

Laboratory 3

Introduction to the Zilog C Development Environment and the La Trobe Keypad/LCD Development Board.

Introduction:

The purpose of this lab session is to explore the Zilog C development environment using parallel interfacing of peripherals to the Z8 Encore!TM processor on the **LaTrobe Keypad/LCD development board**. The software will be written using the C programming language and compiled using the GNU C cross compiler within the Zilog, “ZDSII” Integrated Development Environment (IDE).

Aim:

The two main aims of this lab session are:

1. Learn the basic skills to use Zilog ZDSII IDE effectively.
2. Revise/practice C programming within the scope of an embedded microcontroller by modifying supplied sample ‘c’ code.

Resources:

Most resources should be available in the ZiLOG directory of your local machine. There is an archive which contains the specific files you will need for this lab. See the section “Method” for instructions on downloading these files.

Minimal specific knowledge of the Z8encore processor will be required for this lab.

SCHEMATICS – can be found at the rear of this document. Supplied are 2 sheets showing the full schematic of the **LaTrobe Keypad/LCD development board**.

Requirements:

Setup the IDE and write out answers to the 5 questions in task 2 as you work through the lab. Ensure that your work for all tasks and subsections is seen and marked off by a demonstrator *during the lab*.

Submit a brief report including the answers to the 5 questions in task 2 and a listing of your code from task 3.

Background (from ZiLOG):

The Z8 Encore developer's environment is a tightly coupled collection of software development tools under the direction of a developer's environment, similar to Microsoft's Developer Studio. The developer's environment lets you configure, manage, and execute each of the tools in the system. The Z8 Encore developer's environment is for the Z8 Encore family of processors. Using the Z8 Encore developer's environment, embedded applications are developed more efficiently using the following tools:

- Developer's environment
- Programmer's editor
- Macro Assembler
- ANSI C-Compiler
- Linker/locator
- Integrated Command Processor
- Automated make facility
- Debug mode
- Instruction Simulator
- Emulator drivers

These tools are briefly discussed in the **ZiLOG Developer Studio II Z8 Encore!TM User Manual** (.pdf file available under documentation).

The ZiLOG Integrated Development Environment provides several Project Workspace windows:

- Edit window
- Build Output window
- Debug Output window
- Find in Files Output windows
- Messages Output window
- Command Output window

For more detailed information and instruction on getting started read through the ZiLOG Developer Studio II Z8 Encore!TM User Manual. For a detailed discussion on all the window options and dialog boxes, see page 46.

Method:

TASK 1 – Introduction to the Integrated Development Environment (IDE)

(Total marks: 0 – hurdle requirement)

*NOTE: The screenshots and instructions given in this section are for **version 4.9.5** of the Zilog Development software. A different version may be installed in the laboratory, and so there may be minor differences from the instructions given here to what is required in the laboratory.*

Create a new folder called “lab3” inside your ELE3EMP laboratory folder, i.e. create:

H:\Your_Home_Directory\ELE3EMP\labs\lab3

Go to the website <http://www.ee.latrobe.edu.au/~gt/ele3emp/> and download the following files to your lab3 folder.

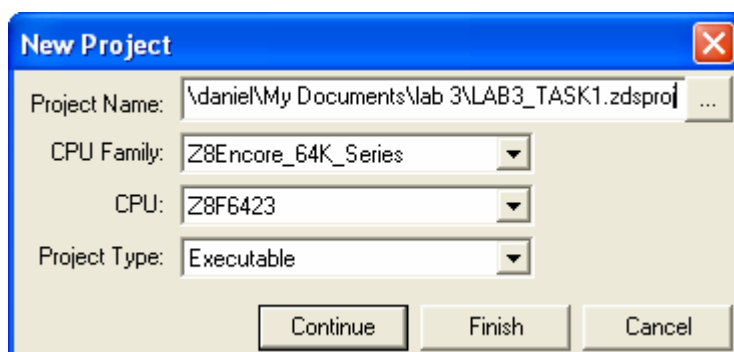
- LAB3-TASK1-SAMPLE-2007.C
- The Z8 Encore!™ manuals

Now copy LAB3-TASK1-SAMPLE-2007.C to a new file named LAB3-TASK1.C so that you have the original handy, in case of mistakes.


Task 1 is basically very simple:

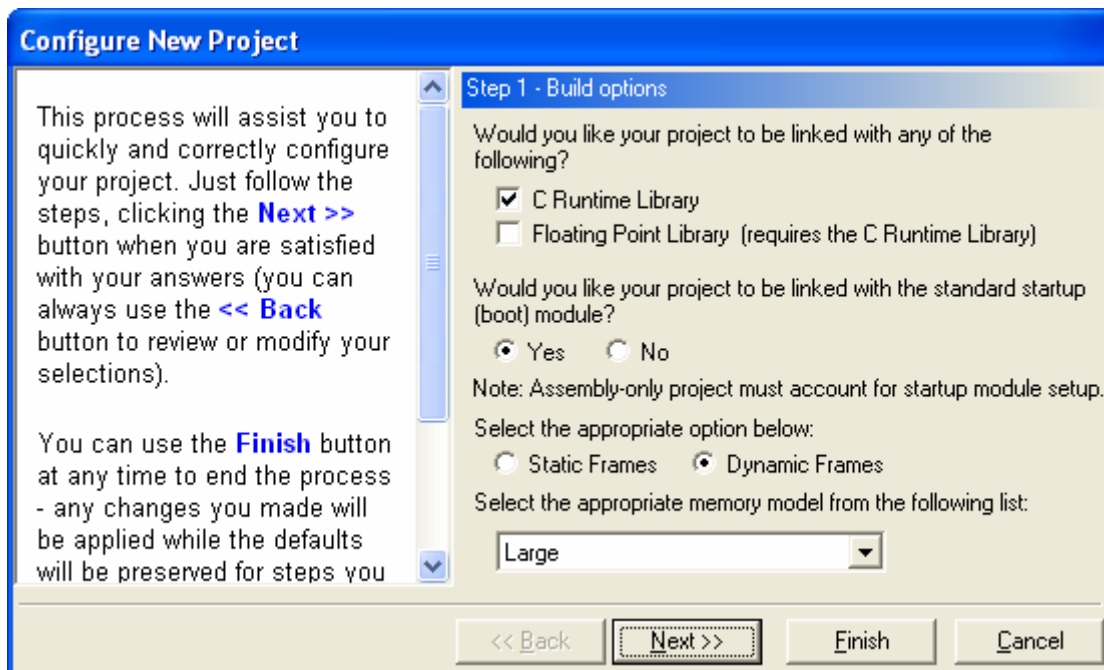
- (1) Create a new project (.zdsproj file) and run this sample code on the LaTrobe Keypad/LCD development board.
- (2) Answer some questions on the operation of the ZDS II IDE software and the operation of the C program.

So to begin, open ZiLOG ZDSII and **create a new project of your own** (call it what you like – we suggest something like LAB3_TASK1.zdsproj) in the directory that contains the unzipped sample C source file.

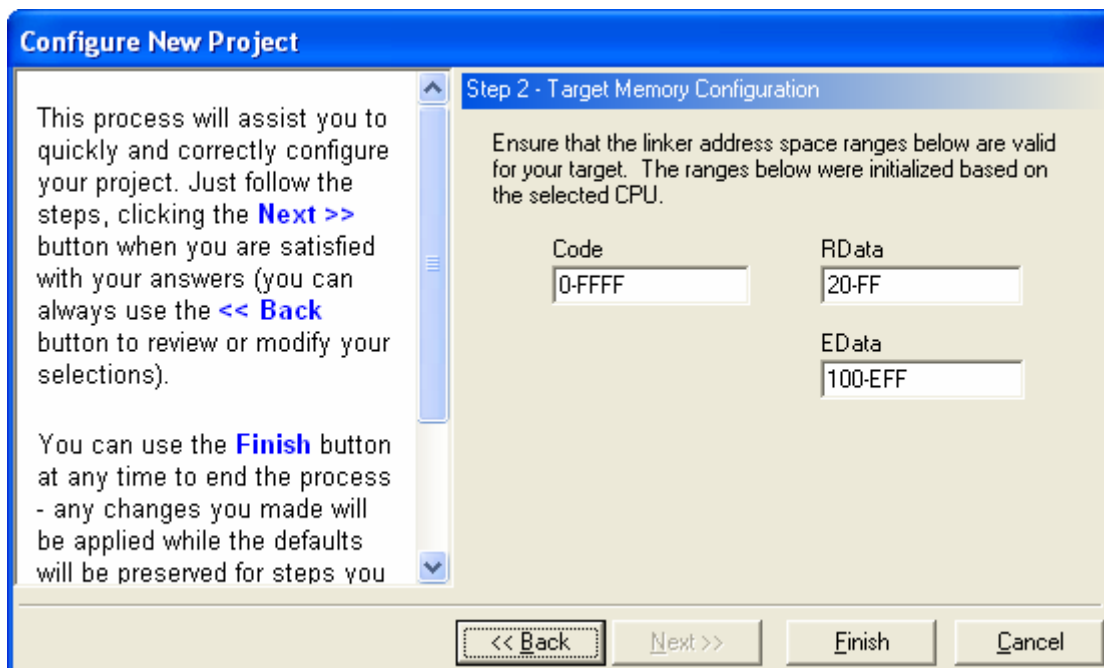


NOTE: The LaTrobe Keypad/LCD development board uses an in house built ‘Z8 chip carrier’ which uses a microcontroller based on the Z8F6423 revision of the Zilog CPU.

Click  and read the next text box (*next page*):



The options shown here are what you require (dynamic frames and large model). Once you have read through this click **Next >>** to reveal the final step in the initial project setup.

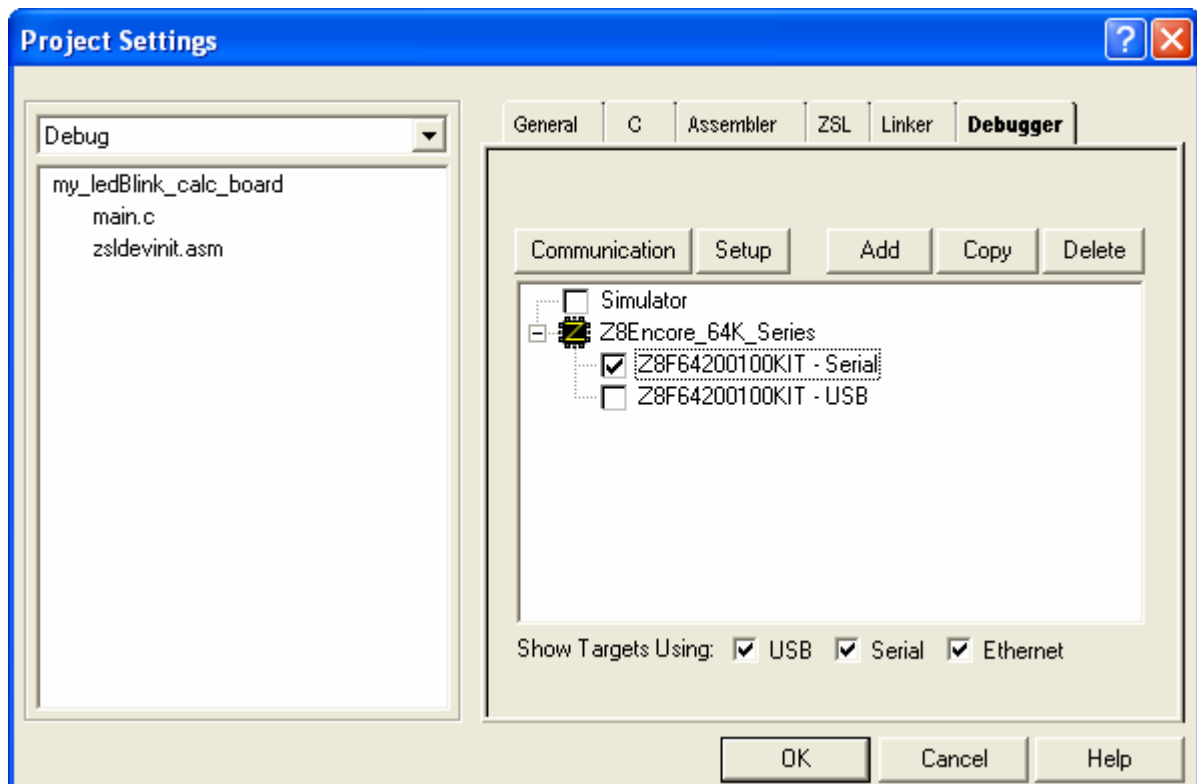


Check that the default memory settings are correct and click **Finish**. Now you have the basic project created.

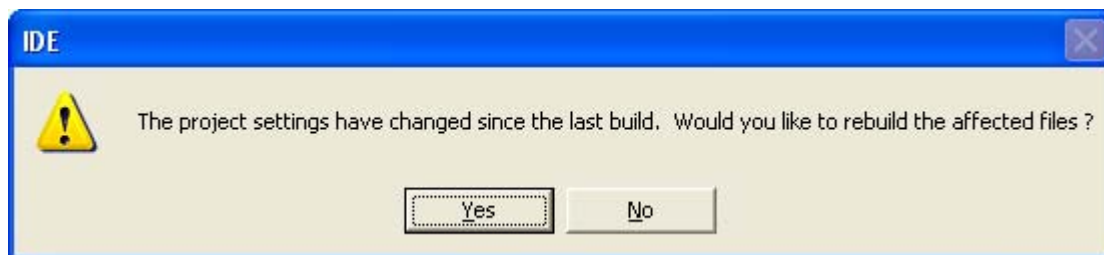
You could write your own code, but to start, use the supplied C file. Right click the white space in the right hand project window (Project Workspace pane) and select “Add Files To Project...”. Choose the file ‘LAB3-TASK1.C’.

The default build target for the compiler is the simulator. We want to build for hardware! Make sure the development board is connected to the serial port of the PC by the La Trobe

Serial Programming Adapter. We select the build options (among other things) with the menu Project=>Settings (Alt+F7). Explore the project options available. When you have satisfied your curiosity, select the debugger tab, then select the serial interface to the hardware as shown:



Now we should also add the dependency files to the project. However, the compiler package does this for you on first compilation, so just click . You may at this point receive a pop-up message like this...



Just click as we want to do this using the main menu in the next step. Project setup is complete. Be sure to save your work at this point.


End of TASK 1


Task 2 – Compiling, Building, Running and Stepping.


(Total marks: 5)

Be sure that the LAB3-TASK1.C file you just added to the project is open. If it's not, double click the file in the Project Workspace Pane to open it. Compile the file – either click the icon or use the Build=>Compile menu item.

Make the project – either click the Build button  or use the Build=>Build menu item. You should have no errors and the messages “Linking...” & “build completed.” at the bottom of the screen in the output window.

Connect to the target – either click on the Connect To Target button  or use Build=>Debug=>Connect To Target menu item. When the connection is established you will be able to see details of the connected microcontroller within the output window.

Now the Z8 CPU is under the control of the Zilog IDE's debugger – awaiting commands. Download the code generated by the build process – either click on the ‘Download Code’ button  or use Build=>Debug=>Download Code menu item. You will see a progress meter appear showing the code is being loaded into the microcontroller's memory. The program is relatively small so this will occur very quickly.

The code will not spontaneously run after downloading since the CPU is still under the control of the debugger. To run your program either click on the GO button  or use Build=>Debug=>Go menu item.

Does it work?

– The 4 red LEDs should be flashing – in unison with a duty cycle $\approx 50\%$.

Open the Zilog start-up assembly file ‘zsldevinit.asm’ (Note: This file is called ‘startup.asm’ in earlier versions of the development software) and examine it.

Expand the external dependencies in the project window by clicking on the ‘+’ sign. Examine this list of files.

Question 1.1 (1 mark)


Describe in a paragraph or two what is happening in all the processes relating to loading running the file above. That is, describe what *compile*, *build*, *download* and *run* steps actually do.

Question 1.2 (1 mark)

- i) Examine main.c. Explain what it is doing and give a brief description of how the three functions below actually work:
 - turn_off_LEDs()
 - led_bit_pattern (byte)
 - delay()
- ii) Give the name of the four registers associated with the Z8's General Purpose Input Output port G. Describe how these are used to configure the various port options.


- iii) What is the meaning of the word “blocking” in the delay() function header (description)?

Question 1.3 (1 mark)


First, stop the processor running – to do this click on the pause button  or use the Build=>Debug=>Break menu item.

Set a **breakpoint** at line 28 (next to init_IO();) in main() by double-clicking in the greyish margin along the far right of the code window. A red dot will appear at this location.

Next reset the CPU’s program counter:


Click on the reset button  in the toolbar or use Build=>Debug=>Reset menu item.

Now instruct the CPU to run to this breakpoint:



Click on the Go button  in the toolbar or use Build=>Debug=>Go (F5) and keep an eye on the LEDs.

What happens? – Hopefully nothing, initially. But notice (on the monitor) the IDE indicates that the CPU has executed everything **up to** but **not including** the line with the yellow arrow.

Short answers will suffice for the following two items...


- i) Click the Step Over button  in the toolbar ONCE (*and only once*) while watching the LEDs. What happens? You can find out and answer *why* this happens in the next part of this question...
- ii) Now click Step Over again – ONCE. What has happened now?

After ii) the LEDs should have all been OFF. Note that this ‘flash’ of the LEDs was not the same flashing you see when the program is running. To get a closer look at what is happening here we will next use the **Step Into** debugging function.



Again, reset the CPU’s program counter in order to begin stepping through the program from the same breakpoint – to do this click on the reset button  followed by the Go button . The yellow arrow should be sitting, pointing at init_IO(); again.


Question 1.4 (1 mark)

Step Into method allows you to observe the step by step execution of the ‘c’ code line by line including jumps into and back from function calls. This is different to the **Step Over** method which will execute, and return from, a function call in one click of the Step Over button.

With the program waiting at the breakpoint, click the Step Into button  and watch as the yellow arrow progresses through the code. Keep an eye on the LEDs each time you click the Step Into button – pay special attention to the exact step which causes them to illuminate for the first time and which step causes them to turn off again for the first time. You will have to press the Step Into button at least 12 times to see the proper results.

- i) Which line causes the LEDs turn on? Why do they turn on? You should be able to see why from the code comments – something which was *not apparent* when we were using the Step Over function above.

Now we will look at the 3rd kind of step – the **Step Out** function. We will set a new breakpoint for this part. Reset the CPU’s program counter by pressing the debugger/CPU reset button . Now double-click the red dot in the margin next to `init_IO()`; – it will disappear. Set a new breakpoint at line 30. Now press the Go button . The yellow arrow will jump to the line that begins `delay()`; and waits.

Press the Step Into button to enter the `delay()`; function. To complete this function step by step we would have to press Step Into or Step Over thousands of times! The Step Out function will cause the CPU to execute all instructions constituting the nested for loops and will return the program counter (and update the IDE display) to the point from which the function was called. Try pressing the Step Out button  at this point to observe this in action.

- ii) What is the difference between the Step Over, Step Into and Step Out functions of the IDE software? When might each be useful?

Reset the CPU as you have done previously and with the ‘LAB3_TASK1_main.c’ as the active window set a breakpoint at line 81 – the line with `PGOUT = ~(bit_pattern) ...`. Now open the main menu View=>Debug Windows=>Disassembly. Watch this window as you hit the **GO** button.

Question 1.5 (1 mark)

Complete the following regarding the current view of the disassembly code that you have just revealed following the above steps:

- i) Draw a table like the one shown and fill it out with the assembly instructions that implement the C code segment:
`PGOUT = (PGIN & ~LED_bits) | (~bit_pattern & LED_bits);`
shown in the assembly window.

Start Memory Location	Hex Data Content	Equivalent Opcode
-----	-----	-----

- ii) From the above, use the Z86423 data sheet to determine how many CPU clock cycles it would take to set the PORTG LEDs to display the binary 'bit_pattern'. Keep in mind that the Z8 CPU has an instruction *pipeline* which allows memory (new instructions and/or data) to be fetched as (concurrently with) instructions are being executed. For this reason consider only *instruction cycles* from the datasheet.

- iii) The chip carrier you are using has a 18.432 MHz crystal oscillator – how long would it take (in μsec) to for the PORTG bits to be set as a result of the 'c' assignment statement?

End of TASK 2

Task 3 – Programming The Keypad/LCD Board Using C

(Total marks: 5)

Bit Masking

Before you continue you will need to revise a common programming method called “bit masking”. In embedded systems programming, bit masking is often used to protect and or ignore particular register bits during byte read/write cycles. When data is being written, bit masking can preserve particular bits within a register from being overwritten. Likewise this method can be used to ignore particular bits within a register during a read.

(aside – on Z8 GPIO)

Within the Z8 MCU we have a physical registers called PGIN and PGOUT (see ZiLOG Z86423 datasheet page 24) which are each 8 bits wide. The value of these registers determine which pins of port A are set (on), and which are clear (off) depending on the configuration of the port G Data Direction Register (DDR). Each port has its own DDR which determines which pin is an input and which is an output. PortX DDR bitY controls the function of PortX pinY as follows:

$PortX\ DDR\ bitY = 1 \rightarrow PortX\ pinY = input$

$PortX\ DDR\ bitY = 0 \rightarrow PortX\ pinY = output$ (see ZiLOG Z86423 datasheet page 57)

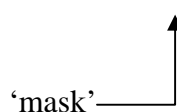
We will look at an example of bit masking making reference to the schematics at the rear of this lab sheet. Look at the schematic and you’ll notice that the CATHODES of the four red LEDs, Led[1..4] are connected to port G pins, PG[0..3]. In order to turn on and off the LEDs one or more at a time, we need to write ‘0’^s and ‘1’^s to the appropriate bits of the PGOUT register. To turn a LED off we would write a ‘1’ to the corresponding bit within the lower order nibble of the PGOUT register. You can see this had been done in the turn_off_LEDs() function in sample_main.c. This simple implementation of turn_off_LEDs() works adequately because there are no other OUTPUTs connected to bits PG[4..7] (actually they’re inputs – switches). If there were other outputs connected here then their bits would be turned off by the turn_off_LEDs() function. This would be undesirable in almost all cases.

We can solve this problem using **bit masking** to affect only the bits we’re interested in PG[0..3]). Consider the following line of code:

```
PGOUT = 0x0F;           // naive turn_off_LEDs() function
```

If PGOUT has the binary value ‘01101010’ then LEDs 1 and 3 will be illuminated. The direct assignment of 0x0F above causes the upper nibble of PGOUT to be reset, and the lower nibble to be set. This indeed causes Led[1..4] to turn off – BUT we have overwritten and lost information from the upper nibble which may have been used for some other purpose. This problem can be resolved by using the following **bit masking** code instead:

```
PGOUT = (PGIN & 0xF0) | 0x0F;           // modified turn_off_LEDs() function  
                                           // only affects PG[0..3]
```



The bitwise AND, 'PGIN & 0xF0' resolves to the binary value '01100000'. The lower nibble of port G has been cleared without losing existing binary values within the upper nibble. This result is then ORed with 0x0F and this new result is stored into PGOUT via the assignment operator '='. After this re-assignment, the new binary value of PGOUT is '01101111', and all LEDs are off and the original state of the upper nibble is preserved.

Similarly, to turn all the LEDs on, we *could* write:

```
PGOUT = (PGIN & 0xF0) | 0x00;    // current flows from Vcc through the
                                   // LED and to GND through the Z8
```

OR, more simply, since ORing with 0x00 has no effect:

```
PGOUT = PGIN & 0xF0;           // current flows from Vcc through the
                                   // LED and to GND through the Z8
```

Keep It Simple!

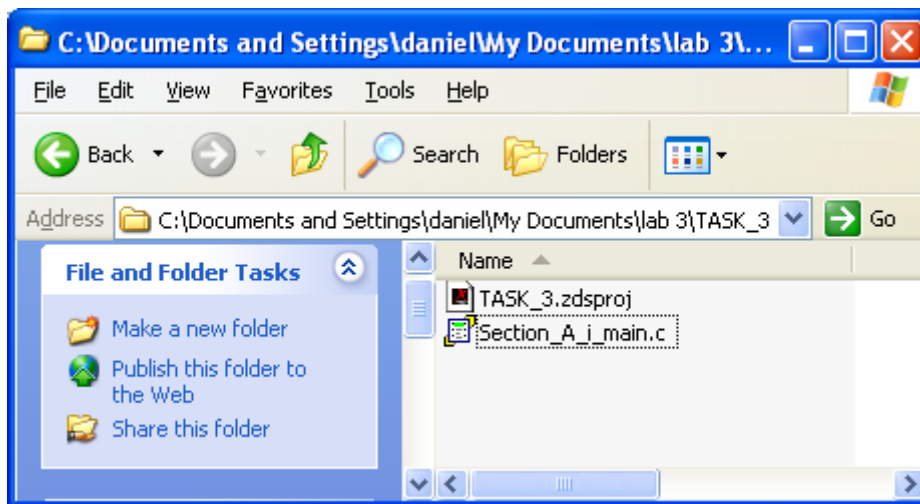
Bit masking can often be done in several ways – some solutions are contorted and confusing while others are simple & elegant – all of which may achieve the same result. You need to be careful that you don't over-complicate the process. Use a pen and piece of paper if necessary! Always review your solution (Boolean algebra) to make sure it is optimal.

In the next task you will be asked to write and run programs within the ZDS II IDE using the skills you have already acquired. This task is broken into sections A & B.

Section A – Basic Control of the LEDs

(Total marks: 3)

To begin with create a new sub-folder of this lab, say, TASK_3. Make a new project within this new folder – call it something like ‘TASK_3.zdsproj’. Once you have done this you should have something like this:



As you do the various sections create new appropriately named ‘.c’ files and add/remove them from your TASK_3.zdsproj project. Then compile and run normally. The idea is to use one project file and change the ‘.c’ files you compile within it.

Here you are required to write/alter the sample code to create an LED ‘seconds’ counter.

i) (1 mark)

First, alter the `delay()` function so that it takes an integer input parameter that represents the number of milliseconds (approx.) of delay. To do this, introduce a *nested* `for(;;)` loop structure within the `delay()` function, and add input data types etc. to its header and prototype.

ii) (2 marks)

Next write a new function that will display a counter represented as a binary number on the LEDs, `Led[1..4]`. Back to the task – The number mentioned above should represent the approximate number of elapsed seconds. There are only 4 LEDs so make sure your counting variable is always between 0 – 15 inclusive. You will need to be familiar with bit masking to complete this task.

Using External ‘.c’ and ‘.h’ Files

This has been discussed briefly in lectures, but here’s a reminder...

When creating new functions that are to be called from main it is a good idea to make a new appropriately named ‘.c’ file to put your functions in. All functions relating to say, reading

the keypad would go into a file called 'keypad.c'. In order that the compiler can find the called functions from the main.c file you must also include your own header file 'keypad.h'. This header file will provide the compiler with knowledge of the new function prototypes pertaining to the external keypad functions.

Section B – Incorporating Pushbutton Functions

(Total marks: 2)

Here you are again required to write new functions that will be used in the already edited the sample code. As mentioned above, you should create a new '.c' file called, say, 'buttons.c' in which you can write your pushbutton related code.

i) *(1 mark)*

Write a function that reads the state of the push buttons S2 and S3. The function should return a value representing one of 3 possible results – whether S2 is pressed, whether S3 is pressed or, whether *both* are pressed.

Have your function print the binary numbers 0001, 0010, 0011 according to the state of the pushbuttons on the LEDs. NOTE: your representation of the binary number will most likely be mirrored due to the physical layout of the LEDs on the development board.

ii) *(1 mark)*

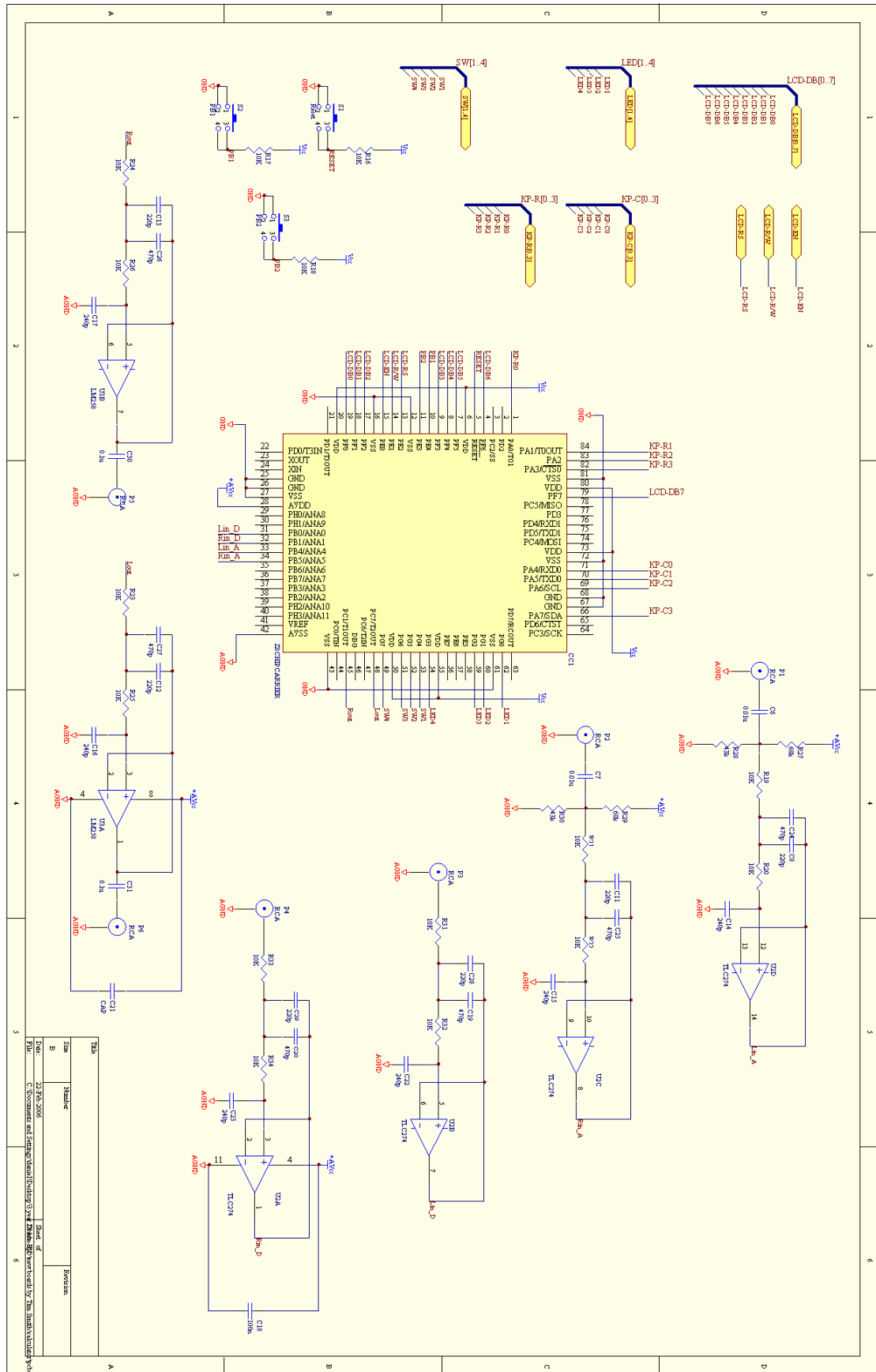
The final task is to add an up/down counter function to your program. The counter value (0-15) should be displayed on the LEDs. When S3 is pressed, the count value should increment by 1. When S2 is pressed the value should decrement by 1. When both buttons are pressed the count should reset to zero. You can create this function within a 'counter.c' file or just add it into 'LAB3_TASK3_main.c' if you like.

DJS – February 21 2006

GT – 15, 30 March 2007 Revised

Appendix – Schematic

LaTrobe Keypad/LCD development board 1 of 2



Appendix – Schematic

LaTrobe Keypad/LCD development board 2 of 2

