1 Overview

For the mini-project, you are required to interface the TM4C123GH6PM microcontroller with an external 20 x 4 Liquid Crystal Display (LCD) and 4x4 keypad to design a simple calculator. The goal of the project is to use the LCD and the keypad to perform some simple calculations. The Keypad will be used as input, and the LCD will output the result of the input calculations. This will entail constructing the hardware using a breadbo[illegible]ese components. You will then write a set of C functions, using the Keil-v5 IDE, to re[illegible]ypad and send commands and data to the LCD.

This document will give on[illegible]is required to complete the mini-project successfully. At this stage of your studies, [illegible]cient at using datasheets and application notes, etc. to solve an engineering probl[illegible]asheets are on Minerva in the same area as this handout. Until you read them, some[illegible]erial will not make sense.

This is an individual mini-p[illegible]by yourself, not with a lab partner.

2 Hardware

2.1 Task 1 – Setting Up the Microcontroller

The goal of the mini project is to interface the Tiva LaunchPad with the LCD and Keypad on a breadboard. Each student was given two mini-breadboards, so you should have these available. Make sure to make good use of the breadboard space for a good circuit layout.

Consult the TM4C123GH6PM data sheet (section 10.5) to check the GPIOs' voltage range and tolerance for correct LCD and Keypad interfacing.

2.2 Task 2 – Connecting the Keypad

Consult the datasheet for the keypad to get the pin assignments. The keypad is a 4x4 matrix (4 rows and 4 columns). Your specification includes the following:

- Keypad rows: input from PORTE [0:3].
- Keypad columns: output to PORTD [0:3].

Note that the row inputs will need pull-down resistors. You can use external physical resistors or find out how to program the internal pull-downs.

2.3 Task 3 – Connecting the LCD

Consult the datasheet for the LCD to get the pin assignments. Connect the relevant power pins to 5V and GND. The following is part of the configuration process when using an external LCD:

- Use a 10 kΩ potentiometer with the middle wire connected to the contrast pin (connect the other potentiometer pins to GND and 5V).
- Use a small valued resistor in series with the LED backlight cathode to limit the current.

You will be using the LCD in 4-bit mode. This means you will only use 4 pins of the microcontroller to send a byte of information. Therefore, you will need to send two nibbles (4-bit fields), one after the other.

The LCD interface can run on 3.3V, and therefore, there is no need for voltage conversion to shift signals between the microcontroller and the LCD. The specification you must follow includes the following port and pin assignments:

- PORTB to the LCD DB pins
- PA2 to EN
- PA3 to RS

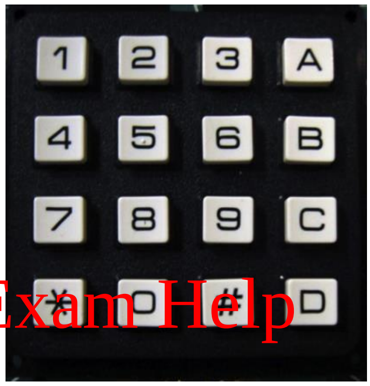
The LCD's R/W pin can be connected to GND, which fixes it at Write, as you will not be reading data from the LCD.

3 Software

When you write software professionally, you will be given a statement of the requirements your software must fulfil. It is often called a functional specification – it specifies what functions your software must perform, but not how it performs them. There could be a little or a lot of detail, anything from a simple statement of what the software must do down to specifying variable names. Your software might become part of a software library a[illegible]nform to some standard.

For this project you will sp[illegible]ur software must be divided into, the names of most of the *#define* constants, and [illegible]e functions.

3.1 The keypad



You will build a standard 1[illegible]own on the right. The keys are labelled with: 0 to 9, A[illegible]bviously, the ten digits are used for themselves. The others will be used for things like plus and equals.

In fact, this immediately presents us with a problem. The minimum requirement for the keys is the ten digits and plus, minus, multiply, divide, decimal point, clear (=rubout) and equals (=make the calculation). This lists 17 keys!

There are several solutions to this problem. The most common one is to use one of the keys as a shift key, as on many commercial calculators. This loses one key (there are now only five plus the digits) but doubles up the use of these five. It gives you what you need with a few spares.

The table below describes an example of how to configure each key, but you are free to propose your own.

Marking on keypad	When not shifted		When shifted		Notes
	Use	Display character [1]	Use	Display character [1]	
A	Plus	+	Times	x	[2,3]
B	Minus	-	Divide	/	[2]
C	Decimal point	.	Times ten to the power	E	[4,5]
D	Shift	[None]	Cancel shift	[None]	[6,7]
*	End Input	[None]	End Input	[None]	
#	Rubout last character	[None]	Delete entire entry	[None]	[4]

Notes:

[1] I.e. on the LCD display.

[2] Implementing this shifted function is essential.

[3] Times must be displayed on the LCD as lower-case x, not upper-case X or the asterisk (*).

[4] Implementing this shifted function is optional but will gain extra marks.

[5] For instance 1.2E3 means 1.2x10^3.

[6] This works like most calculators: you press shift, then release it, then press (e.g.) A for times. It is not like computer keyboards where you press both at once.

[7] Pressing shift a second time cancels it.

3.2 How to start the software part of this project.

3.2.1 Create Project

The first stage is to create a new Keil-v5 project using the Texas Instruments TM4C123GH6PM as target device. You would be wise to put it on your Uni network drive and access this from your laptop or a lab.

As part of this, you will create the standard main.c file.

3.2.2 Starting to write yo[illegible]

In practice, you will probab[illegible]ogramming work is of two types, with different styles of thinking. You can do them [illegible]ou may prefer to alternate between them.

- **Writing the progr[illegible]**

  You are now into [illegible]ing, and it would be wise to follow the divide-and-rule technique: it is eas[illegible]ctions, even if there are more of them, rather than a few enormous ones.

- **Writing all the #*define* statements** that specify port addresses and the initial contents of registers. Many of these can be copied/imported from previous lab work, though some will need changing.

3.2.3 A comment on clocks

There will be many time delays, so you will need the PLL and SysTick to generate them accurately. To ease calculations a setting of 80 MHz is suggested, though you may want to make your own decision.

Clocks control many things (especially LCD timings), so it is probably worth getting them working early.

3.2.4 A comment on the LCD

This is complicated to interface to. There is a lengthy description in Appendix C of this handout.

3.3 Project management tactics

Here are some hints to help you succeed.

3.3.1 What should I do first?

Do the essential things first. Leave the nice refinements for later.

3.3.2 I changed it and it broke!

Once you have a program that works, add things a bit at a time. At each step make sure it still works: if it doesn't, you can undo the last change. That way you always have something working.

3.3.3 My program is mis-behaving – how do I see what it's doing?

A good first step is to use the Keil facilities to place breakpoints and examine variables.

If that doesn't help, you can print debugging messages to the LCD. **Thus, it might be wise to get the LCD software working first.**

3.3.4 The KISS motto

"Keep It Simple, Stupid!" It's tempting to invent very complicated algorithms and code. Good program code is simple – simple enough to be understood by anyone or by you on a Monday morning when you're not really awake.

3.4 Extra marks

The following are considered to be additional tasks for which extra credit will be given:

- Adding a *password* to access the keypad/calculator, with the option for the user to change the password.
- Using the flash memor[illegible]ler, e.g. to store the password.
- Display graphics on th[illegible]
- Any additional tasks th[illegible]t the module leader's approval first).

These may require the use[illegible]e.g. shift-equals).

4 Appendix A - Making software device-independent

4.1 The problem

Suppose your TM4C123GH6PM microcontroller has an LED connected to an output bit of a port – consider the code to turn it on or off. This might include a declaration like

```
#define LED (*([illegible]d long *) 0x12345678))
```

and a command something [illegible]

```
LED = 0x04;
```

to turn it on. This is fine w[illegible] implemented on the same microcontroller. (Actually, it's not. If someone has to mo[illegible]ey might wonder why the LED gets the value 0x04, not just 1 or 0. And what if they set it to 0x01 by mistake - what would happen?).

But suppose you later want to port the software to another microcontroller, i.e. implement it on a different one. This might be because your firm has moved to a better microcontroller manufacturer, or to a newer microcontroller by the same manufacturer. This porting will involve two steps:

1. Understanding why the address was 0x12345678 and the output value was 0x04. Then working out the new ones. This work is inevitable.

2. Going through all the program code looking for anything that refers to LED and changing 0x04 to whatever the new value is. This is where it is very easy to miss things and make mistakes. It is also extra work.

With a simple example like this one LED it would not be too hard, but realistic programs have many of addresses (possibly dozens), all with their strange values to be sent to them.

The problem here is that much of the code is device-dependent – it depends on the specific device (here the TM4C123GH6PM microcontroller). It is far better if your program is as device-independent as possible.

4.2 The solution

... is to put all the device-dependent code in one place, in its own module. This module makes the device-dependent code available in a device-independent way, e.g.

```
void WriteLed(int value)
{
      if (value) // Any non-zero value will turn it on.
            LED   =   0x04;
      else  LED = 0x00;
}
```

This appears merely to replace one command with another - to say WriteLed(1); is no shorter than LED = 0x04; – but the advantage is that it is device-independent.

In this example, any non-zero parameter turns the LED on. In more complicated cases, you might want to include code to check that the parameter has a valid value.

There are possible objections to this practice: the extra function calls might reduce the program's efficiency. For comments on this, see Appendix B. Nevertheless, it is better to have a program which does the right thing slowly than one which does the wrong thing fast.

5 Appendix B – Efficiency concerns and coding style

You often find that a better (e.g. clearer) way of writing your code looks less efficient. It might take more CPU time and/or occupy more program memory space. Do these matter? In both cases, the answer is “it depends” – for some programs, it might, but for most, it probably doesn't.

5.1 CPU time

Your processor runs at a cl[...]megahertz. Even without checking, one would guess that the time to make an extra f[...] of the order of a microsecond or less. If the call is made 100,000 times a second, i[...]consider writing to the LCD display or reading from the keyboard – would you no[...]econd? Even if the job required a thousand accesses, would you notice a millisec[...]

5.2 Program memory space

The function calls will increase the program size (but see below), but probably only by a few bytes per call. Processors are bought with memory sizes increasing in large steps (e.g. 32k, 64k, 96k, ...). If the extra code happens to push the size over one of these boundaries, then the cost will increase, but the probability of this is slight. If you did find that your code was just over the size boundary then you could look into reducing it. (Actually, your first step would be to read up on your compiler's optimisation options.)

5.3 Avoiding these inefficiencies

The way you write your program controls what the C program is like, which is not necessarily what the machine code is like. That also depends on what the compiler does. If you define a function as inline the compiler will consider replacing the call with the program lines inside the function. For details, see your favourite C/C++ programming textbook.

Inlining avoids the time to call the function. As for program memory space, the contents of the function are repeated every time it would have been called. With a large function (including with parameter checks) this would increase memory space. If the function just contains a hardware I/O command (such as the LED = 0x04; above) then it would be about the same.

Also, modern compilers are very good at optimising code. Compare the following two examples to print a string. But before you look at the right-hand one, can you work out what the left-hand one does?


```
char *c = string;
while (*c)
      putc( *(c++) );
```

```
int i = 0;
while ( string[i] != ‘\0’ ) {
putc( string[i] );
i ++;
}
```

A good compiler would probably generate much the same machine code from both. But which is easier to understand? Which is less likely to generate bugs when someone changes the code? Or when it is first written?

With the left-hand one, for instance, what would have happened if the programmer had used ++c instead of c++? Or if they had omitted the inner brackets in the putc? Or if they had used the ++ in the while, not the putc? If you’re not certain, it’s probably an obscure feature of C/C++ and best avoided – it invites mistakes.

The current perception (actually, it’s been around since the 1980s or earlier) is that programs should be written in simple language, easy to understand. Even if the compiler doesn’t optimise well, it’s more important to avoid bugs.

In the nineteenth century, Charles Dickens wrote novels which showed off his ability to handle complicated English grammar. In the earlier days of computing, programmers wrote programs which showed off their ability to handle complicated ways of describing algorithms. Both are now regarded as bad practice.

6 Appendix C – Hints on handling the LCD

This appendix provides an overview of the functions involved in handling an external LCD. Of course, you will need to define your own. One of these, *InitDisplayPort()*, must be called first. The mechanism by which these functions send information to the LCD is slightly complicated and can be defined into another function called *Se*[illegible]s can be designed to handle almost all information sent to the LCD. The only exce[illegible]*nitDisplayPort()*, which is unusual and needs direct 4-bit access to the LCD port. T[illegible]ess using the 4-bit interface is shown on page 11 of the SPLC780D datasheet. Yo[illegible]at you will communicate with the LCD by sending **two nibbles** (4 bits) rather th[illegible]**te**. Therefore, another function can be defined called *SendDisplayNibble()*.

6.1 SendDisplayNibble()

This sends a nibble (4 bits, i.e. a half-byte) to the LCD. It uses two ports of the microcontroller:

- RS on bit 3 of Port A
- EN on bit 2 of Port A
- DB7-DB4 on the bits you choose of Port B.

This function has to do three things:

1. Set up the RS bit appropriately: 0 for instructions or 1 for data.
2. Send the nibble to the bits of the port.
3. Pulse the EN line for 450 ns.

The EN pulse needs extra comments. From the datasheet (Bus Timing Characteristics / Write Operation, page 21[1]) you will see you need a pulse width of at least 450 ns. Due to the amount of delays required for controlling the LCD, it would be a good idea to define a function that creates time delays.

6.2 SendDisplayByte()

This function has the job of sending an 8-bit quantity to the LCD. The eight bits must be send four at a time using *SendDisplayNibble()* twice. The upper four bits 4-7 are sent first on pins DB4 to DB7, respectively. After this, the second four bits 0-3 are sent on pins DB4 to DB7, respectively. Section 5.5 on page 9 of the datasheet shows an example.

After sending both nibbles, there must be another delay of 37 μs for the display to act on what it has received.

6.3 InitDisplayPort()

The basic information to understand this is in the SPLC780D.pdf file on page 11. You should read this, probably several times, until it makes sense. Part of the complication is because the SPLC780D powers up in 8-bit mode, so a 4-bit interface initially has to act in 8-bit mode to set the SPLC780D to 4-bit mode. This is possible because the 4 bits which are not connected are not needed for this initial instruction.

See also the instruction table on page 7 for more details.

Note that the diagram on page 11 shows six bits. The last four are the 4-bit output. The first two are RS and R/notW: RS is on a different port (along with ES), and R/notW is not used but hard-wired to zero. RS (Register Select) is 0 for instructions (1 for data), so for all these initialisation instructions it is 0. The notes on the right at the bottom of page 11 are badly typeset – you have to count paragraphs to see which table row they refer to.

[1] There is a similar table on p.22 with a different time, but that is for using a 5 V supply, so do not be misled by it.

The first four transmissions cannot use *SendDisplayByte()*, so they have to use *SendDisplayNibble()*. They must also wait for the times given on page 11. The delay after the third transmission is not given, so the standard 37 μs is a good guess.

Call your time delay function for all the required delays. You will need to define a time unit for this function, let's say microseconds, and allow it to accept an input parameter to control the delay created.

Looking at page 11, the f[illegible]ons of 0011 (with RS=0, R/W=0 and the other 4 bits unneeded) are the standar[illegible]are the same as for 8-bit mode (see page 10).

The fourth transmission of[illegible]r 4 bits un-needed) sets 4-bit mode. The rightmost 0 is what sets 4-bit mode.

So far, these instructions h[illegible]as one instruction, as the SPLC780D has assumed that all 8 bits were connected; i[illegible] the lower four bits.

From now on the display is in 4-bit mode, so instructions can be sent using *SendToDisplay()*.

The next transmission of 8 bits (sent as two nibbles) repeats the setting of 4-bit data but also specifies the number of display lines N and the font F. The values for these are given at the bottom of Table 6 on page 25. Your display has 5x8-pixel characters.

The last three initializations are clearly explained on page 11.

In addition to the above functions, you could define the follows:

- *clearDisplayScreen():* clear the LCD screen
- *moveDisplayCursor():* move the LCD cursor to a desired position
- *printDisplay():* print a string of characters on the LCD. This function could also handle the line in which the string is printed on

Lastly, the functions described in here are **suggestions** for you to get a general idea of the requirements when interfacing an external LCD to the microcontroller.

7 Assessment of the mini-project

The full detail of the mini project assessment can be found under the “Assessment” tab -> Mini Project Assessment.

In summary, a successful project (as can be demonstrated in a release/demo version video) should include:

- Displaying the resu[illegible]ut buttons (calculations) on the LCD screen to prove you have correctly impl[illegible]mini project functions.
- The calculator is e[illegible]oating-point calculations and nested calculations (more than two operands [illegible]r’s precedence).

Extra features/credits:
The followings are considered [illegible] tasks for which extra credits may be given:

- Adding a *password* to access the keypad/calculator, with the option for the user to change the password.
- Display graphics on the LCD.
- Any additional tasks that you find useful (get the module leader approval first)

8 Important Notice on Plagiarism

Please be reminded/warned that the School takes acts of plagiarism extremely seriously. In today’s Internet age, you will more than likely be able to find a solution to this mini-project online. The Module Leader will be coming around in the laboratory sessions to ask questions to check that you fully understand *every single line of code* that you write. If you are suspected of plagiarism, the Schools standard disciplinary procedures will be followed and if found guilty, the School will push for maximum penalty i.e. exclusion from University. Please do not be under any illusions that this is an idle threat - unfortunately there has been an increase in computer-code plagiarism in recent years which has resulted in several students being excluded from University.

You have been warned.