CAB202 Tutorial 7

Teensy Introduction and Basic IO

Overview

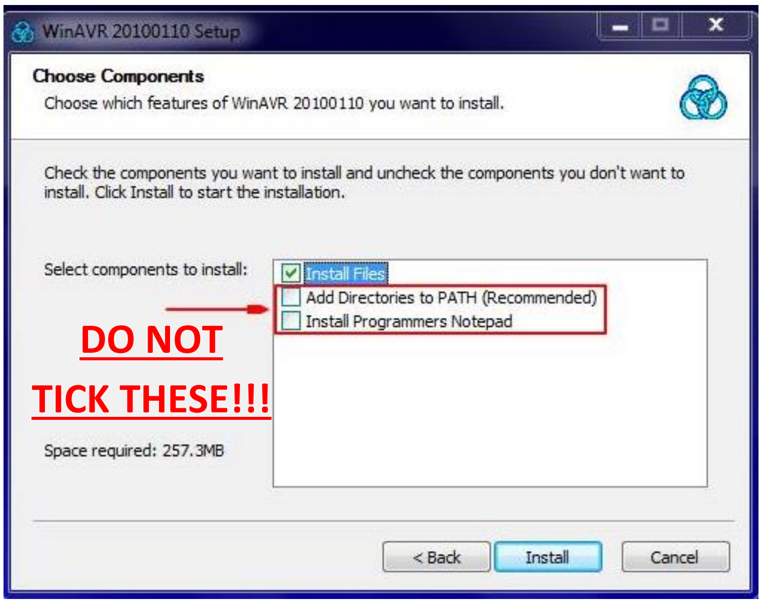
The first half of this unit has used the C programming language to perform complex software-based operations on a computer. The next step is using the capabilities of the C language to perform low level operations (turning on LEDs, responding to button presses, controlling an LCD screen, etc.) on a real microcontroller (the Teensy). This tutorial will work through the setup of the environment required for the microcontroller and will be needed for the remainder of CAB202. The tutorial will also introduce the basics behind configuring and performing digital input and output (IO) on a real microcontroller with C code. This tutorial is worth 3% of your final mark for this subject.

Configuring a development environment for the Teensy

The general development environment we will be using is mostly the same as what has been used throughout the unit so far. You can use the same text editing program you have used in previous tutorials to edit your files. However to compile programs for an AVR, or any microcontroller you need a C compiler with a slightly different configuration. The main noticeable difference, is that rather than compiling source code into a standard executable type, the Teensy loader requires a **`*.hex`** type executable file. The compiler used to generate this executable is called **`avr-gcc`** and it is used to create executables that can be run on the AVR microcontroller architecture. You can run the compiler in a terminal window, including Cygwin, Console2, and Command Prompt.

The installation files and setup instructions can be found here for (the installer is also available in the **cab202_software.zip** on Blackboard, and some QUT computers already have WinAVR installed):

- Windows Systems: Installation Files and Setup Instructions
- Mac / Unix Systems: Setup Instructions (no external download of installation files is required, all done through the terminal)


WinAVR 20100110 Setup

Choose Components

Choose which features of WinAVR 20100110 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Install to start the installation.

Select components to install:

[✓] Install Files
[ ] Add Directories to PATH (Recommended)
[ ] Install Programmers Notepad

DO NOT TICK THESE!!!

Space required: 257.3MB

When installing on Windows, make sure to ONLY tick the box shown in the above image. There is a reported bug in WinAVR that will WIPE your PATH variable rather than appending to it (there are severe consequences from the loss of your PATH variable) if you tick the second box. After installing, you will have to manually add "`<winavr-dir>\bin`" to your PATH variable.

To test that **`avr-gcc`** is installed and setup correctly on your computer, or is already installed on a university computer, type in the following command into your chosen terminal window:

```
avr-gcc --version
```

If everything was installed and setup properly you should see an output similar to this (the version number will more than likely be different which is fine):

```
avr-gcc (WinAVR 20100110) 4.3.3
Copyright   (C)   2008   Free   Software   Foundation,   Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.
```

Compiling your first program

Once you have installed **`avr-gcc`** for your system, you can go through the following process in a terminal window. Make sure the terminal is in the same directory as your source code, just like previously when compiling C code with normal GCC. The compilation for the Teensy microcontroller is done with the following **`avr-gcc`** commands:

```
avr-gcc -mmcu=atmega32u4 -Os -DF_CPU=8000000UL <C source
files> -o <applicationName>.o
```

Once that command has finished, use the following command to complete compilation and produce a **`*.hex`** file:

```
avr-objcopy -O ihex <applicationName>.o <applicationName>.hex
```

So for a **`helloworld`** program you would use the following two lines in a terminal to create a **`*.hex`** file ready for upload to the Teensy with the Teensy uploader application:

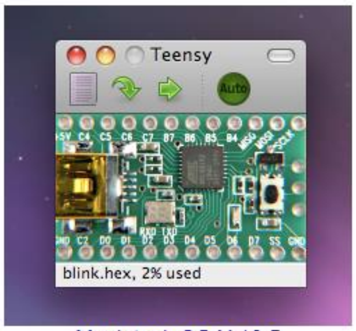
```
avr-gcc -mmcu=atmega32u4 -Os -DF_CPU=8000000UL helloWorld.c
-o helloWorld.o
avr-objcopy -O ihex helloWorld.o helloWorld.hex
```

Please take note that the **-DF_CPU** flag is compiling the program with a CPU speed of 8MHz. In your code you must also request a CPU speed of 8MHz. This is done by setting the CLKPCE bit of the Clock Prescaler Register (CLKPR), and then providing a prescaler selection in bits 3-0. See the **`cpu_speeds.h`** header for a series of macros that perform this (and see the datasheet on Blackboard for the port and pin configurations). In short, always start your `main()` function with:

```
set_clock_speed(CPU_8MHz);
```

Loading your first program

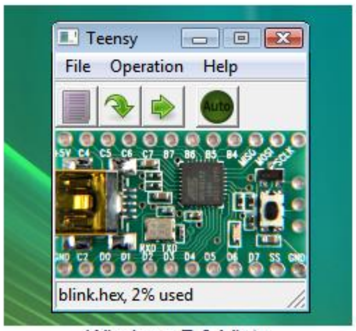
You will also need to download a standalone executable, developed by the same people who created the Teensy board, called **Teensy Loader** (download here). This executable will take the **`*.hex`** file compiled with the compiler and upload them to the Teensy. Explicit instructions on how to use this application can also be found on the site, however, all you need to know is that there are 3 buttons: 1) select file, 2) send file to Teensy, and 3) reboot Teensy.



Macintosh OS X 10.5

Linux (Ubuntu)

Windows XP



Windows 7 & Vista

Configuring and using digital IO registers

There are 3 registers which affect the way in which the microcontroller is setup for digital input and output. These registers are called the data direction register (**DDRx**), the port register (**`PORTx`**) and the pin register (**`PINx`**). The ensuing section discusses how each of these 3 registers are used (http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=37871 provides a great explanation of these as well).

DDRX

The data direction register is used to configure a pin in an AVR microcontroller as an output or an input. Setting a bit to a `1` will make the respective pin an output, while setting it to a `0` will make it an input. The following example would set the 0th pin in **PORTB** (also referred to with **`PB0`**) as an output, while the rest of the pins on the port are set as inputs:

```
DDRB = 0b00000001;
```

This line of code can also be performed using bit shifting. Bit shifting is a lot easier to debug and understand as it explicitly states what bit is being set:

```
DDRB = (1<<0);
```

To set a pin as an input we need to set the respective bit to a 0. This can be achieved through complementing (NOT operator) the bit shifting operation. The following line would bit shift 1 three places, then take the bitwise NOT of the value. The result is pin 3 of PORTB set as an input:

```
DDRB = ~(1<<3);
```

However, it should be noted that if the following lines of code were run sequentially, the second line would override what was done in the first line, and in fact pin 2 of PORTB (PB2) would be configured as an input. The reason for this is simple, we first write the binary value of `0b11111011` to the DDRB register, but then we write the value of `0b11110111`, and so we are overriding that previously written zero in the second bit.

```
DDRB = ~(1<<2);
DDRB = ~(1<<3);
```

There is a common solution to this however; we can use the OR and AND bitwise operators. If we perform a bitwise AND or a bitwise OR on the register with whatever we are trying to setup as an input or output, we will only change those bits we indeed wanted as an input or output (while the rest of the register stays the same). For example to configure pin 3 of PORTB as an input, and not affect the rest of the bits, we would have the following code:

```
DDRB = DDRB & ~(1<<3);
```

The DDRB register is in some unknown state, and all we know is that we don't want to change it. So we shall write it as `0bxxxxxxxx`, where `x` is an unknown bit state. Also performing the bit shifting and complement (NOT) of the second part of the line we would get `0b11110111`. Thus we would have:

```
DDRB = 0bxxxxxxxx & 0b11110111
```

Now let us perform the bitwise AND operation. We have 2 cases, we either have the bitwise AND of `x` and `1`, or the case of `x` and `0`. In each of these cases there are, again, two cases for the value of x. However, for example in case 1, AND of x and `1`, we could have x = `1` or x = `0`. Performing the operation for both of these cases would get a result of `1` for the case x = `1`, and a result of `0` for the case x = `0`. Thus the result is simply the value of x. The same can be done for the second case when the AND of x and `0` is found. These are the results of the 4 possible cases:

	AND of X and 1	AND of X and 0
X = 1	1 (Value of X, so stays in same state as previously)	0 (Bit equals 0 as desired, does not depend on original value of X)
X = 0	0 (Value of X, so stays in same state as previously)	0 (Bit equals 0 as desired, does not depend on original value of X)

So this bitwise AND technique can be used when we wish to set a register to a value of `0`, and do not want to change the any of the other bits. The same sort of process can be applied to bitwise OR, which is used when we want to set a bit in a register to a value of `1` and do not want to change any of the other values. The following table summarise the bitwise OR cases:

	OR of X and 1	OR of X and 0
X = 1	1 (Bit equals 1 as desired, does not depend on original value of X)	1 (Value of X, so stays in same state as previously)
X = 0	1 (Bit equals 1 as desired, does not depend on original value of X)	0 (Value of X, so stays in same state as previously)

It should also be mentioned that these lines of code can be written in a compound form by using the compound assignment operators (http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B):

```
DDRB = DDRB & ~(1<<3); // Long version
DDRB &= ~(1<<3);  // Equivalent using &=
```

PORTX

The PORT register is used to set pins that are configured as outputs to either high / on / 1 or low / off / 0. This is done by configuring the bit you wish to be high or low as 1 or 0 respectively, assuming that the bit you wish to set high or low was already configured by the corresponding DDR register. For example, the following line of code would configure pin 7 on **PORTA** (PA7) to output 1, assuming previously set up as an output using the DDRA register.

```
PORTA |= (1<<7); // set PA7 high using shorthand bitwise OR
```

PINX

The PIN register is used to read the states of a pin, again assuming that the pins you wish to read was setup previously as an input. For example, the following code would read all the values on PINC:

```
char readValue = PINC;
```

However this would read all 8 bits from PINC into the variable readValue. In most cases, you are only interested in reading a single bit of this register (i.e. finding whether it is logical low or high). This becomes a lot clearer with a solid example. For example, consider a scenario where pin 3 on port C is connected to a button and when the button is pressed the pin value goes high (i.e. to 1). Obviously, we are only interested in the value of pin 3 (there could be other buttons on other pins that obviously shouldn't effect checking the state of the button on pin 3). To only check the value of a single bit in PINC, bit operators are used. The following code checks if bit 3 of PINC is logical high and stores the result in switchValue:

```
char switchValue = (PINC>>3) & 1;
```

Assume that the pin on bit 3 is some unknown value (denoted by S) and the values of all the other bits on the PINC register are unknown (denoted by x). So that means PINC defined as:

```
PINC = 0bxxxxSxxx
```

Now, the bit shifting operation is performed (PINC>>3). This means that the value of PINC is shifted 3 places to the right. The result of this operation is:

```
(PINC>>3) = 0b000xxxxS
```

Note that the bit of interest (S) is now in the least significant bit (LSB) position. The AND bitwise operation is then executed with this bit shifted value and the value 1. Therefore, the result is as follows:

```
(PINC>>3) & 1 = 0b000xxxxS & 0b00000001 = 0b0000000S
```

Consequently, the original line of code will store the above value is in the `switchValue` variable. Therefore:

```
switchValue = 0b0000000S
```

So if S was equal to `1`, then the result would be `1`, and if S was `0` the result would be `0`. Importantly, as was desired at the start of this explanation, `switchValue` would be equal to `1` if the switch was logical high, or `0` if it was logical low.

Assessed Exercises

NOTE: All questions are to be marked in your allocated tutorial session only!

1. Write your first Teensy program (0.5 marks)

Write a program for your Teensy that blinks one of the LEDs at 1Hz (i.e. out of every second the light should be on for 500ms and off for 500ms).

2. Bit operations mayhem (1 mark)

Download the template file called **`question_2.c`** from Blackboard. For this exercise, you've been given a template file with 10 bitwise operations to complete. This program must be compiled and run **on your computer**. Your job is to complete the corresponding **`answer_*()`** functions so that you pass all of the problems. You do not need to do anything other than complete correct implementations for the numbered functions ranging from **`answer_1()`** to **`answer_10()`**. Show your tutor the output of your completed implementation. If you have 5 or more correct, you will receive half a mark, and if you have all 10 correct you will receive a full mark.

The testing driver in the `main()` function performs tests on your implementation for each function, and provides a lot of feedback to the terminal via `printf()`. While processing this feedback, you may find it easier to view it in your preferred text editor rather than the terminal. You can achieve this by redirecting the output of this program to a file. For example, to store the output of `question_2` in a file called "**`results.txt`**" (in your current directory):

```
./question_2 > results.txt
```

3. Write a program using both input and output (0.5 marks)

Write a program for your Teensy that uses the right button (SW3) to turn on the left LED (LED0). When the button is pressed down, the light should go on, and when not pressed the light should be off.

4. Write a program that makes use of multiple buttons (1 mark)

Write a program for your Teensy that controls the state of the LEDs with the 5-way push button. Each of the 5 possible buttons should do the following when pressed:

- The left button turns the left LED on, and right LED off
- The right button turns the right LED on, and left LED off
- The up button turns both LEDs on
- The down button turns both LEDs off
- The centre press toggles the current state of each LED. After the LEDs are toggled, there should be a delay of 500ms (purely so that the toggle is visible when held down).

Note: demonstrating this exercise to your tutor will award you the marks for questions 1 and 3