ENGT5259 EMBEDDED SYSTEMS
Laboratory guidance notes
Part 3: LCD Display

Author: M A Oliver, J A Gow *v1.1*

Table of Contents

Version record

Version	Notes	Date	By
1.0	Initial version	06/03/22	MAO/JAG
1.1	Minor revision	09/03/22	MAO/JAG

PLEASE READ THIS CAREFULLY

This is the third of a series of guidance documents to get you started with your embedded systems project. They are aligned with the key aspects of the coursework specification, and thus this document should be read in conjunction with the coursework specification

document. These documents will start off with some fairly close guidance - however as you develop your skills the notes will provide less and less detailed guidance and become more of an overview.

Note that the implementation is yours, and **you are encouraged to experiment**, or to try different approaches. As long as (a) you clearly understand the code you have written and (b) your result complies with the specification, you will pass. There is no 'right answers' in this module. The benchmarks are: (a) have you hit the learning outcomes and (b) does your final project code comply with the specification.

Note that this module is not a C/C++ programming course. Most embedded system development is carried out in C/C++ and you should have had exposure to these languages as part of your undergraduate studies. If you are rusty, or are not familiar with this language there are plenty of tutorials available online (some links are available on the Blackboard shell) that can guide you. It is possible to learn C/C++ alongside this project. They are not especially 'difficult' languages to learn when you are using them to solve a problem and there are some 'templates' available to help.

You are advised to read this worksheet in its entity before commencing, as it contains useful information throughout.

1 Aims and objectives

This document covers an introduction to the implementation of the requirement of section 1 of the coursework specification. This should take you about 1 to 2 weeks to work through. It also details how the provided template files can be integrated into your existing project.

1.1 LCD Display Driver

This work focusses on the development of firmware for displaying RPM information on the 16x2 LCD display.

Data is sent from the host microcontroller to the 16x2 LCD display via an integrated I2C interface board. Rather than working at the I2C-level, you will be permitted to use an open-source driver-library to facilitate your development.

1.2 Operation

The display will feature two modes of operation: Normal Mode and Entry mode. Below are ***examples*** of what the display ***could*** look like. You might like to show some creativity and make your display look different so long as the functionality is per the specification.

1.2.1 Normal Mode

In this mode the Demand RPM and Actual RPM are displayed. An example of this is shown below. Note that leading zeroes are displayed.

1.2.2 Entry Mode

The Demand RPM can be changed in Entry Mode. Pressing a numbered key on the keypad will invoke Entry Mode.

In the example below, the '1' key has been pressed, thus invoking entry mode and the '1' is displayed as the leading digit. The cursor now appears on the second digit.

In the example below, the '2' key has been pressed. This is now displayed and the cursor now appears on the least significant digit.

In the example below, the '3' key has been pressed. This is now displayed.

Once the Demand RPM has been typed pressing the '#' key will return to Normal Mode providing the value is in range. If it is not, an error message will be displayed for 2 seconds, before allowing a user to re-enter their Demand RPM.

Now, pressing the '*' key effects a back-space. In the example below, the '*' key has been pressed twice and the cursor now resides on the middle digit.

1.2.3 Display Library

This contains two files: display.h and display.cpp

This is the driver library for the display. This is largely incomplete. It is your task to complete this library.

2 Integrating the template into your current project.

The following steps are useful for integrating the two module files from the template into your current project.

2.1 Adding the files

Before embarking on this part of the process, please ensure that you have closed down your integrated development environment (most likely the Arduino IDE). ***You would be advised to back up your existing work at this stage before proceeding.***

The following files should now be copied into your containing directory (folder):-

display.h

display.cpp

For those using the Arduino IDE, when you restart the application, these files should now appear in your workspace.

Now, the file control.cpp contains some test code to generate test messages. You might like to incorporate this into your workspace as well.

2.2 Creating new message IDs

Examine the code inside the file common.h. It currently contains several message IDs.

For the display we require three new message IDs to reflect when the Actual RPM needs updating, when the Demand RPM needs updating and the Demand RPM has been successfully updated via the keypad.

Therefore, in common.h, you will need to create three new message IDs (with unique ID numbers) for the following messages:-

- New Actual RPM
- New Demand RPM
- New RPM from Keypad

You are advised to retain the naming convention that has been used with the existing message IDs.

2.3 Initialization of the display module

Identify the function `UserInit()` in your .ino file. You will need to add a single function call

```
DISPinitialize();
```

to initialize the display functionality

Please note that the order in which functions are called within `UserInit()` is important.

Also, remember to `#include` the Display module into this file otherwise you will run into compilation errors.

2.4 Control module

In this update, you have code in control.cpp that can be used to partially test some of the functionality of the display module. This will be discussed later.

2.5 First compilation

Assuming your code from Exercises 1 and 2 completely compiles, your updated project containing the template should also compile successfully.

3 Implementing the display functionality

Your next task is to implement the keypad functionality, according to section 1 of the specification. There is a framework already present in the demo code to help get you started.

From the template, the header file for the display module (display.h) should not require any modification.

The source file for the display module (display.cpp) is essentially a skeleton template that needs development.

3.1 display.cpp Preliminaries.

The line

```
#include <LiquidCrystal_I2C.h>
```

includes the library for driving the 16x2 LCD Display into the module.

The line

```
LiquidCrystal_I2C lcd(DISP_I2C_ADDR,16,2);
```

creates a display object called **`lcd`**. This is properly configures the `lcd` object as a 16x2 device with the correct I2C address.

The variable declarations

```
static unsigned int ActualRPM = 0;
static unsigned int DemandRPM = 0;
```

respectively contain the actual RPM and the demanded RPM that are to be displayed.

The variable declaration

```
static unsigned int EnteredRPM = 0;
```

contains an RPM value that has been entered from the keypad that is to be validated.

The variable declaration:-

```
static char numarr[5];
```

holds the character sequence entered by the user. This is used in both the task handler and the callback functions.

An enumerated type

```
typedef enum _DISPSTATE {
     DISPSTATE_REFSH,
     DISPSTATE_IDLE,
     DISPSTATE_UPDATING,
     DISPSTATE_VALIDATE,
     DISPSTATE_ERROR
} DISPSTATE;
```

is used to define the 5 states of the task handler. These states will be covered later.

The variable declaration:-

```
DISPSTATE state = DISPSTATE_REFSH;
```

declares a variable called `state` of type `DISPSTATE` and initializes it to `DISPSTATE_REFSH`

The module contains 5 functions that you will need to work on:

`DISPTask()` is the display task handler

`DISPUpdateRPM()` is the message handler for actual RPM updates

`DISPUpdateDemandRPM()` is the message handler for demand RPM updates

`DISPKeyPressed()` is the callback in response to keypresses

`DISPInitialize()` is used to initialize the display module.

3.2 Display Module Initialization.

The first block of code has been written for you. This initializes the 16x2 display, illuminates the backlight, clears the display and sets the cursor to the top-left position.

The second block of code requires you to set-up three message queue subscriptions:-

1. The callback function `DispUpdateRPM()` should be mapped against the message ID indicating New Actual RPM.

2. The callback function `DISPKeyPressed()` should be mapped against the message ID indicating a keypress (from the “Laboratory Guidance Notes Part 2”).
3. The callback function `DISPUpdateDemandRPM()` should be mapped against the message ID indicating New Demand RPM.

You will need to write the third and final block of code within `DISPInitialize()` is used to register the task handler (`DISPTask`) with the kernel. This is currently empty.

3.3 Updating Actual RPM

You are provided with a function called `DISPUpdateRPM()`. It starts with a line of code implemented that casts the variable `newrpm` from the `context` pointer.

This function is provided for you. When this function is invoked, it will display the Actual RPM on the display as a 3-digit quantity.

Now, writing to the 16x2 display is a relatively slow process. Therefore it should only be updated if absolutely necessary. Based on this, the display is updated if the following two conditions are both met:-

1. The value of `state` is either `DISPSTATE_REFSH` or `DISPSTATE_IDLE`, AND
2. The value of `newrpm` differs from that of `ActualRPM`

From the code, it can be seen that if both cases are met, then:-

- The value of `ActualRPM` is set to `newrpm`
- The `lcd.setCursor()` function to place the cursor at the position to display your three-digit value. You may wish to change this.
- The `sprintf()` function from `<string.h>` is used to create a formatted text string from the RPM value.
- The `lcd.print()` function is used to write your formatted string to the display.

Now, at this point, you currently do not have an Actual RPM value to work with. If you examine the control.cpp code provided, there is code which will set up a 1 second timer (`TestRPMTimer`). On each timeout a variable called `actualrpm` is incremented. This value is sent to the New Actual RPM message queue and the timer is reset.

When you run the code at this point, you should see the message “Starting..” at the top-left of the display and you should see a test value of Actual RPM increasing every second.

3.4 Updating Demand RPM

You are provided with a skeleton function called `DISPUpdateDemandRPM()`. You will need to complete this function. This skeleton function contains single line of code that casts the variable `newrpm` from the `context` pointer.

Once implemented, this function will display the Demanded RPM on the display as a 3-digit quantity with leading zeros when necessary.

Now, writing to the 16x2 display is a relatively slow process. Therefore it should only be updated if absolutely necessary.

You are advised to update the display if the following two conditions are both met:-

1. The value of `state` is either `DISPSTATE_REFSH` or `DISPSTATE_IDLE`, AND
2. The value of `newrpm` differs from that of `DemandRPM`

If both cases are met, then:-

- Set the value of `DemandRPM` to `newrpm`
- Use the `lcd.setCursor()` function to place the cursor at the position where you wish to display your three-digit value.
- Use the `sprintf()` function from `<string.h>` to create a formatted text string for your RPM value.
- Use the `lcd.print()` function to write your formatted string to the display.

At this point, you do not have a Demand RPM value to work with. However, you could test this by modifying control.cpp basing it on the test code that is already provided.

- Create a new test RPM timer (with a different update time)
- Duplicate the existing block of RPM test code, replacing the timer name with your new timer name and updating the timing accordingly.
- Replace the variable `actualrpm` with the variable `demandrpm` throughout this short block of code.
- Send the value of `demandrpm` to the New Demand RPM message queue.

When you run the code, you should hopefully see simulated Actual and Demand RPM values incrementing independently at different time instants.

You will need to comment-out this block of test code once this is working.

3.5 Key Pressed Callback Function

You are provided with a skeleton function called `DISPKeyPressed()`.

When configured properly, this function should be invoked on keypress messages.

Firstly, it contains a line of code that populates the variable `keypress` from the `context` pointer.

The course of action taken depends on the value of `state`.

3.5.1 `DISPSTATE_IDLE` or `DISPSTATE_REFSH`

Suppose the value of `state` is either `DISPSTATE_IDLE` or `DISPSTATE_REFSH`. The code to handle this has been written for you to study (this should help you implement the `DISPSTATE_UPDATING` code). Now, for `DISPSTATE_IDLE` or `DISPSTATE_REFSH`, a check is made on the value of `keypress`. If this value does not lie in the range 0 to 9, then this keypress will be ignored. Otherwise we have a valid numeric keypress that will need processing.

The line:-

```
curpos=9;
```

sets the x-position of the cursor to 9. Feel free to change this if required.

The line

```
sprintf(numarr,"%3.3d",DemandRPM);
```

takes the value of the demand RPM and stores it as a formatted textual string (with leading zeros) in the character array `numarr`. The string `numarr` therefore holds the value of the demand RPM as a human-readable string.

Now the ASCII table provides a list of human readable characters and their corresponding numeric codes. These codes are known as ASCII codes and are summarized in the link below.

https://www.asciitable.com/

Next, the character associated with the keypress is placed in the first element of `numarr`. At this point it must be remembered that the ASCII code for the character '0' is 0x30, '1' is 0x31, etc, and the ASCII code for '9' is 0x39.Therefore to convert the key value to its corresponding ASCII character, 0x30 needs to be added:-

```
numarr[0]=0x30+keyval;
```

The code

```
lcd.clear();
lcd.setCursor(0,0);
lcd.print(F("New RPM:"));
```

clears the display, moves the cursor to the top left position, and writes the message “New RPM:”. The `F()` function ensures that the fixed string to be written is stored in program memory rather than RAM to economize on RAM.

The code

```
lcd.setCursor(curpos,0);
lcd.print(numarr);
```

moves the cursor to the column `curpos` (initially 9), row 0 (top row). The string `numarr` is printed onto the display commencing from this position.

Now with

```
lcd.setCursor(++curpos,0);
lcd.blink();
```

the value of `curpos` is incremented and the cursor is therefore positioned in the next column. The character at this new position is made to blink. Any subsequent key presses will be effective on this highlighted digit.

Finally

```
state=DISPSTATE_UPDATING;
```

is used to set the value of `state` to `DISPSTATE_UPDATING`.

3.5.2 DISPSTATE_UPDATING

This is blank and left for you to complete.

There are three situations that will need consideration.

A numeric keypress.
Here the value of `keypress` is between 0 and 9. If the number of characters is less than 3 then you will need to update the corresponding element in `numarr[]`, remembering to store the ASCII representation of the key. Also, remember to update the display. Hint: suppose `numarr[]` is displayed from column 9 onwards, and `curpos` is the current cursor position, then the index for `numarr[]` is simply `curpos-9`.

A "backspace" keypress.
Here the value of `keypress` is 0x0A corresponding to a press of the "*" key. Assuming `numarr[]` is displayed from column 9 onwards. If the cursor position `curpos` is greater than 9, decrement `curpos`, then use `lcd.setCursor()` appropriately to reposition the cursor.

An "enter" keypress.
Here the value of keypress is 0x0B corresponding to a press of the "#" key. The cursor and blinking can be switched off respectively using `lcd.noCursor()` and `lcd.noBlink()`. The value of `curpos` is restored to its original value (presumably 9).

At this point the state can be changed to `DISPSTATE_VALIDATE` as the value has been committed and is awaiting validation.

Now, the entered value of RPM is currently stored in `numarr[]`. One method for extracting this final entered value is to use:

```
sscanf(numarr,"%d",&EnteredRPM);
```

3.6 Display Task Handler

3.6.1 Overview

The function `DISPTask()` is the task handler for the Display module. This is run from the kernel. A skeleton function for `DISPTask()` is provided. The state machine for the Display module contains five states:

- `DISPSTATE_REFSH`
- `DISPSTATE_IDLE`
- `DISPSTATE_UPDATING`
- `DISPSTATE_VALIDATE`
- `DISPSTATE_ERROR`

3.6.2 Implementation

You will need to implement code for each of the five states. The requirement for each state is described below:

DISPSTATE_REFSH

You will need to write the code to refresh the output on the screen. In other words to display the Demand RPM and the Actual RPM on the 16x2 display.

DISPSTATE_IDLE

The function `DISPKeyPressed()` takes care of this state. This results in an empty case; no code is required.

DISPSTATE_UPDATING

The function `DISPKeyPressed()` takes care of this state. This results in an empty case; no code is required.

DISPSTATE_VALIDATE

We land in this state once a demanded RPM value has been entered; the entered value is stored in the variable `EnteredRPM`.

This contains an `if` statement. You will need to rewrite the condition of this `if` statement check for out-of-range conditions for `EnteredRPM`. (If the `EnteredRPM` lies between `RPM_MIN` and `RPM_MAX` it is valid. If it is zero, it is also valid. Otherwise it is out of range.)

For out-of-range values, a 2-second timer is instantiated (code provided). You will need to write code to display an error message. Following this, the `state` is changed to `DISPSTATE_ERROR`.

However, if `EnteredRPM` is in range, you will need to add code to publish the `EnteredRPM` value to your "New RPM from Keypad" message ID.

`DISPSTATE_ERROR`

This code is partially implemented. If the timer has expired, the timer will be deleted and the value of `state` set back to `DISPSTATE_UPDATING`. You will need to write some code to display the EnteredRPM on the 16x2 LCD display and place a blinking cursor over its first character.

3.7 The task

You will be provided with skeleton code modules that you will need to add to your project. These contain some sample code that should help you with certain aspects, such as the control of the 16x2 LCD module.

However, there are regions of the code that you will need to implement. Study the comments. If you see a comment starting “TODO: ”, this will indicate the coding requirements that need to be implemented below that comment.

Once everything is implemented, testing and working, you will have already completed and implemented the majority of Requirement 1 of the specification!