ENGT5259 EMBEDDED SYSTEMS
Laboratory guidance notes
Part 2: Keypad

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

Table of Contents

Version record

Version	Notes	Date	By
1.0	Initial version	16/02/22	MAO/JAG
1.1	Minor word changes	20/02/22	MAO/JAG

PLEASE READ THIS CAREFULLY

This is the second 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.

1 Aims and objectives

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

1.1 Keypad Overview

This work focusses on the development of driver firmware for the keypad on the DMU-designed ATmega328P Microcontroller Development boards.

The keypad contains 12 switches arranged into 4 rows and 3 columns as shown below in Figure 1.

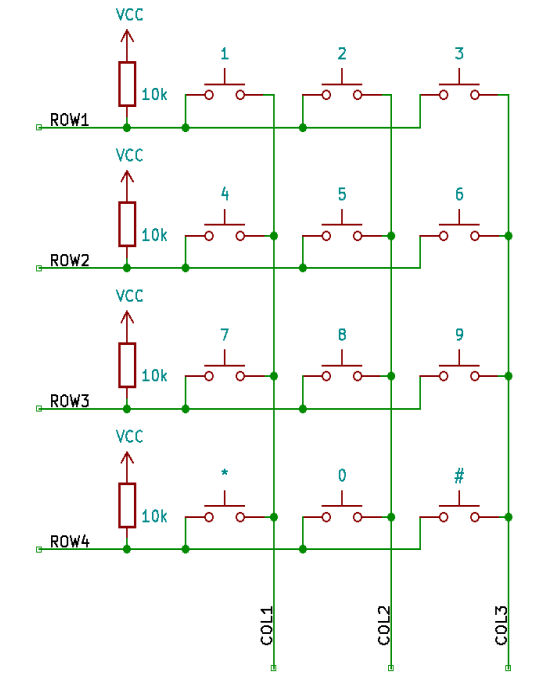

Figure 1: 4 Row by 3 Column Matrix Keypad Arrangement

The four rows are pulled-up to supply by 10k resistors. When interfaced to a microcontroller, the rows are considered to be inputs. With no switches pressed, the state of each row will be high owing to the pull-up resistors.

When interfaced to a microcontroller, the columns should be driven from outputs.

To select Column 1, the Column 1 line is grounded (i.e. by setting the corresponding output to logic 0), whilst Columns 2 and 3 are pulled to a voltage around VCC (i.e. by setting the corresponding outputs to logic 1). This is conceptually illustrated in Figure 2.

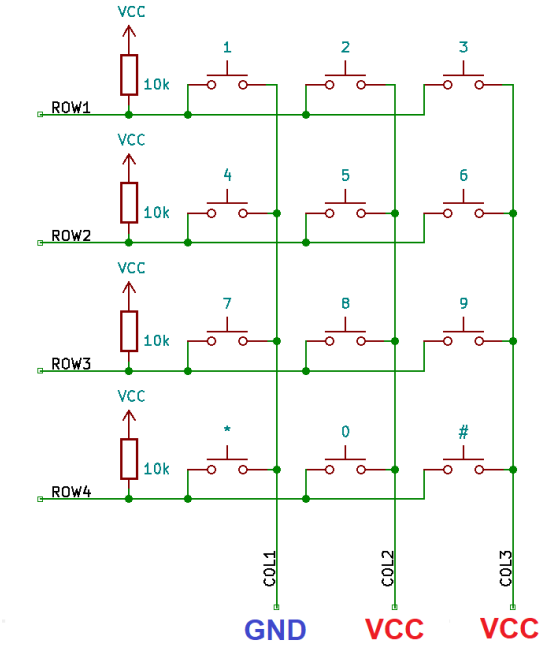

Figure 2: Selecting Column 1

With no switches pressed, the inputs on rows 1 to 4 will all report logic 1 due to the pull-up resistors.

If any key on Columns 2 or 3 is pressed, the corresponding row is essentially connected to VCC. This also results in all inputs on rows 1 to 4 reporting logic 1.

Suppose key 7 is depressed. The Row 3 line gets essentially connected to GND as illustrated in Figure 3. Now when the inputs are interrogated, Rows 1, 3 and 4 are still at logic 1, whilst Row 2 is now at logic 0. Knowing that Row 3 is active when Column 1 is selected indicates key 7 is pressed.

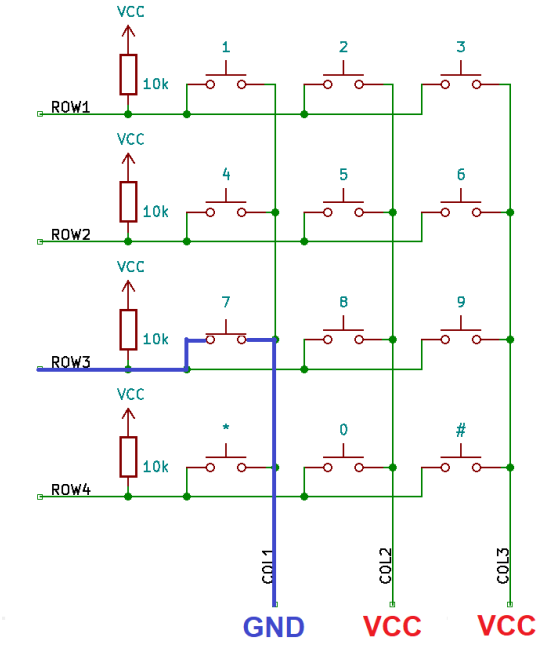

Figure 3. Depressing a key on a selected column

With Column 1 selected, if no key is found to be depressed, then Column 2 is selected. This involves setting the Column 1 and 3 lines to logic 1, and the Column 2 line to logic 0. The states of the rows are then interrogated.

With Column 2 selected, if no key is found to be depressed, then Column 3 is selected. This involves setting the Column 1 and 2 lines to logic 1, and the Column 3 line to logic 0. The states of the rows are then interrogated.

With Column 3 selected, if no key is found to be depressed, then we go back to Column 1, and so on until a key is detected.

The selection is summarized in the table below.

SELECTED COLUMN	COL 1 OUTPUT	COL 2 OUTPUT	COL 3 OUTPUT
Column 1	0	1	1
Column 2	1	0	1
Column 3	1	1	0

1.2 Schematic

Figures 4 and 5 show how the matrix keypad is integrated into the DMU ATmega328P Microcontroller Development Boards.

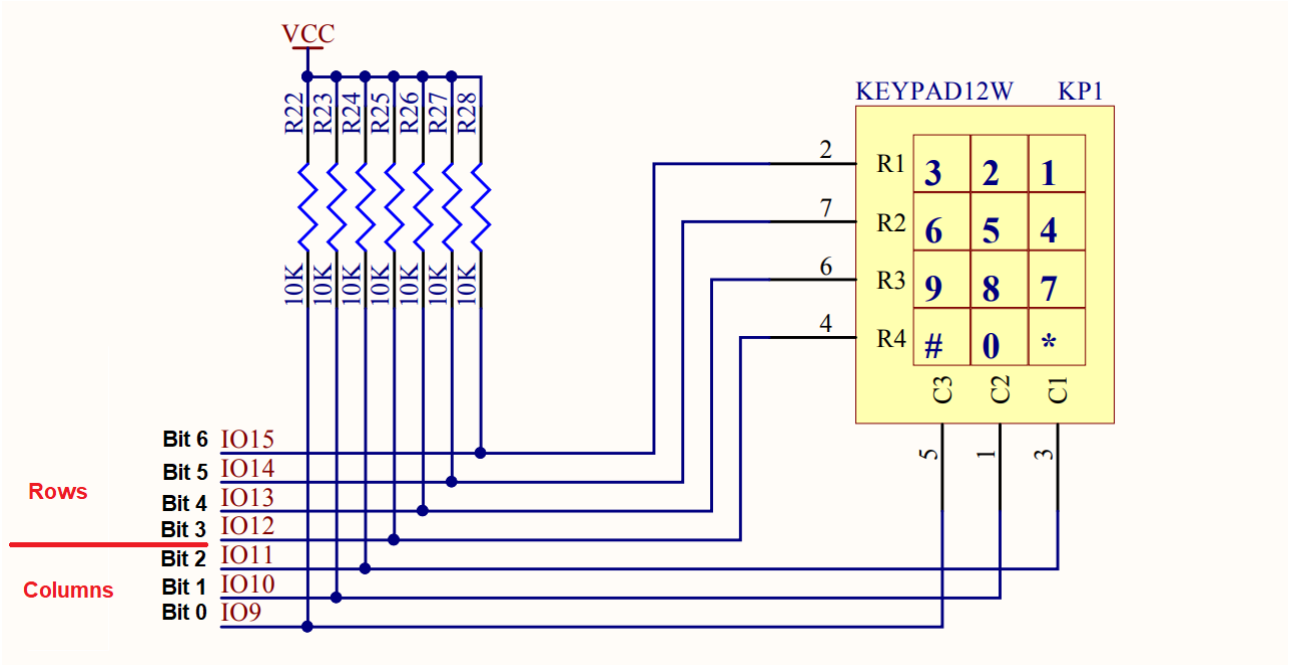

Figure 4. Interfacing the Keypad (i)

Consider the schematic of Figure 4. This shows the 4x3 keypad. The keypad has 3 columns and 4 rows.

In terms of columns:-

Keys '1', '4', '7' and '*' reside on column 1 (Bit 2).

Keys '2', '5', '8' and '0' reside on column 2 (Bit 1).

Keys '3', '6', '9' and '#' reside on column 3 (Bit 0).

In terms of rows:-

Keys '1', '2' and '3' reside on row 1 (Bit 6).

Keys '4', '5' and '6' reside on row 2 (Bit 5).

Keys '7', '8' and '9' reside on row 3 (Bit 4).

Keys '*', '0' and '#' reside on row 4 (Bit 3).

Now these lines could be connected directly to pins on a microcontroller. However, this configuration would require 7 vacant I/O lines. As our ATmega328P does not have 7 spare lines, an alternative approach needed to be taken. Now, the MCP23017 is a 2-byte port extender that interfaces to a microcontroller via an Inter-Integrated Circuit (IIC or I2C) bus; this is also colloquially known as a Two Wire Interface (TWI). The MCP23017 provides an additional configurable 16 input/output lines thus providing ample capacity for accommodating the matrix keypad. The interfacing is shown in Figure 5.

The MCP23017 connects via the Serial Data (SDA) and Serial Clock (SCK) lines, that are respectively found on pins PC4 and PC5 of the microcontroller. Data is interchanged between the microcontroller and the MCP23017 using short command sequences.

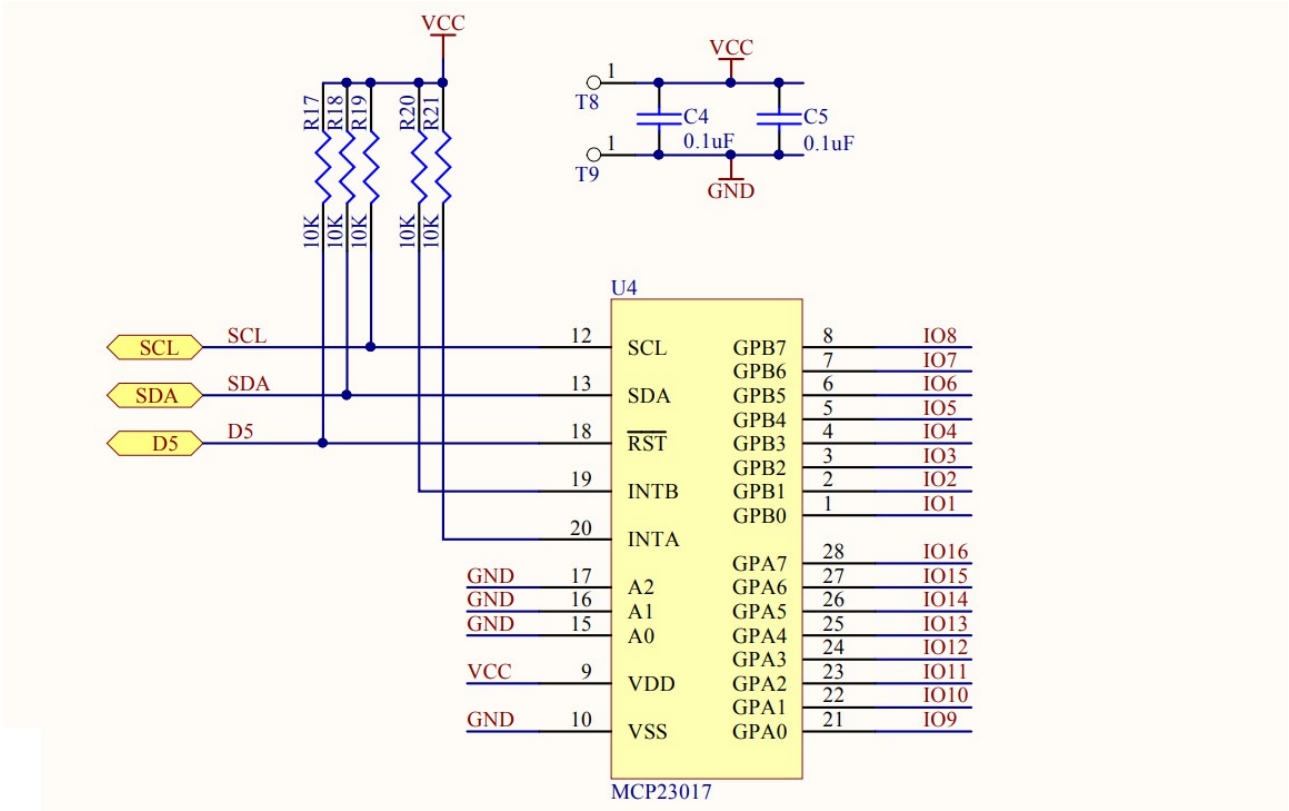

Figure 5: Interfacing the Keypad (ii)

1.3 Template

A template containing four files is provided to add two new libraries into your project. Each module contains a header (.h) and a source (.cpp) file.

1.3.1 IIC Module

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

The iic module contains the functions for accessing the I2C bus. For this part of the project it is used to read and write to the I2C port expander. However, as the project develops, this module will also be used to drive other I2C peripherals.

There are native libraries available for driving accessing the I2C bus. The advantage of using our own module is that it can be ported to other platforms. It also disassociates the project from any of the Arduino libraries that could prove inflexible and slow.

Even though you should not need to modify the code in iic.h and iic.cpp, it would be a good exercise to study the source code in conjunction with the ATmega328P datasheet to develop a deeper understanding.

1.3.2 Keypad Module

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

This is the driver module for the keypad. This is largely incomplete. It is your task to complete this module.

2 Integrating the template into your current project.

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

2.1 Renaming the project

This step is optional. Now, the project file that you worked on last time had the name SevenSeg.ino. You might want to rename this: for example to project.ino or part2.ino. If you do, please ensure that you rename the containing directory (folder) to reflect this. For example, if you renamed the file to Project.ino, then the containing directory (folder) should be renamed to Project.

2.2 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).

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

iic.h

iic.cpp

keypad.h

keypad.cpp

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

2.3 Creating new message IDs

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

For the keypad we require two new message IDs to reflect when a key is pressed and a key is released.

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

- Key Pressed
- Key Released

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

2.4 Initialization of the iic and keypad modules

Identify the function `UserInit()` in your .ino file. You will need to add two function calls to this function:-

- One function call using `IICInitialize()` is used to initialize the iic functionality

- The other function call `KEYInitializeKeypad()` is used to initialize the keypad functionality.

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

Also, remember to `#include` the IIC and Keypad headers into this file otherwise you will run into compilation errors.

2.5 Control module

You should have code in control.cpp that was used to test the seven-segment driver. For this task, it is advised that this portion of code is disabled. Commenting-out this code should suffice.

2.6 First compilation

Assuming your code from Exercise 1 compiles, your updated project containing the template should also compile successfully.

3 Implementing the keypad

Your next task is to implement the keypad functionality, according to section 2 of the specification. There is a framework already present in the demo code to help get you started.

From the template, the I2C module files (iic.h and iic.cpp) should not need modification. However, studying the code alongside the ATmega328P datasheet should be an informative learning exercise. For information:-

- The function `IICInitialize()` as mentioned earlier is used to initialize our I2C module.
- The function `IICWrite()` is used to write a string of data bytes to the specified I2C address. It takes three arguments:-
  - `addr`. This is the I2C address of the recipient device.
  - `dbyte`. This is a pointer to the data to be transmitted.
  - `nToSend`. This is the number of bytes to be transmitted.
- The function `IICRead()` is used to read a string of data bytes from the specified I2C address. It also takes three arguments:-
  - `addr`. This is the I2C address of the recipient device.
  - `dbyte`. This is a pointer to a buffer to store the received data.
  - `nToRecv`. This is the number of bytes to be received.

The header file for the keypad module (keypad.h) should also not require modification.

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

3.1 Keypad Initialization.

The function `KEYInitializeKeypad()` within keypad.cpp initializes the MCP23017. Some of this code is written for you. Some of this requires your own input.

Towards the top of keypad.cpp you will see the following line.

```
#define KEY_ADDR_IIC     0x40
```

The I2C address of the MCP23017 in the development boards is 0x40 (64 decimal).

The first block of code within `KEYInitializeKeypad()`

```
iicreg[0]=0x00;          // IODIRA
iicreg[1]=0xf8;          // bottom 3 pins output
IICWrite(KEY_ADDR_IIC,iicreg,2);
```

writes the bit value 11111000 (0xF8) to the IO Data Direction Register for Port A on the MCP23017. This sets the 3 lowest significant bits (i.e. the column bits) to outputs, and the remaining bits (i.e. the row bits) to inputs.

The second block of code within `KEYInitializeKeypad()`

```
iicreg[0]=0x12;          // GPIOA
iicreg[1]=0x06;          // bottommost bit zero
IICWrite(KEY_ADDR_IIC,iicreg,2);
```

writes the value 00000110 (0x06) to Port A on the MCP23017. Referring to Figure 4, this means that Columns 1 and 2 are at logic-level 1, whilst Column 3 is at logic-level 0. In other words Column 3 is selected.

The third block of code within `KEYInitializeKeypad()`

```
iicreg[0]=0x02;
iicreg[1]=0b01111000;
IICWrite(KEY_ADDR_IIC,iicreg,2);
```

inverts the states of bits 3,4,5 and 6 (i.e. the row inputs). This means that a logic 1 indicates a switch on that row being depressed, and a logic 0 indicates no switch on that row is being pressed.

The fourth and final block of code within `KEYInitializeKeypad()` is used to register the task handler (`KEYTaskHandler`) with the kernel. This is currently empty. Registering task handlers is something you will have encountered in control.cpp during exercise 1. So you should be able to complete this by yourself.

3.2 Keypad Task Handler

3.2.1 Overview

The implementation of the Keypad Task Handler `KEYTaskHandler()` is the main element of this exercise. This is where you will be selecting a column and seeing if a switch is depressed by examining the row inputs. If no switch is detected on a row then you can try the next column and so-on until a switch press is detected.

This process itself requires some thought. What do you think would happen if you continually poll for a keypress until a key is finally detected? Would there be more merit in simply checking just one column every time this task handler is called?

3.3 Finite State Machine Overview

The keypad handler can be implemented as a Finite State Machine (FSM) containing four states. These states are implemented in the template as an enumerated type:-

```
typedef enum _KEYSTATE {
     KEY_IDLE,
     KEY_PRESSDETECTED,
     KEY_PRESSED,
     KEY_RELEASEDETECTED
} KEYSTATE;
```

The first state indicates the idle state where no key is pressed.

The second state indicates when a key is first detected. At this point, the switch is not debounced.

The third state indicates that a key is pressed and has been registered as debounced.

The fourth and final state indicates that a debounced pressed key is released.

The skeleton of this FSM is provided in the template.

3.4 Timer

The `KEYTaskHandler()` has a timer called `KeyTimer` associated with it. This is primarily used for debounce timing. It is initialized to 10ms.

```
static Kernel::OSTimer   KeyTimer(10);
```

3.5 Implementation

This is where your task lies. You will need to determine whether or not a key is pressed. All keypresses should be properly debounced.

One of the first things when determining keypresses is to read the state of the GPIO Port A. This is done using the following code which is provided in the template:-

```
iicreg[0]=0x12;                    // GPIOA register address
IICWrite(KEY_ADDR_IIC,iicreg,1);   // write the address.
IICRead(KEY_ADDR_IIC,&matrix,1);   // read the value
```

The first two lines write the address 0x12 (for the GPIO Port A register) to the MCP23017 via the I2C bus. The third line reads the contents of this address and stores the result in the variable `matrix`.

By examining the value of matrix, it is possible to determine whether or not a key is being pressed, and the actual value of that key.

The bits within the byte `matrix` have the following definitions:-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Not used	Row 1	Row 2	Row 3	Row 4	Column 1	Column 2	Column 3

A detected row (where a key is pressed) yields the value 1. An undetected row (corresponding key is released in selected column) yields the value 0.

Conversely for columns, 0 indicates a selected column, whereas 1 represents an unselected column.

Suppose the value of `matrix` is 0x06 (00000110)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	1	1	0

Looking at bits 0-2, Column 3 is selected. Looking at bits 3-6, no row is detected. Therefore no key on Column 3 is currently pressed. It is time to look at the next column.....

Suppose the value of `matrix` is 0x13 (00010011)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	1	0	0	1	1

Looking at bits 0-2, Column 1 is selected. Looking at bits 3-6, bit 4 is 1 indicating Row 3 is detected. A Row 3 detection on Column 1 indicates that key '7' is pressed.

In the template you have a skeleton Finite State Machine (FSM) for monitoring the status of the keypad and performing debouncing.

The FSM will initialize to the `KEY_IDLE` state. It will remain in this state until a keypress is detected. If a keypress is detected a 10ms debounce timer will be initialized before moving to the `KEY_PRESSDETECTED` state.

In the `KEY_PRESSDETECTED` state, you will need to check whether the timer has expired. If it has not the system will remain in the `KEY_PRESSDETECTED` state. If the timer has expired then the keypad is sampled. If there is a difference between the original keypress and the current sample, then the system will return to the `KEY_IDLE` state. If the two match, it indicates no change in switch state meaning the switch is debounced; at this stage, the value of the pressed key should be posted to the 7-segment and Switch Pressed Message IDs, before moving onto the `KEY_PRESSED` state.

In the `KEY_PRESSED` state, the keypad will also need to be sampled. If there is no difference between the original keypress and the current sample, then the system will remain in the `KEY_PRESSED` state. If there is a difference, a dot-point message will need to be posted to the 7-segment Message ID, whilst the original key value should be written to the Switch Released Message ID; the system should now return to the `KEY_IDLE` state.

Ask yourself the question: Is there a redundant state in the FSM?

3.6 The task

You need to work out the values you would expect to see in the `matrix` variable for each of the 12 key presses. You may want to store these in code as an array or table for rapid decoding.

You will then need to implement the Finite State Machine for keypad debounce. As a start point it might be worthwhile drawing the Finite State Machine on paper before implementing it.

To test the debounce, it might be worthwhile temporarily increasing the debounce time from 10ms to 500ms. A key will need to be depressed for 500ms before it is registered as pressed. If this behaviour is observed, the switch debounce mechanism works and the 10ms debounce time can be restored.

When you press a key, the value of the key should show on the 7-segment display; a dot-point should be displayed for no key press.

And then you have already completed and implemented the majority of requirement 2 of the specification!