SOEN 422

Tutorial 2

Outline

1. Installing AVR-C Development Environment
2. Basics of AVR-C Programming

Installing AVR-C Development Environment

- The lab computers should already have the AVR-C development environment installed, but if you wish to work on your own laptop the installation procedure is outlined here.
- Download and install the AVR-GCC toolchain for your operating system.
    - **Windows**: WinAVR
    - **OS X**: CrossPack for AVR Development
    - **Ubuntu Linux**:

```
sudo apt-get install gcc-avr binutils-avr gdb-avr avr-libc make -y
```

- AVR-C programs can be loaded using the Teensy Loader through the Arduino IDE, however a command line tool exists to load programs without using the Arduino IDE if desired.

Introduction to AVR-C

- To write programs in AVR-C, you must manipulate the microcontroller's registers to obtain the behavior which you desire.
- AVR-C programs are written in C and make use of libraries which are specific to AVR microcontrollers allowing access to the microcontroller's registers.
- To access these registers, you need to include `io.h`

```c
#include <avr/io.h>
```

Compiling and Loading AVR-C Programs

- Compile the C code and any libraries into an object file:

```
avr-gcc -mmcu=at90usb1286 -O -o {output.o} {input.c} {someLibrary.c}
```

- Generate a HEX file from the object file.

```
avr-objcopy -O ihex {output.o} {output.hex}
```

- Load the HEX file from the Teensy Loader.
    - Open the HEX file from the file menu of the Teensy Loader and press the reset button on your Teensy to load the HEX file.
- A makefile is available on Github to automate the procedure of compilation and hex file generation.

Teensy++ 2.0 Ports

- The Teensy++ 2.0 has 6 ports, PORTA, PORTB, PORTC, PORTD, PORTE, PORTF.
    - Each port has its own Data Direction Register and Data Register.
    - Data Direction Registers are accessed using **DDRx** where x is the letter of the port. (ex. DDRD for Port D's Data Direction Register)
    - Data Registers are accessed differently when reading and writing:
        - Data Registers are written to using **PORTx** where x is the letter of the port. (ex. PORTD for writing to Port D's Data Register).
        - Data Registers are read from using **PINx** where x is the letter of the port. (ex. PIND for reading from Port D's Data Register).

Bitwise Operations

- It is common to use bitwise operations to manipulate registers.
- Register values are commonly written using `|=`, `&=` and `^=` where the value stored is the result of an OR, AND, or XOR operation between the register's current value and the value on the right hand side of the operator.
- Assigned values are commonly determined using the left shift operator to shift a 1 into the desired register location.
    - ex: (1 << PD7) means shift a 1 into the last bit of Port D's data register.
    - Register bits are addressed from 0.
    - `PD7 == 7`
        - `PD7` is simply a helper constant to represent Pin 7 of Port D.
- Helpful Operator: ~ is the negation operator in C.

CPU Speed

- Some functions depend on variables specifying the CPU speed.
- The microcontroller on the Teensy++ 2.0 runs at 16Mhz. To indicate this, include the following define statements at the top of your program:

```
#define CPU_PRESCALE(n) (CLKPR = 0x80, CLKPR = (n))
#define CPU_16MHz 0x00
#define F_CPU 16000000L
```

- At the beginning of the `main()` function, set the CPU prescaler value.

```
CPU_PRESCALE(CPU_16MHz);
```

- More information on this topic is available on the Teensy++ 2.0 Website and in the datasheet under section 7.9.1

Digital Output

- To output a digital signal on a pin, that pin's data direction must first be set to output (1) in the correct register location using its corresponding data direction register.
- Setting mode to output (use OR operation so that 1 will be stored regardless of the original value):

```
DDRD |= (1 << PD6);
```

- To set a pin to `HIGH` or `LOW`, store a 1 for high and 0 for low in the data register using **PORTx**. In the following example, **Set Port D, Pin 6 (PD6) to High**

```
// Set PD6 to output mode.
DDRD |= (1 << PD6);

// Write HIGH to PD6.
PORTD |= (1 << PD6);
```

Digital Input

- To read a digital signal on a pin, that pin's data direction must first be set to input (0) in the correct register location using its corresponding data direction register.
- Setting mode to input (use AND operation with negation to ensure any previously stored 1 is erased and no other bits are affected):

```
DDRD &= ~(1 << PD6)
```

- To enable the internal pull-up resistor, set the corresponding PORT bit to 1.

```
PORTD |= (1 << PD6);
```

Digital Input

- The state of a pin is read using **PINx**:

```
//Set PD6 to INPUT with internal pull up resistor enabled.
DDRD &= ~(1 << PD6)
PORTD |= (1 << PD6);

//Read Port D Data Register.
int value = PIND;

if (value & (1 << PD6)) {
  // Do something when PD6 is HIGH.
}
```

Harware Timer Use Cases

- The hardware timers can be used for a variety of use cases.
    - Analog Signals (PWM, Fast PWM)
    - Toggling a Signal (CTC)
    - Interrupts
- Sections 14, 15 and 16 of the datasheet are very important for these use cases.

Analog Output

- To use PWM in AVR-C, a number of registers must be configured.
- The relevant documentation is in sections 14, 15 and 16 of the datasheet.
- NOTE: Not all pins support PWM, you must verify on your pinout charts to find those which do.
    - PWM is available on output compare pins labeled **OCxx**.

Analog Output - 8 bit Timer

- Using an 8 bit timer as an example (Timer 0):
  - 2 Output Compare Pins: **OC0A** and **OC0B**
    - These are the pins that are manipulated.
  - 2 Timer/Counter Control Registers for controlling how the output compare pins behave:
    - TCCR0A and TCCR0B
  - TCCR0A
    - 6 bits are used:
    - 2 for OC0A behavior
    - 2 for OC0B behavior
    - 2 for Waveform Generation (Off, PWM, Fast PWM, CTC)
  - TCCR0B
    - 4 bits are used:
    - 1 for Waveform Generation (Off, PWM, Fast PWM, CTC)
    - 3 for Clock Select (Internal with or without prescaling or external)

Analog Output - 8 bit Timer

- Timer/Counter Register: **TCNT0**
  - 8 bit register which is incremented to keep track of time.
- Output Compare Register A: **OCR0A**
  - 8 bit compare register to set the compare value (pulse width/duty cycle).
- Output Compare Register B: **OCR0B**
  - Another 8 bit compare register to set the compare value (pulse width/duty cycle).

Analog Output - 8 bit Timer PWM Example

- The following is an example on how to manipulate the timer registers to get a PWM.

```
// Set OC0A to output.
DDRB |= (1 << PB7);

// Setup the Timer/Counter Control Registers.
// Clear pin on Compare Match, Fast PWM update at top.
TCCR0A |= (1 << COM0A1) | (1 << WGM01) | (1 << WGM00);

// No prescaling.
TCCR0B |= (1 << CS00);

// Set duty cycle to 50%.
OCR0A = 128;
```

Hardware Timers and Interrupts

- To make use of interrupts, you need to use the same timer configuration procedure as with CTC and PWM, but you must enable interrupts using the Timer/Counter Interrupt Mask Register.
    - You can either enable match interrupts which trigger when the output compare register's value matches the Timer/Counter register's value or when the Timer/Counter register overflows.
- You must enable global interrupts in addition to the specific interrupt you are interested in.
    - To enable global interrupts use `sei()` and to disable global interrupts use `cli()`.

```
// Enable Timer/Counter register overflow interrupts on Timer 1
TIMSK0 |= (1 << TOIE0);

// Enable Interrupts (Global Interrupt Mask)
sei();
```

Hardware Timers and Interrupts

- Finally, you must write an interrupt handler.

```
ISR(INTERRUPT_TYPE_vect) {
  // Your interrupt handler code.
}
```

- `INTERRUPT_TYPE_vect` is the type of interrupt you want to catch. A list of interrupt vectors is found on page 68 of the datasheet. The value you are looking for is under the "Source" column.
    - Replace spaces with underscores and append "vect" at the end.
    - ex. TIMER1 OVF == TIMER1_OVF_vect

Analog Input

- To read an analog signal on a pin, the microcontroller's analog/digital converted (ADC) must be used. The relevant documentation is in section 26 of the datasheet.
- Before an analog signal can be read, the ADC registers must be configured.
- First, the ADC Multiplexer Selection Register must be set up with the pin to be read and the reference voltage.
    - Analog signals can be read on pins F0 to F1. To indicate the pin, perform a logical OR with the pin number to be used.
    - In most cases AVcc (REFS0) should suffice for the reference voltage.

```
// Set Voltage reference to AVcc, Set ADC Multiplexer to read pin F2.
ADMUX |= (1 << REFS0) | 0x2;
```

Analog Input

- Next, the ADC must be enabled. To do this, we set the appropriate pin on an ADC Control and Status Register.

```
// Enable ADC
ADCSRA |= (1 << ADEN);
```

- Finally, to read an analog signal, the ADC conversion must be started and you must wait for the ADC interrupt flag to be set. Once it is set, you may read the value from the ADC register and clear the flag by setting a 1 to the appropriate bit in the ADC Control and Status Register.

```
// Start the ADC reading.
ADCSRA |= (1 << ADSC);

if (ADCSRA & (1 << ADIF)) {
  int value = ADC;
} else {
  // The value is not ready yet, keep waiting.
}
```

Serial Communication

- Instead of implementing our own serial communication over USB, we will use a library provided at the Teensy++ 2.0 Website.
- Download the library and unzip it into your project directory.
- Be sure to include this library when compiling your code.
- The API is documented at the Teensy++ 2.0 Website.
- Three key functions:
  - `usb_init()`: Initialize the usb serial library.
  - `serial_write()`: Write serial data through the usb connection.
  - `usb_serial_getchar()`: Reads one byte from the input buffer.

References

- AT90USB1286 Datasheet

Copyrights