD text interface displayed on the LCD keypad shield.
- *board_definition.h* – contains enums and function prototypes to drive the button-input, text-based menu interface on the LCD keypad shield.
- *board_definition.cpp* – implements the function prototypes found in board_definition.h. Some functions (e.g., EnqueueInterface(), DequeueInterface(), Print()) interacts with the array of queues using functions you'll be writing.
- *queue.h* – defines how many queues will be in the array of queues (NO_OF_QUEUES). It defines a struct for an item that goes in the stack (containing only an int timestamp. and a pointer to next). as well as another struct that defines a Stack (includes a top and stack_of_items array). Finally, there are function prototypes that define interactions with the stack.
- *queue.cpp* - will contain implementations of functions defined in *queue.h*. Some will be filled out, others will not. Functions that will need to be completed include: CreateItem(), Enqueue(), Dequeue(), IsQEmpty(), DequeueAll(), and GetTime(), FUNCTION PARAMETERS AND RETURN TYPES MUST NOT BE MODIFIED.

Again, you should look over all files to ensure that you understand how the program functions.

Deliverables

queue.cpp

- Completed CreateItem(), Enqueue(), Dequeue() and DequeueAll() functions that allow for proper execution of the program as described above.
- GetTime() should represent a representation of a timestamp generated from the DS3231 module that can be returned in int form. Examples could include: epoch, hhmmss, seconds elapsed since today etc. You can simply copy the code you have written for GetTime() in stack.cpp
- Headline comment section need to be filled in with your name and student ID, your partner's name, as well as any additional contributors to your code (including websites and resources you used to assist in the lab exercise).

The file queue.cpp should be uploaded to OnQ before the next lab session commences on Monday.

D. Grading Rubric

Component	Item	Mark	Max
lab2_1	stack.cpp		25
	`Push`		7
	`Pop`		7
	`PopAll`		2
	`IsStackEmpty`		2
	`IsStackFull`		2
	`GetTime`		5
lab2_2	queue.cpp		20
	`IsQEmpty`		2
	`Enqueue`		7
	`Dequeue`		7
	`DequeueAll`		4
	`GetTime`		0
	Total		45