EE8205: Embedded Computer System

Electrical and Computer Engineering, Ryerson University

Introduction to Keil uVision and ARM Cortex M3

1. Objectives

The purpose of this lab is to introduce students to the Keil uVision IDE, the ARM Cortex M3 architecture, and some of its features. Specifically, the basic steps of coding and execution with the ARM Cortex M3 and its NXP LPC1768 microcontroller will be provided, including how to run a simple program on the MCB1700 dev board. The lab will allow students to become familiar with the uVision environment, its simulating capabilities, and the tools needed to assess various CPU performance factors. As majority of embedded systems use ARM based for low-power consumption and competitive performance, students will find the skill sets obtained from this lab very useful.

2. Developing Software for Cortex with Keil uVision

In this section, you will learn how to create a uVision project, import necessary files, compile, and simulate an application to assess performance. In particular, this example will demonstrate a simple project called *blinky*. The code will read the voltage provided by the microcontroller's ADC channel AIN2 (the potentiometer found on the MCB1700 board). Based on the value set on the channel, the LEDs will flash at a certain speed. If enabled, a bar graph and voltage reading will also appear on the LCD display.

2.1. Creating a new uVision Project

We will be working with the **NXP LPC1768** chip for this lab. This chip can be found on the Keil MCB1700 evaluation board located at your workstation.

To run uVision IDE on the EE network, launch **Windows XP** through **VMWare**. Once Windows is running, you will find the uVision program on your desktop. Open the application.

Fig. 1: uVision Icon

1. When uVision has launched, select Project >> New uVision Project in the main menu bar. If a project already exists, first close the project by selecting Project >> Close Project.

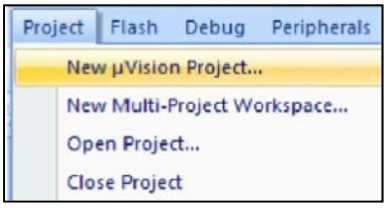

Fig 2: New uVision Project

2. Create a "EE8205" folder in your "**S:**\\" directory to successfully save the project in your EE account. **If you do not save your project in the S:\\..\ directory, your project will be lost when you logoff.**
3. You should see a window appearing similar to the one shown in Fig 3. Select the icon for "New Folder" and name your working folder "Blinky".

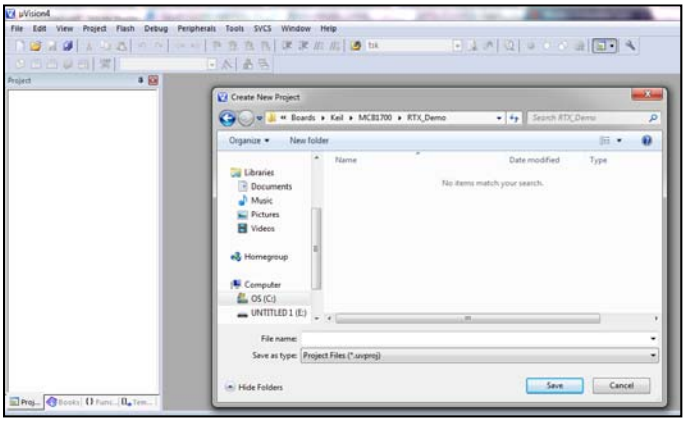

Fig 3: Create New Project

4. Double click the folder to enter the directory.

5. Name your project "Blinky" and click save. This directory will contain the source code file s necessary for your project.

6. Next you will see a window open prompting to "Select Device for 'Target 1'..." as seen in Fig 4. This window is the Keil Device Database and lists all devices that Keil supports in uVision.

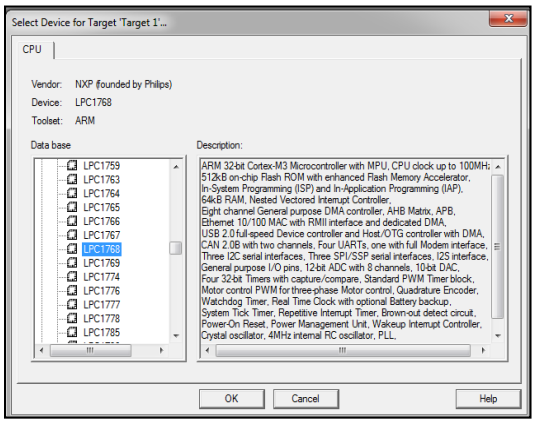

Fig 4: Selecting LPC1768 Target Device

7. Under "Data base", locate the **NXP** directory. Expand the directory and select **LPC1768**. Once selected, a description of the processor should be displayed in the "Description" box. Click OK.

8. Next, a window will open asking whether to copy the default LPC17xx.s startup code to your project folder and add the file to your project. Click "Yes".

9. Your **uVision project workspace** is found in the left column window entitled "Project", with a folder in it called "Target1". Ensure that the *LPC17xx.s* file was added to the project by clicking the "+" sign beside the "Target1" >> "Source Group 1" folders. The source file *startup_LPC17xx.s* is used to *initiate* our target hardware LPC1768 (the Cortex-M3's microcontroller unit on the MCB1700 dev board). You may now begin to use the full capabilities of the IDE.

10. Click *once* on the "Target1" field in the left column (with the field highlighted first). Rename the field to "LPC1700". Press ENTER. Your workspace should now resemble Fig 5.

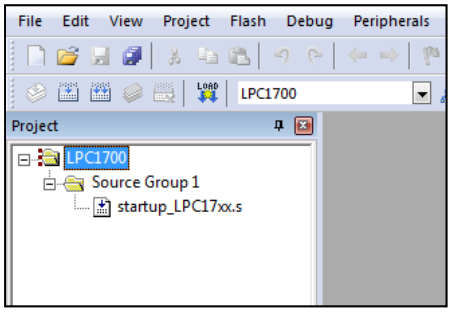

Fig 5: Initial Project Structure (Project Workspace)

2.2 The uVision Workspace and Building the Project

1. Using Microsoft Explore, browse to the "**U:\\ee8205\project\labs\uvision**" directory to find, copy and paste ALL the files to your project directory. There should be a total of 11 files copied.

2. In the project workspace (found in the upper left column of uVision), right-click on the "Source Group 1" folder and select the option **Add files to Group "Source Group 1..."**. Select **all** the **.c and .h files**.

3. Click "Add" and then "Close". All the files selected should now be displayed in the Project workspace under the "Source Group 1" folder tree, resembling that of Fig. 6.

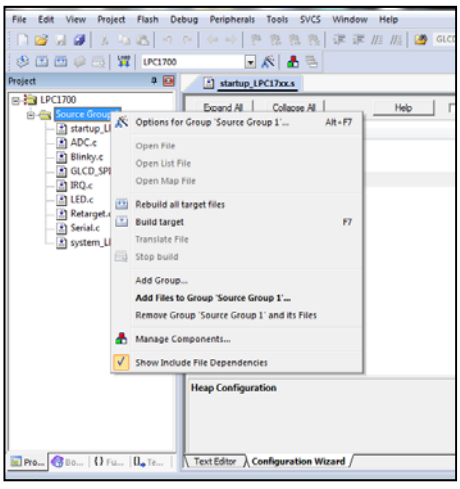

Fig 6: Importing all .c and .h files to uVision

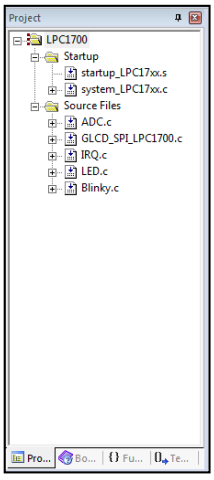

Fig.7: Final Workspace

4. Right click the LPC1700 folder and select "Add Group". Another folder will appear as a sub-tree under LPC1700. Name it "Source Files".

5. With the "Source Group" folder highlighted, click on it once and rename it as "Startup".

6. Drag all the files **except** "startup_LPC17xx.s" and "system_LPC17xx.c" to the "Source Files" folder. Your workspace should now resemble Fig. 7.

7. Double click on *startup_LPC17xx.s* to open the editor. Click on the "**Configuration Wizard**" tab at the bottom of the editor window. The Wizard window converts the "Text Editor" window so that the programmer may view the configuration options more easily. It is possible to adjust the stack and heap sizes of the LCP1768 chip here if necessary.

8. Similarly by clicking on the "**Books**" tab at the bottom of the **Project workspace window**, the "Complete User Guide Selection" opens up to provide you with FAQs and system help. Once you have finished inspecting the user guide, switch back to the "Project" tab in the Project window.

9. During the first part of this lab, we will be simulating the blinky.c program. Thus we must define certain preprocessor symbols for the compiler to interpret. In your main menu, select Project >> Options for Target 'LPC1700'. Click the tab entitled "C/C++".

10. In the box "Preprocessor Symbols", write "__ADC_IRQ" in the textbox *Define*. Click OK.

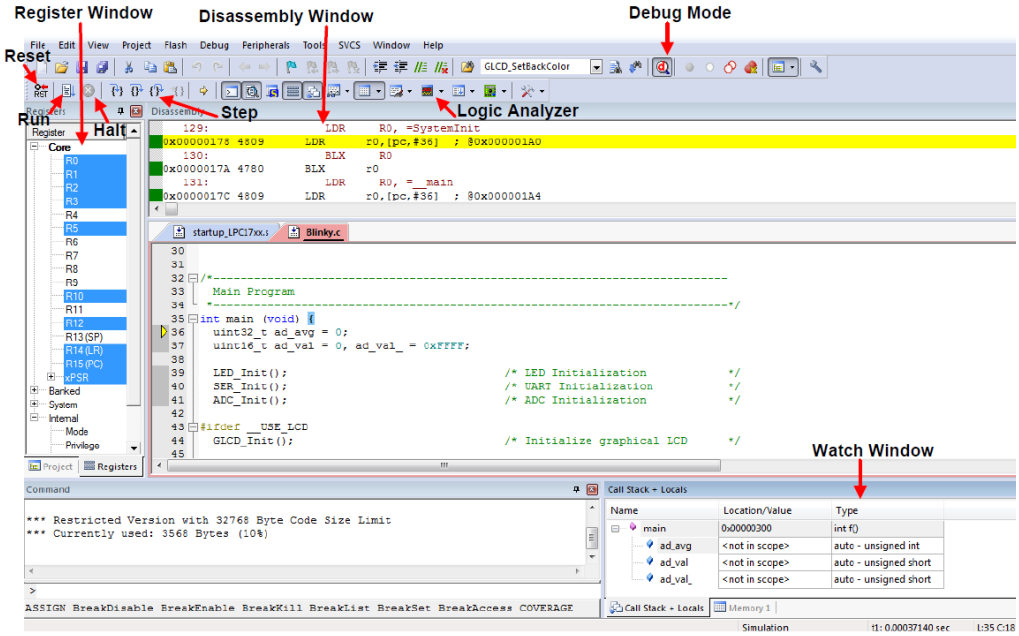

Fig. 8: uVision Debug Mode

11. Enabling printf: Project >> Options for Target 'LPC1700'. Select the Debug tab, select "Use" on the right side, and then click the Settings box. Under the Trace tab, click "Trace Enable". Ensure that the Core Clock is set to 96 MHz, and that the SWO Clock has "Autodetect" enabled. In the ITM Stimulus Ports, set Enable to 0xFFFFFFFF, and ensure that the lower port checkbox, Port 7..0 is unchecked. Click OK. In the "Options..." window, select "Use Simulator" once again. Click OK. Also notice the source code necessary in Blinky.c to support the printf function.

12. To compile and link the .c modules, click the build icon  . You can alternatively build the project by pressing F7. Ensure that the project compiles and links without any errors or warnings. A *newline at end of file* warning may appear: this is fine.

Side Note: Examining the Application

Before we continue to work with Debug mode, it is important to understand what each part of the Blinky.c application is responsible for. Take a minute to analyze the code provided to you. In particular, examine blinky.c, ADC.c, IRQ.c, LED.c. How do they work together? What are their functionalities?

Browse through the NXP LP17xx user manual for the board found in the COE718 labs directory. You will need to refer to this manual for this lab and future labs as well. Note the specifications which pertain to the AD conversion (ADC.c) and GPIO (LED.c) programming pins used for this lab in the code provided to you. For further elaboration on the programming of these pins, refer to the Appendix of this lab.

Blinky.c - main file, initializes the LED, Serial and ADC functions. To use the LCD uncomment #define _USE_LCD before main(). Sets up the timer interrupt every 10ms. Reads and averages AD conversion for LED.c to interpret. Prints out AD value to both Debug (printf) stdout (using a clock_ms timer flag) and the LCD if enabled. Examine the ITM Stimulus code for printf functionality in uVision.

ADC.c - Initializes ADC channels and variables used by Blinky.c. Contains an interrupt handler for ADC which clears the ADC flag and stores the converted value.

IRQ.c - Contains the timer interrupt handler routine needed by blinky.c. It is responsible for keeping track of clock_ms (10 ms timer flag) and the LED blinking rate.

LED.c - initializes the LEDs and contains routines to turn the LEDs on and off. Also contains the function needed by IRQ to adjust the speed of the LEDs according to the ADC converted value.

GLCD_SPI_LPC1700.c - contains all functions needed to use the MCB1700 dev board's LCD screen.

2.3 Simulation with Debug Mode

Next, we will enter **Debug mode**. Debug mode is an environment that provides capabilities to assess your application and its performance characteristics.

1. Enter Debug mode by clicking on the  icon. A window will pop up displaying: "EVALUATION MODE Running with Code Size Limit:32K" Click Ok. uVision will transform into a new succession of windows, including the disassembled version of your .c code. If you have entered the Debug mode correctly, you will see a number of windows pop up which will allow you to examine and control the execution of your code. You should observe something similar to that of Fig. 8.The Debug mode will connect uVision to a simulation model of your program, downloading the project's image into the microcontroller's simulated memory.
2. Reset the program using the  icon.
3. Execute the program by clicking the  RUN icon. STOP (or pause) the program by selecting the  icon.

Congratulations, you've executed your first program on uVision. Now, what do all these windows in Debug mode actually do? What does this all mean?

2.4 uVision Debug Features and Analysis

uVision possesses many features for assessing the status and performance of your application software running in Cortex. The following is a list of useful features that can be used to view and control your applications. Note that they can only be accessed when in Debug mode.

a) Watch Window

A watch window allows you to keep track and view local and global variables, as well as raw memory values. These values can be observed by running or stepping through your program. It may be beneficial to pair the watch window with the use of steps and/or breakpoints in your code for debugging.

Note on steps and breakpoints:

- Steps - (See Fig. 8) As opposed to running through the whole code, the step keys allow you to step through your code line by line, step through a function etc.
- Breakpoints - Move the mouse cursor into the grey area next to the line numbers in your .c code in the debugger. Left click the line (with a dark grey area) that you would like to set a breakpoint. A red dot will appear if you were successful. Click it again to remove the breakpoint.
    - Note when the code is executed, the dark grey boxes will turn green indicating that the line has been executed.

1. To open a watch window (in Debug mode), select View >> Watch Window >> Watch 1. Note, a watch window may open up automatically when entering Debug mode.

2. Find the column entitled "Name" in the Watch 1 window. The subsequent rows under this column should read <Enter expression>. Highlight the field and press backspace. Enter "ADC_Dbg" in the first row.
    - When you click the RUN icon to execute the program, the value of ADC_Dbg will change depending on the ADC value inputted on analog channel 2 (AIN2). (More on inputting analog input in the *Peripheral* section)
    - To automatically input a variable in the watch window, go to the blinky.c code. Right-click on the variable *AD_Dbg*. A pop-up menu will appear. Select "Add *ADC_Dbg* to..." >> Watch 1.

3. It is also possible to change ADC_Dbg's value during execution. If you enter a '0' in the value field of the watch window, you may modify the variable's value without affecting the CPU cycles during executing.

b) Register Window

The register window (see Fig. 8) displays the contents of the CPU's register file (R0 - R15), the program status register (xPSR), the main stack pointer (MSP) and the process stack pointer (PSP). This window will automatically open when transitioning to Debug mode. These registers may be used for debugging purposes, in conjunction with the watch window, steps, and breakpoints.

c) Disassembly Window

The disassembly window displays the low level assembly code, where its respective C code is appended beside it as a comment. This window is useful for viewing compiler optimizations and the .c code's assembly generation. The left margin of the disassembly window is also useful for keeping track of execution (green blocks), possible executable blocks (grey), line numbers, and setting breakpoints.

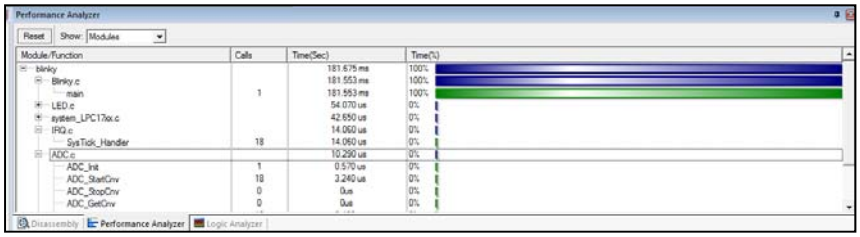

Fig. 9: Performance Analyzer Window

d) Performance Analyzer

The Performance Analyzer (PA) tool is extremely useful for determining the time your program spends executing a certain task. It presents itself as a horizontal bar graph dynamically changing with respect to the total execution time of its respective tasks. Separate columns display the exact execution time and the number of calls for each task. To use this feature:

1. (In Debug mode) Select View >> Analysis Windows >> Performance Analyzer. Alternatively you can select the  icon's downward arrow, and select Performance Analyzer. A new PA window should appear.
2. Expand some of the tasks in the PA window by pressing the "+" sign located next to the heading. There should be a list of functions present under this heading tree.
3. Press the  Reset icon to reset the program (ensure that the program has been stopped first). Click RUN .
4. Watch the program execution and how the functions are called. You will see something similar to that of Fig. 9. The analyzer is able to gather various statistics dynamically from the program, which may be useful for both debugging and performance assessment. Stop the program when you have finished analyzing with the PA tool.

e) Execution Profiling

An alternative to the PA is the Execution Profiling (EP) tool. EP is useful for determining how many times a function call has occurred and/or the total time spent executing a certain line of code and/or function. Therefore, the PA tool would technically be the graphical representation of the EP tool. To use this feature:

1. From the menu select Debug >> Execution Profiling >> and either Show Times or Show Calls. A left column will expand on your source code, indicating either the execution time per task, or the number of calls respectively.
2. Regardless of the option you choose, if you hover the mouse over a number in the left column, all the information will be displayed as if you chose both options (i.e execution time and the number of calls).

f) Logic Analyzer

The Logic analyzer in debug mode allows you to visualize a logic trace for a variable during its execution. Thus we could use this as a visualization for the variables we place in the watch window. For this lab, we will graphically follow the AD_Dbg value in our code:

1. Press the  arrow on the icon, and select Logic analyzer. A window will appear (if not already present).
2. In the blinky.c code, right-click on the variable AD_Dbg. A pop-up menu will appear. Select "Add *AD_Dbg* to..." >> Logic Analyzer. The variable will appear in the Logic analyzer window.

3. If you click run, you will see the AD_Dbg trace generate as a straight line on the zero mark. It should correspond to the value you are seeing in the Watch 1 window.

4. Under the Zoom heading in the Logic analyzer, click "All". This will scale your window according to the execution trace time (horizontally).

5. Under the Min/Max heading, select "Auto" to scale the trace's amplitude (vertically).

This AD_Dbg value will keep running with a zero value. Why? The AD_Dbg is representative of the value which we place on the board's potentiometer (AD input channel 2). Since we are not inputting any values on the channel, it will logically continue to trace at '0'. It is evident how we would go about turning the potentiometer on the dev board, but how could we simulate the pot for testing in Debug mode?

g) Peripherals (A/D Converter, System Tick Timer, and GPIOs)

uVision debugger allows you to model the microcontroller's peripherals. With peripheral modeling, it is possible to adjust input states of the peripherals and examine outputs generated from your program. In our Blinky.c program, the peripherals of interest are the AD converter (since we are inputting AD values from AIN2 - pot), the systick timer, and the GPIOs (the output to the LEDs). We will not model the LCD in this lab as it possesses high CPU utilization times and is more for demo purposes. Therefore make sure that *#define _USE_LCD* remains commented in the code during debugging.

1. To open the AD Converter window, in the main menu select Peripherals >> A/D Converter. A window will appear similar to that of Fig. 10.

2. To open the System Tick Timer window, in the main menu select Peripherals >> Core Peripherals >> System Tick Timer. A window will appear resembling Fig. 11.

3. To open the GPIO 2 analyzer (LED simulator), select Peripherals >> GPIO Fast Interface >> Port 2. A window will appear as shown in Fig. 12. Also open Port 1, i.e. Peripherals >> GPIO Fast Interface >> Port 1 (as the first 3 LEDs belong to Port1, last 5 belong to Port 2).

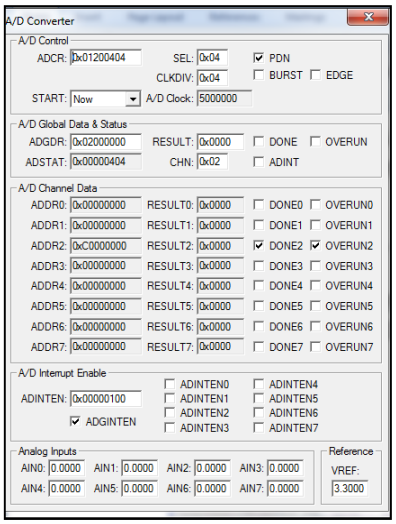

Fig. 10: A/D Converter Window

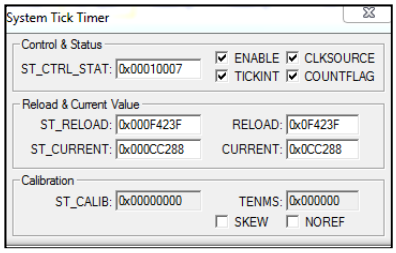

Fig. 11: System Tick Timer Window

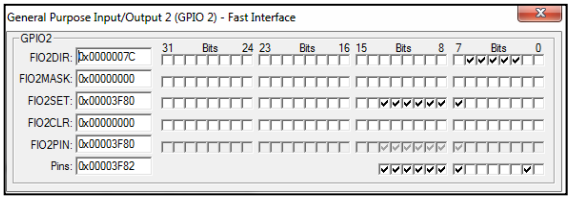

Fig. 12: GPIO Peripheral Window

4. To open the Debug window and view printf statements in the code, select View >> Serial Windows >> Debug (printf) Viewer.

5. Reset the program, and run the blinky application until it has **simulated** for 1 second. Watch how the asserted "Pins" on the GPIO windows transition. This represents the LEDs on the dev board

and how they will transition when the program is executed. Note that in reality, these transitions are occurring at a much faster speed than they are during this simulation. Why?

- Simulators require long computational runtimes to simulate a short period of hardware runtime. This is a well-known problem in software.

6. Notice the System Tick Timer and its quick transitions within each of the fields in the window.
7. Once 1 second of simulation time has been reached, there are two possible ways to change the value of the simulated pot:
   - Locate the A/D converter window and type 3.3000 in the AIN2 textbox under "Analog Inputs". This will simulate the value transition for your pot from 0V to 3.3V (notice the Vref voltage of 3.3V, which cannot be exceeded).
   - Alternatively in the "Command window" found in the debugger, type "AIN2 = 3.3". This will execute the same result as the A/D converter window.
8. Now interrupt enable must be asserted to simulate the value inputted on the AIN2 channel. To enable the interrupt, locate the A/D Interrupt Enable box in the A/D Converter window. Check off the ADINTEN2 box. Wait for a moment. Uncheck the box.
9. Wait for approximately 0.1sec (simulated time) or so. Your logic analyzer and watch windows will update the inputted A/D information accordingly (Note this transition may take slightly longer. To speedup the process, you may also click and unclick the "BURST" checkbox at the top of the A/D Interrupt window).
10. Note the GPIO windows and how the speed of the LED flashes have also changed (will transition at a slower pace).
11. Keep transitioning to different values using this simulated potentiometer method. Your simulation should then resemble close to Fig. 13.

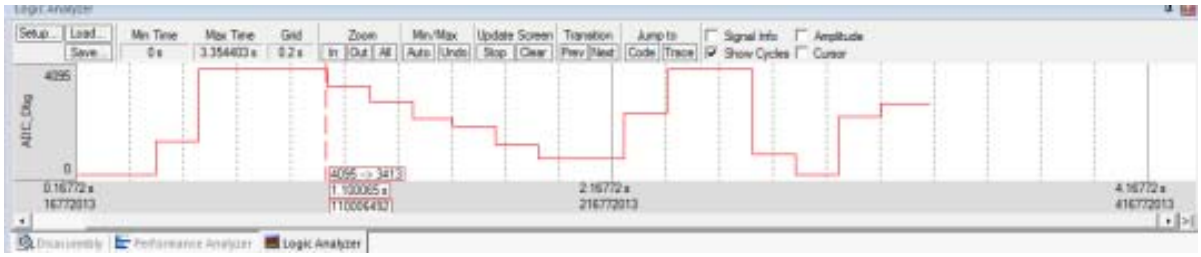

Fig. 13: Simulating the Pot and A/D Conversion using Logic Analyzer

12. While your program continues to execute, watch the application using the Performance Analyzer, Watch window, execution times in the Disassembly window, and the Execution Profiler. This will help you analyze the application. Where does your program spend most of its time executing?
13. Once you have finished executing your simulated blinky application, exit Debug mode by clicking the  icon once again.
14. Try executing bliny with the LCD display (i.e. uncomment the statement **#define _USE_LCD** in blinky.c). Go back to debug mode and execute the program. See how the utilization times of your applications differ from the results you obtained by not including the LCD.

3. Executing with Target Mode (MCB1700 Dev Board)

Up until this point, we have simulated the application. In this section we will focus on how to use target mode to actually execute blinky.c on the Keil MCB1700 development board. To upload the program to the board's flash memory:

1. Since we want to use the LCD during the demo, ensure that the statement **#define _USE_LCD** in blinky.c is uncommented.
2. Build the program to check for errors. Ensure that there are no errors.
3. With the dev board powered on, click on the load 

button located on top of the Project Workspace column on the left side.
4. The bottom of uVision will present a horizontal blue bar line indicating the status of uploading the software to the board:

```
Build target 'LPC1700'
linking...
Program Size: Code=5420 RO-data=6540 RW-data=40 ZI-data=608
"blinky.axf" - 0 Error(s), 0 Warning(s).
Load "S:\\...\\blinky.axf"
Erase Done.
Programming Done.
Verify OK.
```

   Ensure a loading failure has not occurred. If an error has occurred, refer to the troubleshooting section.

5. Your LCD and the rest of the board peripherals will appear blank. Press the **reset** button on the board.
6. Writing should appear on the LCD, and the LEDs should be flashing at a constant rate.
7. Locate the potentiometer and rotate it either clockwise or counter clockwise.
8. This rotation will provide the board with a variable A/D reading (varying the resistance of the voltage), which will be converted to a hex value (digital reading) and displayed on the LCD screen. A bar graph will transition underneath accordingly. Similarly the LEDs will flash at a different rate depending on the reading of the channel.

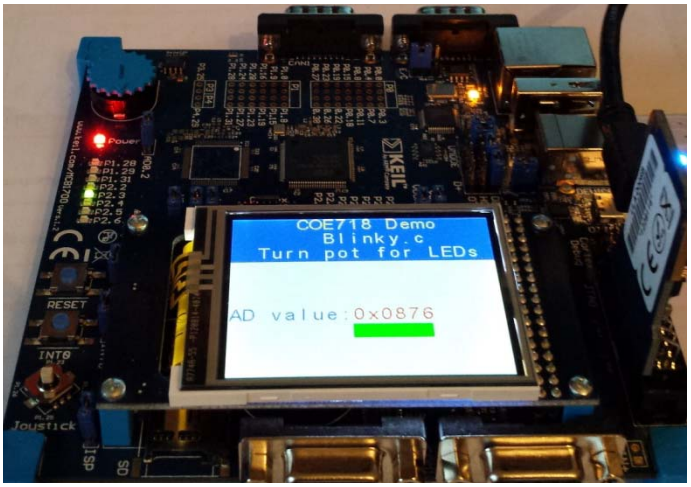

Fig. 14: Executing Blinky.c on the MCB1700 Dev Board

Troubleshoot Note: If the board is powered on, all the USB`s are connected and there still is a problem uploading the program to Flash, it is possible that this is a VMWare connectivity issue. In the VMWare window, select Virtual Machine >> Removable Devices >> Ulink-Me >> Connect (Disconnect from Host). Try loading the program using the previous steps once again.

4. Optional Lab Assignment

With the code given to you in this tutorial, and the joystick (KBD.*) files and peripheral notes found in the Appendix of this lab, create a program which will read the direction that the joystick is pressed on the MCB1700 dev board. Based on the direction of the joystick, the following peripherals should function as follows:

- LCD - will display the last direction that the joystick has been tilted/pressed. Design a suitable header and title on your LCD for demo purposes.
- LEDs - depending on the direction the joystick has been last tilted/pressed, an LED(s) will turn on representing the direction. Examine LED.c from the tutorial to understand how LEDs are turned on and off.

Have an option which enables or disables the LCD similar to the blinky tutorial. Disable the LCD when in Debug mode. Analyze your program (your TA may ask you questions regarding simulation during your demo).

Show the working of this lab in week 4 or 5 of the course. You are expected to provide the following:

1. Print the source code for your lab, including the main files, and any .h or .c files provided to you during the tutorial that you may have altered for your application.
2. Present a demo displaying your joystick implementation, and the output to the LCD and LEDs on the development board.

References

1. "The Keil RTX Real Time Operating System and µVision" www.keil.com. Keil an ARM Company.

2. "Keil µVision and Microsemi SmartFusion" - *Cortex-M3 Lab* by Robert Boys www.keil.com.

Acknowledgement

This tutorial has been adapted from introductory notes by Robert Boys "Cortex-M3 Lab" and "The Keil RTX Real Time Operating System and µVision" available at www.keil.com.
Keil is an ARM Company

Appendix

Peripheral Programming with the LPC1768

Peripheral pins on the LPC1768 are divided into 5 ports ranging from 0 to 4. Thus during the course of this lab you may have noticed that pin naming conventions (for GPIOs etc) were in the format Pi.j, where i is the port number and j in the pin number. For instance, if we take a look at the first LED on the MCB1700 dev board, we will see the label P1.28, signifying that the LED can be found on Port 1, Pin 28. A pin may also take on any one of four operating modes: GPIO (default), first alternate function, second alternate function, and third alternate function. It is important to note that only pins on Ports 0 - 2 can generate interrupts.

To use the peripherals provided to you on the dev board, ensure that you abide by the following steps. Let us take KBD.c as an example which can be found at the end of this Appendix. Note: masking register bits with |= (...) will turn the specified port pins high, while &= ~(...) will alternatively place them as low.

1) Power up the Peripheral

Looking at the NXP LPC17XX User Manual provided to you in the course directory, refer to Chapter 4: Clocking and Power Control (in particular pp. 63). The PCONP register is responsible for powering up various peripherals on the board, represented as a total of 32 bits.

The joystick is considered as a GPIO and therefore we are concerned with bit 15 for powering up. Note that the default value is '1' when the chip is reset. Thus GPIOs are powered up by default on reset. When coding for KBD_Init() we must then include the following code to power up the GPIO:

```
LPC_SC->PCONP      |= (1 << 15);
```

2) Specify the operating mode

The pins that need to be used by the peripherals must be connected to a Pin Connect Block (LPC_PINCON macro in LPC17xx.h). The registers which connect the peripheral pins to the LPC_PINCON are referred to as PINSEL, containing 11 registers in total.

The joystick pins are located on Port 1, pins 20, 23, 24, 25, and 26 (verify on the dev board). Referring to the manual (i.e. Table 82 on pp. 109) we observe that PINSEL3 is responsible for configuring these pin functions. Thus we include the following in KBD.c:

```
/* P1.20, P1.23..26 is GPIO (Joystick) */
  LPC_PINCON->PINSEL3 &= ~((3<< 8)|(3<<14)|(3<<16)|(3<<18)|(3<<20));
```

These pins are automatically selected as GPIOs upon reset according to Table 82. Thus we keep the "00" value assigned to them (re-declare these values as good practice).

3) Specify the direction of the pin

The I/O direction of the peripheral pins must also be specified (input/output). The FIODIR registers are used to set pin directions accordingly, where '0' represents input, and '1' is output. By default all registers are assigned as input. As the joystick is on port 1 in the LPC1768, we can configure specific pins as input as follows (pins on the LPC_GPIO1 macro):

```
/* P1.20, P1.23..26 is input */
  LPC_GPIO1->FIODIR   &= ~((1<<20)|(1<<23)|(1<<24)|(1<<25)|(1<<26));
```

```
/*----------------------------------------------------------------------------
 * Name:    KBD.c
 * Purpose: MCB1700 low level Joystick
 * Version: V2.0
 *----------------------------------------------------------------------------
 * This file is part of the uVision/ARM development tools.
 * This software may only be used under the terms of a valid, current,
 * end user licence from KEIL for a compatible version of KEIL software
 * development tools. Nothing else gives you the right to use this software.
 *
 * This software is supplied "AS IS" without warranties of any kind.
 *
 * Copyright (c) 2008 Keil - An ARM Company. All rights reserved.
 *----------------------------------------------------------------------------
 * History:
 *          V2.0 - updated by Anita Tino for LPC1768
 *----------------------------------------------------------------------------*/

#include <LPC17xx.H>                    /* LPC17xx definitions                */
#include "KBD.h"

uint32_t KBD_val  = 0;

/*----------------------------------------------------------------------------
  initialize Joystick
 *----------------------------------------------------------------------------*/
void KBD_Init (void) {

  LPC_SC->PCONP     |= (1 << 15);            /* enable power to GPIO & IOCON  */


/* P1.20, P1.23..26 is GPIO (Joystick) */
  LPC_PINCON->PINSEL3 &= ~((3<< 8)|(3<<14)|(3<<16)|(3<<18)|(3<<20));

/* P1.20, P1.23..26 is input */
  LPC_GPIO1->FIODIR   &= ~((1<<20)|(1<<23)|(1<<24)|(1<<25)|(1<<26));
}

/*----------------------------------------------------------------------------
  Get Joystick value.. part of get_button
 *----------------------------------------------------------------------------*/
uint32_t KBD_get  (void) {
  uint32_t kbd_val;

  kbd_val = (LPC_GPIO1->FIOPIN >> 20) & KBD_MASK;
  return (kbd_val);
}

/*----------------------------------------------------------------------------
  Get Joystick value
 *----------------------------------------------------------------------------*/
uint32_t get_button (void) {
   uint32_t val = 0;

   val = KBD_get();                         /* read Joystick state    */
   val = (~val & KBD_MASK);                /* key pressed is read as a non '0' value*/

   return (val);
}
```

```
/*----------------------------------------------------------------------------
 * Name:    KBD.h
 * Purpose: MCB1700 low level Joystick definitions
 * Version: V2.00
 * Note(s): Positioning of Joystick on MCB1700
 *           Revised by: Anita Tino
 *----------------------------------------------------------------------------
 * This file is part of the uVision/ARM development tools.
 * This software may only be used under the terms of a valid, current,
 * end user licence from KEIL for a compatible version of KEIL software
 * development tools. Nothing else gives you the right to use this software.
 *
 * Copyright (c) 2008 Keil - An ARM Company. All rights reserved.
 *----------------------------------------------------------------------------*/

#ifndef __KBD_H
#define __KBD_H

#define KBD_SELECT      0x01
#define KBD_UP          0x08
#define KBD_RIGHT       0x10
#define KBD_DOWN        0x20
#define KBD_LEFT        0x40
#define KBD_MASK        0x79

extern uint32_t KBD_val;

extern void     KBD_Init(void);
extern uint32_t KBD_get (void);
extern uint32_t get_button (void);

#endif
```