EE8205: Embedded Computer Systems
Electrical, Computer & Biomedical Engineering, Ryerson University

Introduction to Keil uVision and ARM Cortex M3

1. Objectives

The purpose of this lab is to introduce students with the installation of uVision on their home computers. They will also be made known to the Keil uVision IDE along with the ARM Cortex M3 architecture and some of its features. Specifically, the basic steps of coding and execution with the ARM Cortex M3 (NXP LPC1768) embedded processor. Students will learn to execute simple Cortex M3 programs using uVision. The lab will allow students to become familiar with the uVision environment, its simulating capabilities, and the tools needed to assess various Cortex M3-CPU performance features and factors. As majority of embedded systems use ARM processors for low-power consumption and competitive performance, students will find the skills obtained from this lab very useful.

2. KEIL uVision 5 Installation

The first step is to download the MDK531.EXE or the latest version available. Follow the link below **https://www.keil.com/demo/eval/arm.htm** fill the form and you will see the download screen of Figure 1. Download the MDK531.EXE file to your machine.
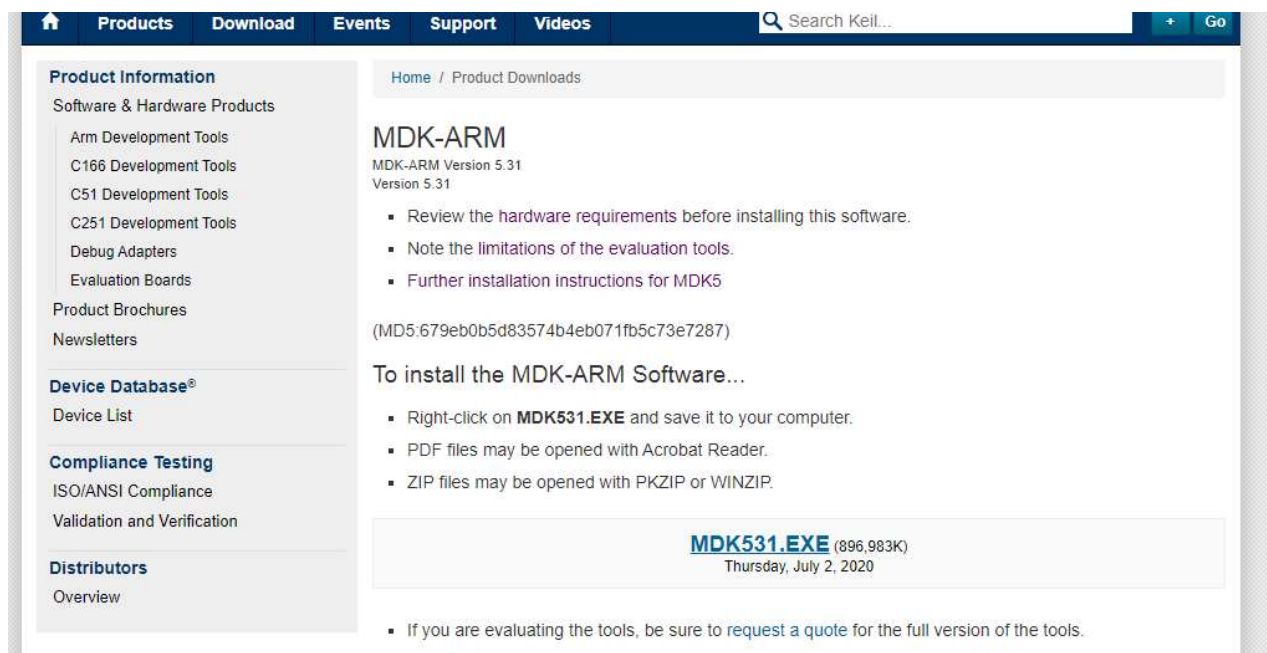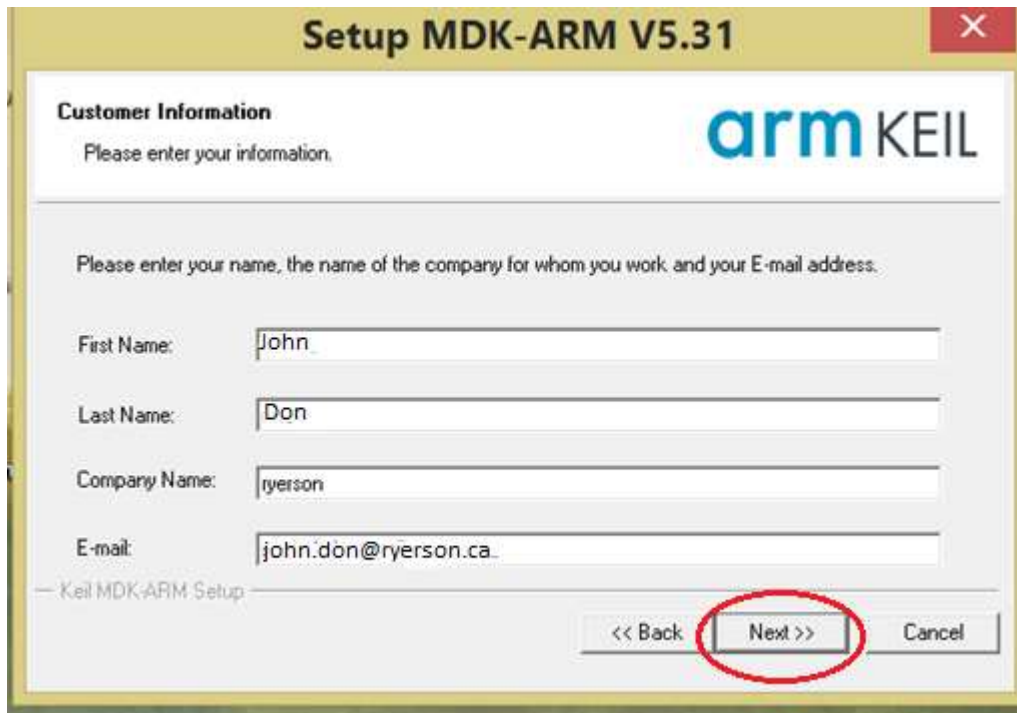

Figure 1: KIEL uVision 5 Downalod page

Once the file is downloaded, double click and run the setup file (MDK531.EXE). Accept the license agreement, select installation folder, enter your information, and complete MDK setup as depicted in Figures 2 to 7.

Figure 2: Setup Options

Figure 3: Program Directory

Figure 4: Your Information

Figure 5: Setup Running

All the Cortex M3 based labs are remote and online only and we cannot use the Cortex M3 hardware boards in Lab ENG408. Therefore, we do not need the ULINK driver. However, if any student wants to obtain/buy Cortex M3 (or M4) processor development board, he/she should install the driver also.

Figure 6: Device Driver

Figure 7: MDK Installation Completed

3. Package Installation:

Select **NXP LPC1768** processor from the Devices tab (See Figure 8) as this is the processor used in the Cortex M3 based MCB1700 boards available in our ENG408 lab. Install all the packages as mentioned in Figure 9.
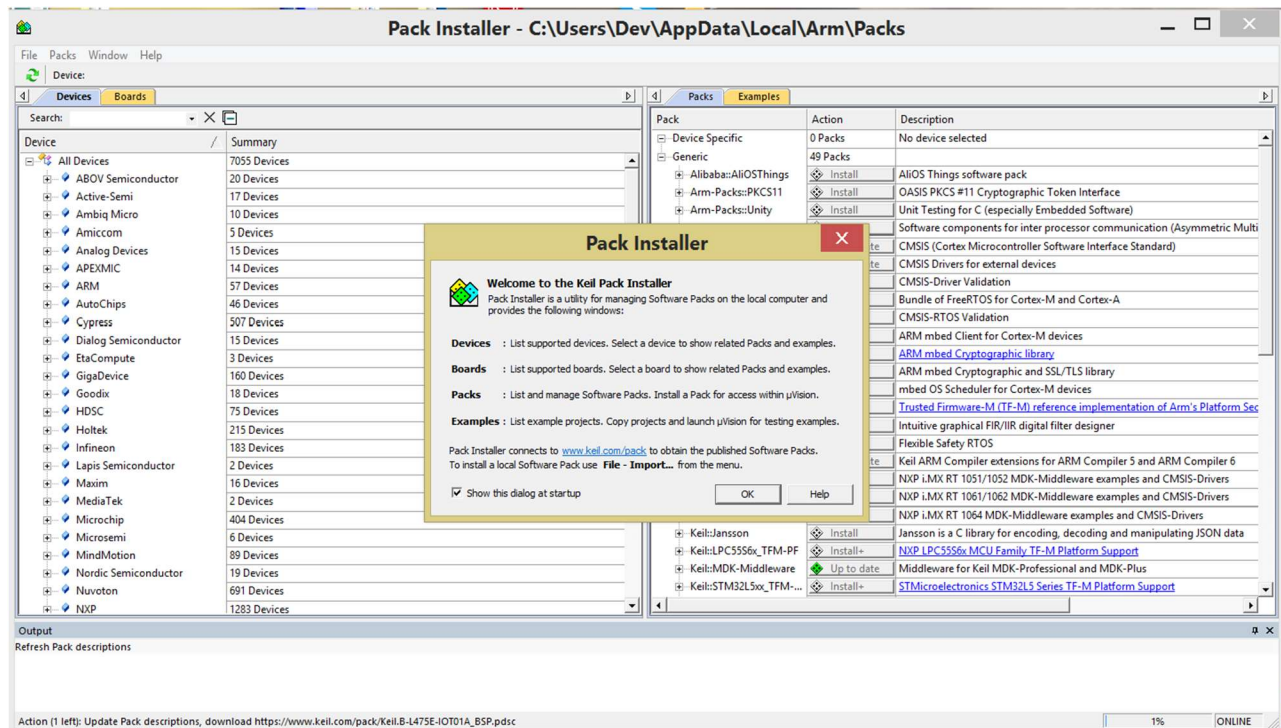
Figure 8: Package Installation
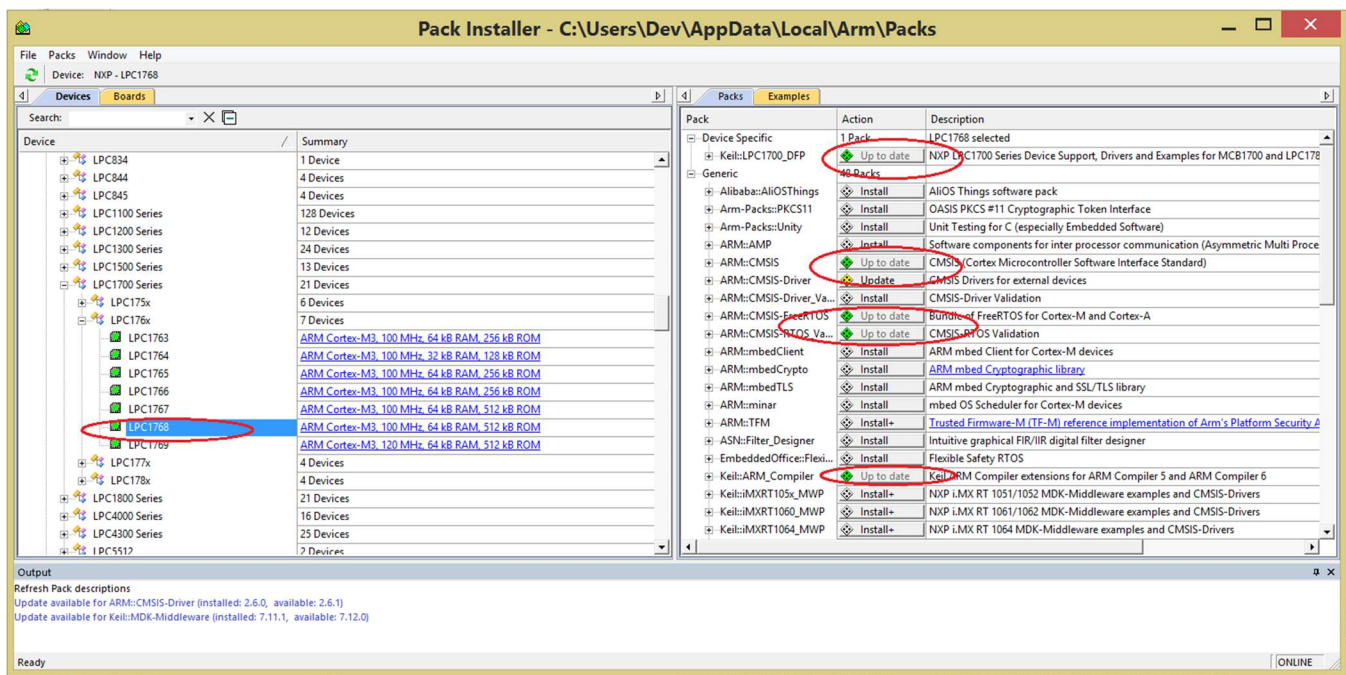
Figure 9: Device and Supported Package Installation

4. Developing Software for Cortex with Keil uVision5
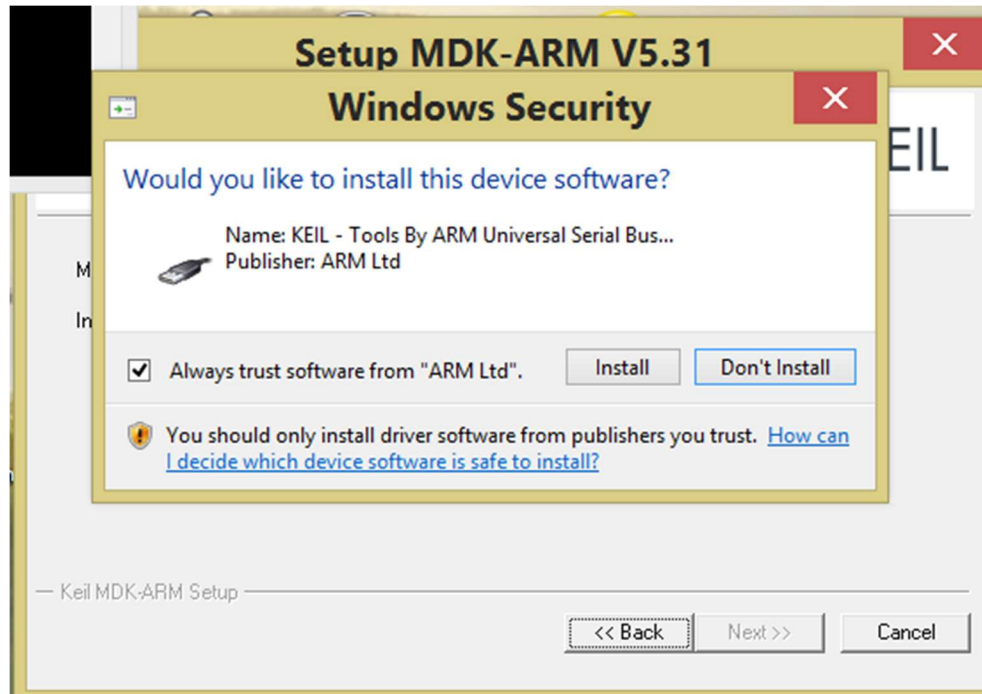
In this section, you will learn to create a uVision project, import necessary files, compile, and simulate an application to assess the performance. In particular, the example will demonstrate a simple project called *blinky*. The code will read the voltage provided by the microcontroller's ADC channel AIN2 (the potentiometer available 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.

4.1. Creating a new uVision Project

We will be working with the **NXP LPC1768** (Cortex M3 produced by NXP) processor in these labs. This processor chip is used in the Keil MCB1700 evaluation board. You will find a lot of online resources and tutorials for assistance.

To run uVision IDE, double click uVision program on your desktop. Open the application.
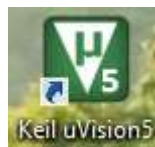
Figure 10: uVision5 Icon

1. When uVision has launched and if a project already exists, then first close the project by selecting Project >> Close Project.

2. Now from the top bar select Pack Installer option in the top bar as shown in Figure 11.
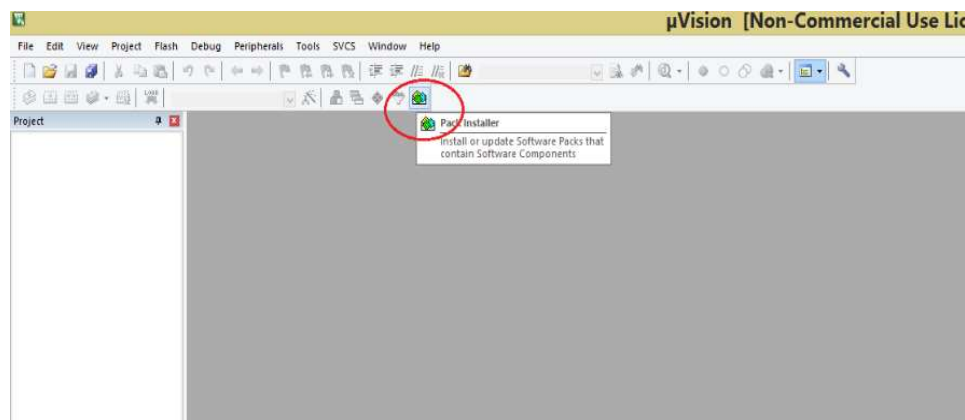
Figure 11: Select Pack installer

3. The Pack Installer Opens up Select the Blinky ULp Project. As shown in Figure 12.

4. Create New Directory for COE718 Labs and a subfolder for Lab1. Copy to that folder as shown in Figure 13.

5. Your workspace should now resemble Figure 14.
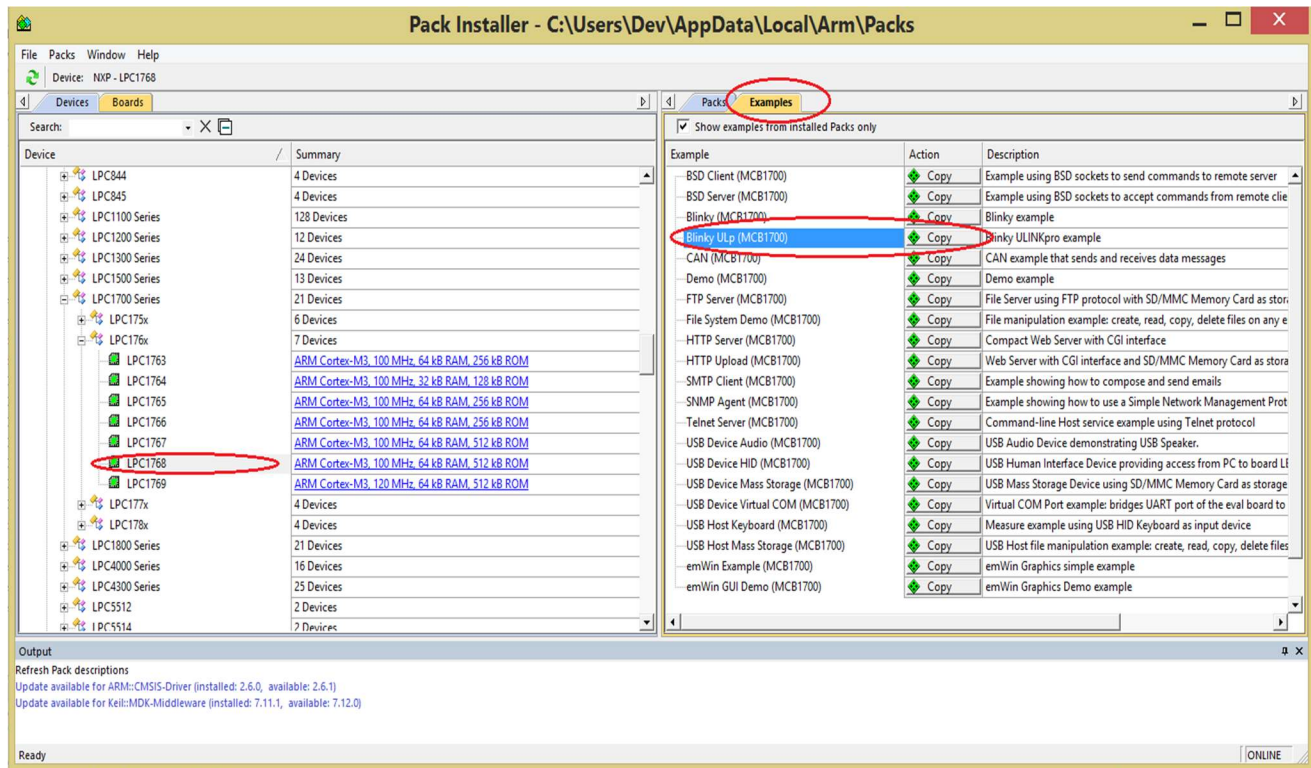
Figure 12: Copy Blinky Project
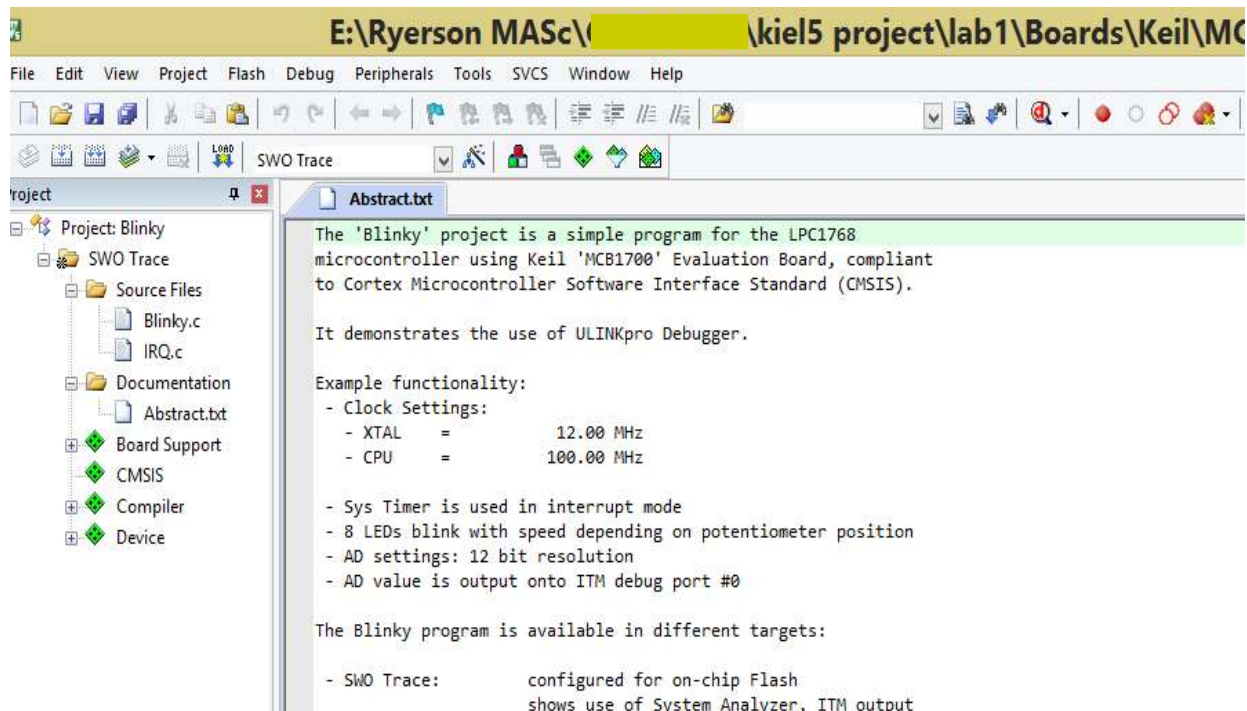
Figure 13: Copy Blinky to Lab1 folder
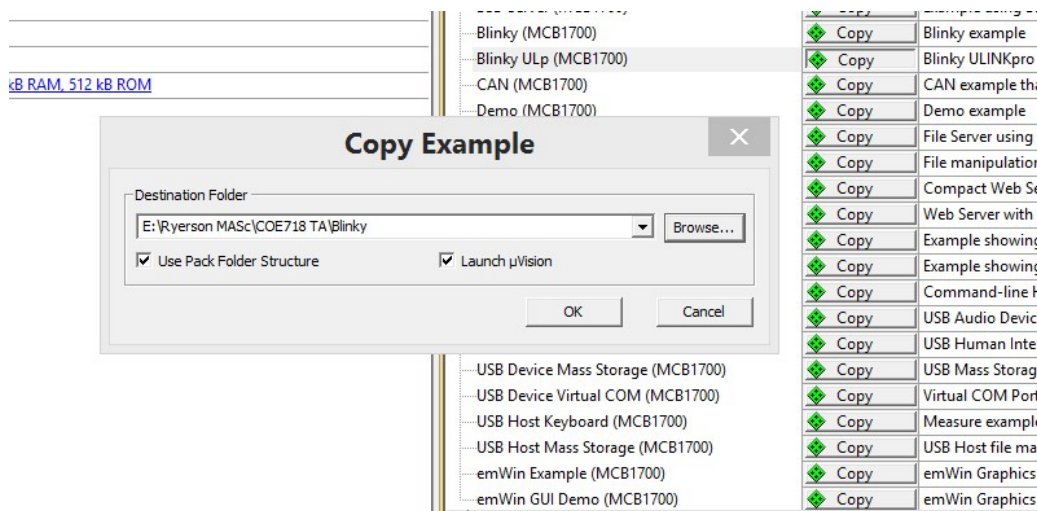
Figure 14: Workspace

6. Double click on *startup_LPC17xx.s* to open the editor. Click on the "**Configuration Wizard**" tab at the bottom of the editor window as shown in Figure 15. 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.
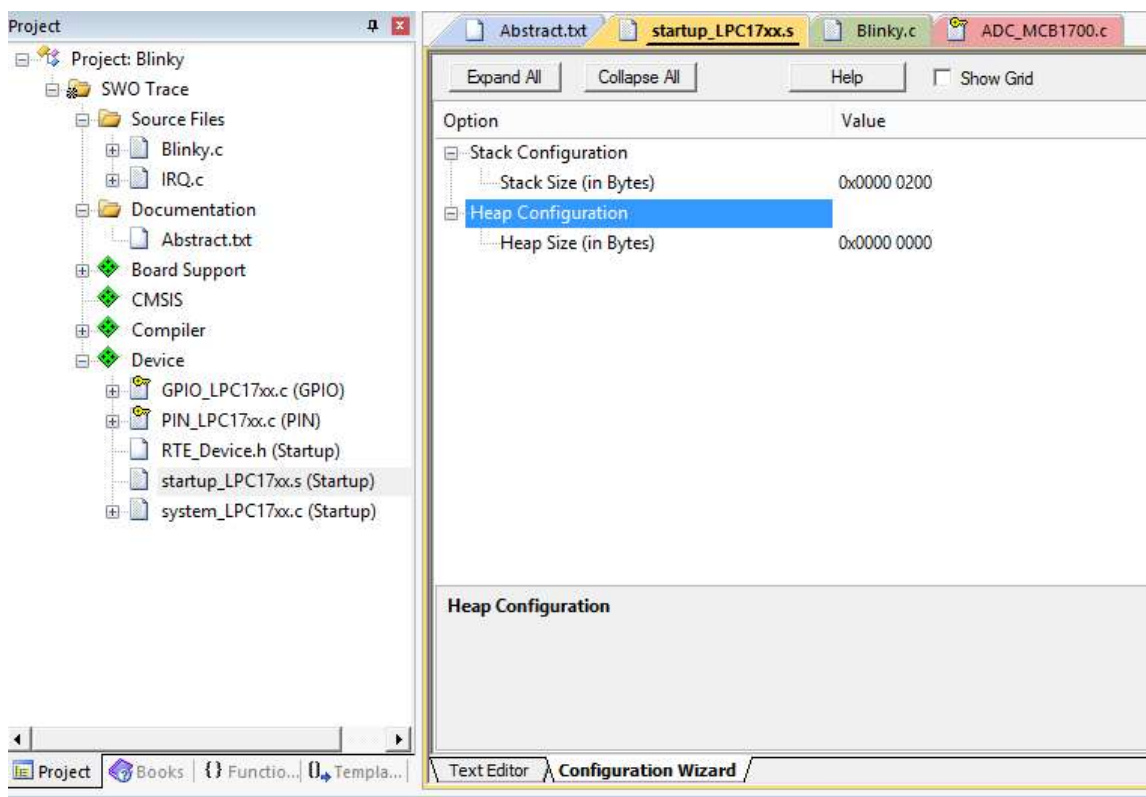
Figure 15: Project Files

7. 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.

8. During this lab, we will be <u>simulating</u> 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 'SWO Trace'. Click the tab entitled "C/C++".

9. In the box "Preprocessor Symbols", write "ADC_IRQ" in the textbox *Define*. Click OK.

    Enabling printf: Project >> Options for Target 'SWO Trace. Select the Debug tab (see Figure 16), 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.

10. To compile and link the .c modules, click the build icon ![build icon]. You can alternatively build the project by pressing F7. Make sure that the project compiles and links without any errors or warnings. A *newline at end of file* warning may appear; this is fine.
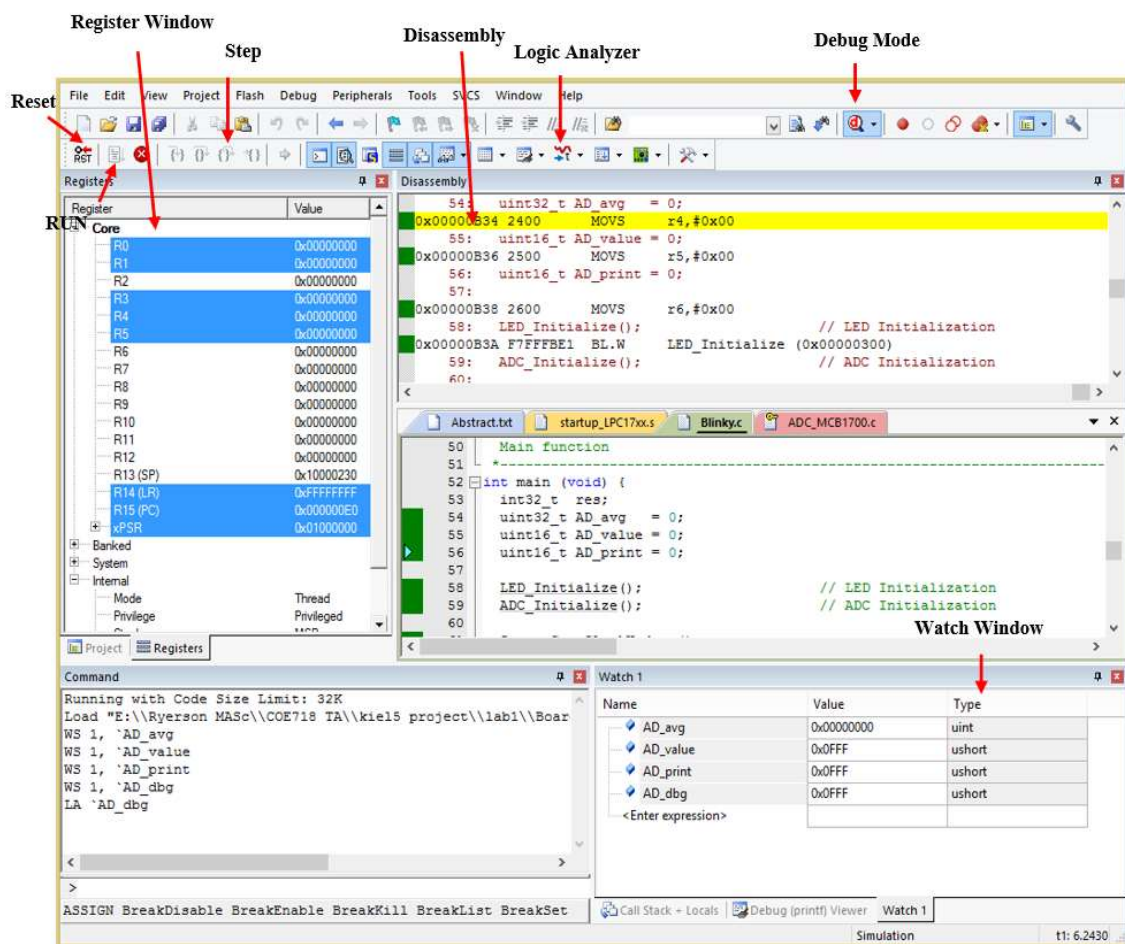
Figure 16: uVision Debug Window

Side Note: Examining the Application

Before we continue to work with the 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, IRQ.c. How do they work together? What are their functionalities?

4.2. 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

> **Blinky.c** - main file, initializes the LED, Serial and ADC functions
> **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.

Enter the Debug mode by clicking on the ![icon] icon. A window will pop up displaying: "EVALUATION MODE Running with Code Size Limit:32K", Click OK and uVision will transform into new successive multiple 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 Figure 16 window. The Debug mode will connect uVision to a simulation model of your program, downloading the project's image into the microcontroller's simulated memory.

1. Reset the program using the ![RST] icon.

2. Execute the program by clicking the ![RUN] RUN icon. STOP (or pause) the program by selecting the ![stop] icon.

Congratulations, you've executed your first program Cortex M3 program through uVision. Now, you know what all these windows in Debug mode actually do and what does this all mean?

4.3. 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 watch the window with the use of steps and/or breakpoints in your code for debugging. A note on steps and breakpoints is given below.
- Steps - (See Figure 16) 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 are 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 the 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 entered on analog channel 2 (AIN2). (More on entering 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 the value of "ADC_dbg" 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 execution.

b) Register Window

The register window (see Figure 16) 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.

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 (In Debug mode).

1. 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.
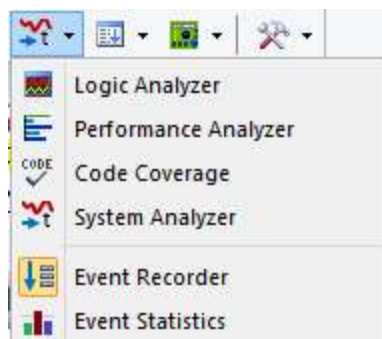
2. Expand some of the tasks in the PA window by pressing the "+" sign located next to the heading. There should be a list of functions present under this heading tree.

3. Press ⬚ Reset icon to reset the program (ensure that the program has been stopped). Click RUN.

4. Watch the program execution and how the functions are called. You will see something similar to that of Figure 17. The analyzer is able to gather various statistics dynamically from the program, useful for both debugging and performance assessment. Stop the program when you have finished analyzing with the PA tool.

Figure 17: Performance Analyzer Window

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.
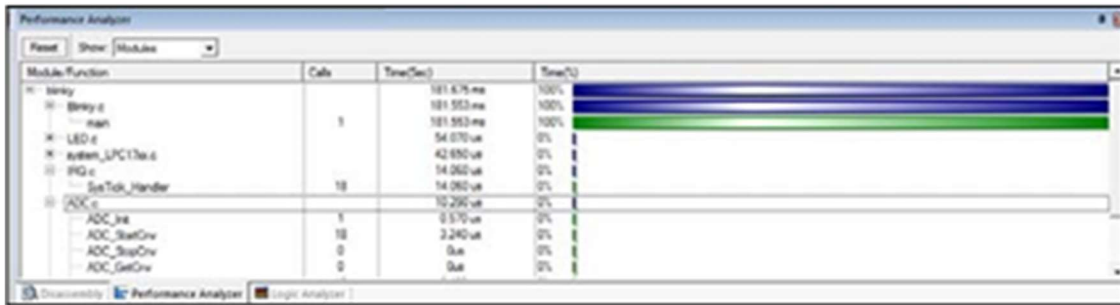2. Regardless of the option , if you hover the mouse over a number in the left column, all the information will be displayed as if you chose both options (i.e., execution time and the number of calls).

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 [icon] 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 GPIO 2 analyzer (LED simulator), select Peripherals >> GPIO Fast Interface >> Port 2. A window will appear as shown in Figure 18. 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).
2. To open the System Tick Timer window, in the main menu select Peripherals >> Core Peripherals >> System Tick Timer. A window will appear resembling Figure 19.
3. To open the AD Converter window, in the main menu select Peripherals >> A/D Converter. A window will appear similar to that of Figure 20.
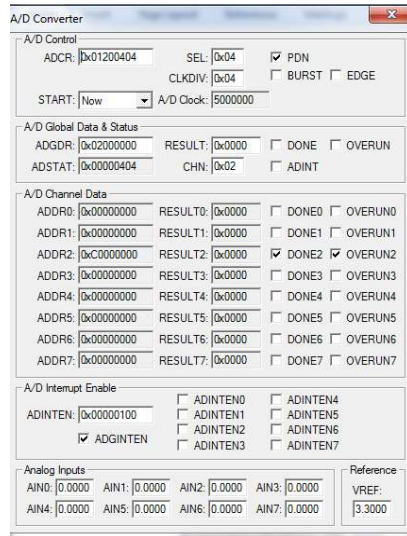
Figure 18: GPIO Peripheral Window

Figure 20:A/D converter Window
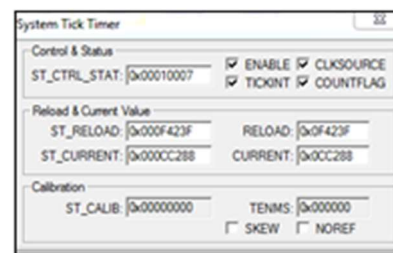
Figure 19: System Tick Timer Window

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 one 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 all the fields of the window.
7. Once, the one 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 speed up 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 has also changed (will transition at a slower pace).

11. Keep transitioning to different values using this simulated potentiometer method. Your simulation should then resemble and close to Figure 21.
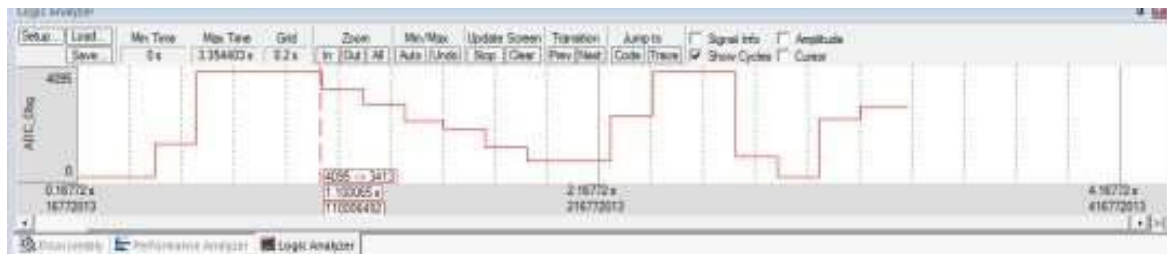

Figure 21: Simulating the Port and A/D Conversion using Logic Analyzer

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.

4. Optional Tutorial Assignment

With the code used in this tutorial, and the joystick files and peripheral notes found in the Appendix of this lab, edit the **Blinky.c** 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 following.

- Demonstrate how will you add joystick to the project from the Manage Run-time Environment.
- Printf- the direction of joystick. (use-- joystick_initialization( ) joystick_stats( ) and printf) **
  **Due to remote learning we cannot physically check it but printing the initial position of the joystick would be enough for the demo.

Hint: Joystick files can be added by clicking button and choosing the Board Support>Joystick (API). Once the file is added explore the joystick_MCB1700.c and call its functions in **Blinky** to perform the assignment task.

Create a pdf file of 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. Add relevant screen shots where required. submit the pdf file through D2L assignment submission system.

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 is 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.

https://www.keil.com/support/man/docs/mcb1700/mcb1700_to_joystick.htm

The 5-position joystick control on the MCB1700 board grounds one of 5 possible port pins depending on how the joystick control handle is positioned. The control may be positioned left, right, up, down or pushed toward the board (select).

- The Left joystick position connects pin C to port pin P1.26 of the LPC17xx device to ground.
- The Right joystick position connects pin B to port pin P1.24 of the LPC17xx device to ground.
- The Up joystick position connects pin A to port pin P1.23 of the LPC17xx device to ground.
- The Down joystick position connects pin D to port pin P1.25 of the LPC17xx device to ground.
- The Select joystick position connects the Cntr pin to port pin P1.20 of the LPC17xx device to ground.

To use the peripherals provided to you on the dev board, ensure that you abide by the following steps. Let us take joystick.MCB_1700.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 joystick, Inititialize() 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 joystick-MCB1700.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));
```

```
    *---------------------------------------------------------------------
* Copyright (c) 2013 - 2019 Arm Limited (or its affiliates). All
* rights reserved.
*
* SPDX-License-Identifier: BSD-3-Clause
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions are met:
*   1.Redistributions of source code must retain the above copyright
*     notice, this list of conditions and the following disclaimer.
*   2.Redistributions in binary form must reproduce the above copyright
*     notice, this list of conditions and the following disclaimer in the
*     documentation and/or other materials provided with the distribution.
*   3.Neither the name of Arm nor the names of its contributors may be used
*     to endorse or promote products derived from this software without
*     specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDERS AND CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
    *---------------------------------------------------------------------
     * Name:    Joystick_MCB1700.c
     * Purpose: Joystick interface for MCB1700 evaluation board
     * Rev.:    1.01
     ---------------------------------------------------------------------*/

    #include "LPC17xx.h"

    #include "PIN_LPC17xx.h"
    #include "GPIO_LPC17xx.h"

    #include "Board_Joystick.h"

    #define JOYSTICK_COUNT               (5U)

    /* Joystick pins:
       - center: P1_20 = GPIO1[20]
       - up:     P1_23 = GPIO1[23]
       - down:   P1_25 = GPIO1[25]
       - left:   P1_26 = GPIO1[26]
       - right:  P1_24 = GPIO1[24] */


    /* Joystick pin definitions */
```

```
static const PIN JOYSTICK_PIN[] = {
  { 1U, 20U},
  { 1U, 23U},
  { 1U, 25U},
  { 1U, 26U},
  { 1U, 24U}
};


/**
  \fn          int32_t Joystick_Initialize (void)
  \brief       Initialize joystick
  \returns
   - \b  0: function succeeded
   - \b -1: function failed
*/
int32_t Joystick_Initialize (void) {
  uint32_t n;

  /* Enable GPIO clock */
  GPIO_PortClock    (1U);

  /* Configure pins */
  for (n = 0U; n < JOYSTICK_COUNT; n++) {
    PIN_Configure (JOYSTICK_PIN[n].Portnum, JOYSTICK_PIN[n].Pinnum, PIN_FUNC_0, 0U, 0U);
    GPIO_SetDir   (JOYSTICK_PIN[n].Portnum, JOYSTICK_PIN[n].Pinnum, GPIO_DIR_INPUT);
  }
  return 0;
}

/**
  \fn          int32_t Joystick_Uninitialize (void)
  \brief       De-initialize joystick
  \returns
   - \b  0: function succeeded
   - \b -1: function failed
*/
int32_t Joystick_Uninitialize (void) {
  uint32_t n;

  /* Unconfigure pins */
  for (n = 0U; n < JOYSTICK_COUNT; n++) {
    PIN_Configure (JOYSTICK_PIN[n].Portnum, JOYSTICK_PIN[n].Pinnum, 0U, 0U, 0U);
  }
  return 0;
}

/**
  \fn          uint32_t Joystick_GetState (void)
  \brief       Get joystick state
  \returns     Joystick state
*/
uint32_t Joystick_GetState (void) {
  uint32_t val;

  val  = 0U;
  if   (!(GPIO_PinRead  (JOYSTICK_PIN[0].Portnum,  JOYSTICK_PIN[0].Pinnum)))  val  |=
JOYSTICK_CENTER;
  if   (!(GPIO_PinRead  (JOYSTICK_PIN[1].Portnum,  JOYSTICK_PIN[1].Pinnum)))  val  |=
JOYSTICK_UP;
  if   (!(GPIO_PinRead  (JOYSTICK_PIN[2].Portnum,  JOYSTICK_PIN[2].Pinnum)))  val  |=
JOYSTICK_DOWN;
  if   (!(GPIO_PinRead  (JOYSTICK_PIN[3].Portnum,  JOYSTICK_PIN[3].Pinnum)))  val  |=
JOYSTICK_LEFT;
```

```
    if  (!(GPIO_PinRead  (JOYSTICK_PIN[4].Portnum,  JOYSTICK_PIN[4].Pinnum)))  val  |=
JOYSTICK_RIGHT;

    return val;
  }



/*-----------------------------------------------------------------------------
 * Copyright (c) 2013 - 2019 Arm Limited (or its affiliates). All
 * rights reserved.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *   1.Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   2.Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *   3.Neither the name of Arm nor the names of its contributors may be used
 *     to endorse or promote products derived from this software without
 *     specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDERS AND CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *-----------------------------------------------------------------------------
 * Name:    Board_Joystick.h
 * Purpose: Joystick interface header file
 * Rev.:    1.0.0
 *-----------------------------------------------------------------------------*/

#ifndef __BOARD_JOYSTICK_H
#define __BOARD_JOYSTICK_H

#include <stdint.h>

#define JOYSTICK_LEFT                   (1 << 0)  /// Defines the Left-button
```

```c
#define JOYSTICK_RIGHT                    (1 << 1)  /// Defines the Right-button
#define JOYSTICK_CENTER                   (1 << 2)  /// Defines the Center-button
#define JOYSTICK_UP                       (1 << 3)  /// Defines the Up-button
#define JOYSTICK_DOWN                     (1 << 4)  /// Defines the Down-button


/**
  \fn           int32_t Joystick_Initialize (void)
  \brief        Initialize joystick
  \returns
   - \b  0: function succeeded
   - \b -1: function failed
*/
/**
  \fn           int32_t Joystick_Uninitialize (void)
  \brief        De-initialize joystick
  \returns
   - \b  0: function succeeded
   - \b -1: function failed
*/
/**
  \fn           uint32_t Joystick_GetState (void)
  \brief        Get joystick state
  \returns      Joystick state
*/

extern int32_t  Joystick_Initialize     (void);
extern int32_t  Joystick_Uninitialize   (void);
extern uint32_t Joystick_GetState       (void);


#endif /* __BOARD_JOYSTICK_H */
```