edcetera engineering

Designing with Microcontrollers

Mark A. Strain, P.E.

PDH302
3 Hours

Course Material and Final Exam

Introduction

In my course entitled "The Basics of Microcontrollers" I discussed the architecture of microcontrollers. I showed how the central processing unit fetches instructions (or a program) from memory and decomposes the instructions into components that the control unit and the arithmetic logic unit can use to perform the desired operation or function. Here I will discuss how to design a simple circuit incorporating a microcontroller with a small footprint, small pin count, and a small amount of internal memory (both program and data memory). I will give program examples using the C programming language.

Microcontrollers are simply microprocessors that include program and data memory and peripherals such as general-purpose input/output ports, timers, serial communications controllers, analog-to-digital converter, etc.

For this course I will utilize the Atmel AVR series of microcontrollers, specifically the Atmel ATtiny2313A series with 2048 bytes of internal flash program memory and 128 bytes of internal data memory. The Atmel AVR series is one of several different processor options a developer can use. Other example microcontrollers include, but are not limited to the Microchip PIC, Texas Instruments MSP430, Intel 8051, STMicroelectronics STM8, Freescale 68HC11, and multiple versions of the ARM core from many vendors.

Design Considerations

When designing a microcontroller-based system, there are three things that need to be considered: the microcontroller, the compiler, and the device programmer.

Microcontroller

First of all, consider the microcontroller. The processor must be sized appropriately to the desired task. Consider the following parameters:

- bus width (8-bit, 16-bit, 32-bit)
- processor speed
- amount of program and data memory
- amount of input/output pins
- peripherals
- power consumption

The examples in this course perform basic operations, such as controlling an LED, reading a button, and utilizing a timer peripheral and some interrupts. A microcontroller with an 8-bit architecture is sufficient. The 8-bit, 16-bit, and 32-bit architecture nomenclature refers to the width of the bus within the core of the microprocessor. That is, how many bits the core can process at once. An 8-bit machine is sufficient for simple systems, such as the ones exemplified in this course, as well as thermostats, toys that have LEDs and buttons that need to be controlled and read. Microprocessors with larger bus-widths are used for devices that need more computing horsepower such as cell phones, GPS navigation devices, and MP3 players.

The processor speed will determine how fast an instruction can be executed and how much data can be processed during a given slice of time. For example, a device transmitting and receiving data via a USB port will need to have a faster processor speed (and probably larger bus-width) than a device simply controlling a couple of LEDs and responding to a button. Most microprocessors are clocked by an external clock (or oscillator) which determines the speed of the master clock. Some low-power, low-horsepower microcontrollers may be clocked by an internal oscillator circuit that requires no external components like a crystal or capacitors. The microcontroller used in the examples in this course utilize an internal oscillator circuit contained within the chip.

Memory is also a consideration, both program memory and data memory. Program memory is where the program (or set of instructions specific to the task at hand) is stored. Program memory is persistent and is maintained over power cycles. Data memory is used by a program during execution for the stack and to store variables. Both program memory and data memory may be internal to the microcontroller or external to the chip and accessed by an address/data bus. The program memory must be large enough to store the binary (or compiled source code) for the project. The data memory must be large enough to contain the stack and for all of the variables during program execution.

The number of input/output pins depends on the application or the system that the microcontroller will control. Output pins may control an LED, a motor, a relay, or some pins on an external peripheral device (such as a communications controller). Input pins may be used to read an individual button, or a keypad. In some systems, input/output ports may not be required at all.

Peripherals are those subsystems that interface with the microprocessor and pass data to and from the processor and memory. Peripherals may include systems such as communication devices, such as a UART (universal asynchronous receiver transmitter) or a USB (universal serial bus) controller, timers, pulse-width modulators, and analog-to-digital converters.

Power may or may not be a major consideration. If the processor is controlling or monitoring an industrial process, like monitoring and controlling the temperature of a room, then the device will most likely be plugged into the building's power source. In this case a few extra milliwatts is not a major consideration. However, if the processor is going to control a small handheld device, like a garage door opener remote control or a keyless entry remote, then power consumption is a major concern.

Power consumption needs to be considered from two angles: the amount of power consumed at runtime and while the processor is sleeping. Also, different processors have different levels of power saving modes. Most will let the developer turn off unused peripherals. Some modes will actually halt the clock and resume due to an external signal (like a button push). This power saving feature is useful for devices (such as remote controls) that do not need to do anything until a button is pushed; it then can perform its intended operation and then go back to sleep.

Compiler

The second consideration when designing a microcontroller-based system is what development tools are available. Software needs to be written (whether in assembly or in a higher-level language such as the C programming language). This software needs to be compiled and/or assembled into a binary file that can be loaded onto the device.

The compiler, assembler and linker tool chain need to be considered. Some simple projects can be done in assembly. Most assembler tool chains are free. Most C programs if written properly can be very compact, using very little memory. The use of a compiler (such as a C compiler) allows for ease of design and prevents the developer from having to use processor-specific assembly code instructions. This allows more complex programs to be more easily maintained. Even some of the tiniest microcontrollers are supported by some of the available C compiler tool chains.

Some C compilers have a monetary cost and require some sort of licensing, while others are free (or open source). The tool chain used in this course is from the GNU suite of tools, specifically, WinAVR. It is open source and free.

The linker is usually (almost always) bundled with the compiler/assembler. The job of the linker is to link all of the object code together into a single programming file.

Device Programmer

The third consideration when designing a microcontroller-based system is how to program the system. The software that is written is assembled, compiled and linked, creating a single binary file that needs to be written to the device's nonvolatile memory. The development PC containing the file to be programmed transfers this file to the device programmer via one of the ports of the PC: serial, parallel or USB. The device programmer then transfers the file to the microcontroller's program memory. The file is transferred from the device programmer to the microcontroller usually via a serial interface, like a SPI (serial peripheral interface) or JTAG (Joint Technical Architecture Group) interface.

The device programmer used for proving the examples in this course is the USBASP programmer. It is an inexpensive programmer (supported by the AVRDUDE command line interface) that interfaces to the PC via a USB port.

For high volume production runs, device programmers are not the best solution. Once a system is in production, the program memory chips (usually flash memory) are pre-programmed at the factory or by a third-party by a multi-chip programmer. Or, if a microcontroller is utilized with internal flash memory, the microcontroller may be programmed by at the factory or by a third-party.

Microcontroller

The microcontroller used for the exercises in this course is Atmel ATtiny2313A microcontroller. It is a low power, 8-bit, reduced instruction set (or RISC) microcontroller. The one used here comes in a 20-pin DIP (dual in-line) package which makes it easy to insert into a breadboard for experimentation. It has the following features:

- 2048 bytes internal flash (in-system programmable)
- 128 bytes of internal RAM
- internal oscillator
- 18 programmable input/output lines
- an 8-bit timer with separate prescaler and compare mode
- a 16-bit timer with separate prescaler, compare and capture modes

- 4 pulse-width modulation (PWM) channels
- on-chip analog comparator
- serial communications controller
- USART (universal synchronous/asynchronous receiver transmitter)
- low-power idle, standby, and power down modes
- 1.8 - 5.5 volt operation

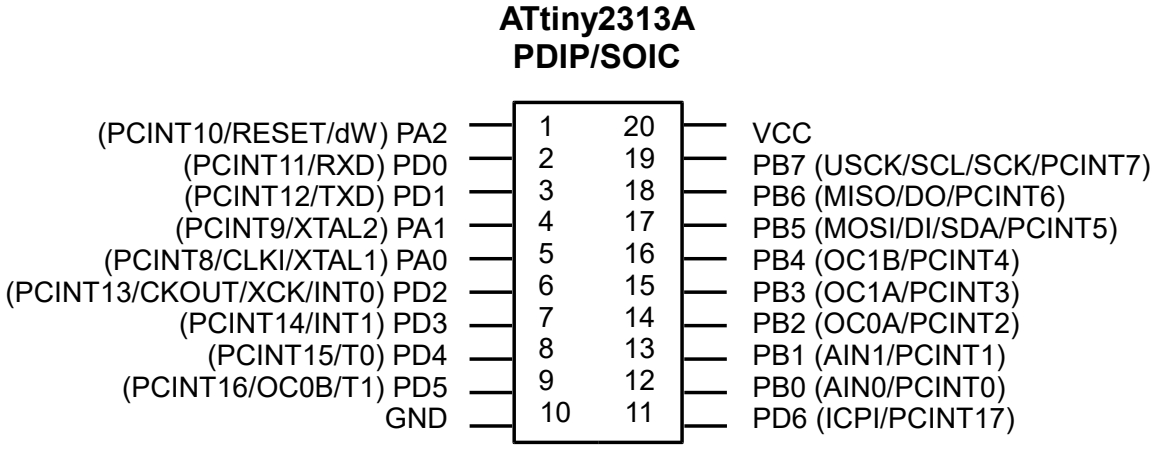

Figure 1 - ATtiny2313A pinout

Another microcontroller in the Atmel AVR series is the ATtiny10. The ATtiny10 device is a powerful microcontroller for its size. It is barely larger than the head of a small nail, but has internal program flash and data memories, and many powerful peripherals, including a 16-bit timer with two PWM channels. Its programming interface is different than the ATtiny2313A processor. It uses a 2-wire serial programming instead of a 3-wire serial interface.

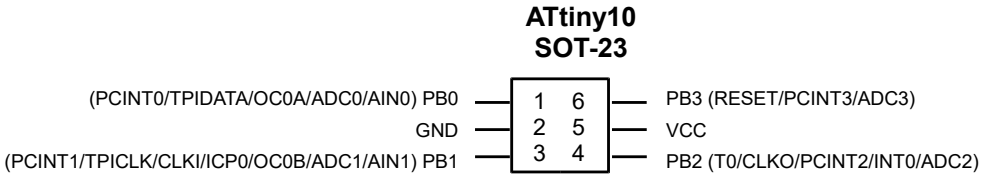

Figure 2 - ATtiny10 pinout

Architecture

The ATtiny2313A is a member of the Atmel AVR series of microcontrollers. It is a reduced instruction set computer (RISC). This means that it has fewer instructions than an x86 processor (for example), but each instruction is powerful, thereby requiring fewer instructions for a

complete instruction set. The RISC processors are more code efficient than older-model processors. Many of the instructions may be executed in a single clock cycle.

The AVR employs a Harvard architecture. Therefore, the core interfaces to program and data memories using separate busses. With a Harvard architecture program and data memories can be accessed simultaneously. While an instruction is being executed, the next instruction is fetched from the program memory. Many microcontrollers employ a Harvard architecture to speed processing (with a lower clock rate) by allowing simultaneous access of program memory and data memory. Since the data and program memory have separate busses to the core, there are no bus collision problems. This feature makes a processor designed with a Harvard architecture faster than a similar processor designed with a von Neumann architecture (where data and program instructions are accessed from the same memory device across the same bus).

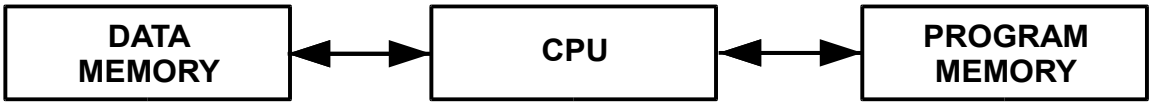

Figure 3 - Harvard Architecture

The AVR core includes 32x8 bit general purpose registers that are directly connected to the arithmetic logic unit (ALU). The main purpose of the central processing unit (CPU), or core, is to maintain correct program execution. The core accesses program and data memories, performs calculations, controls peripherals and handles interrupts.

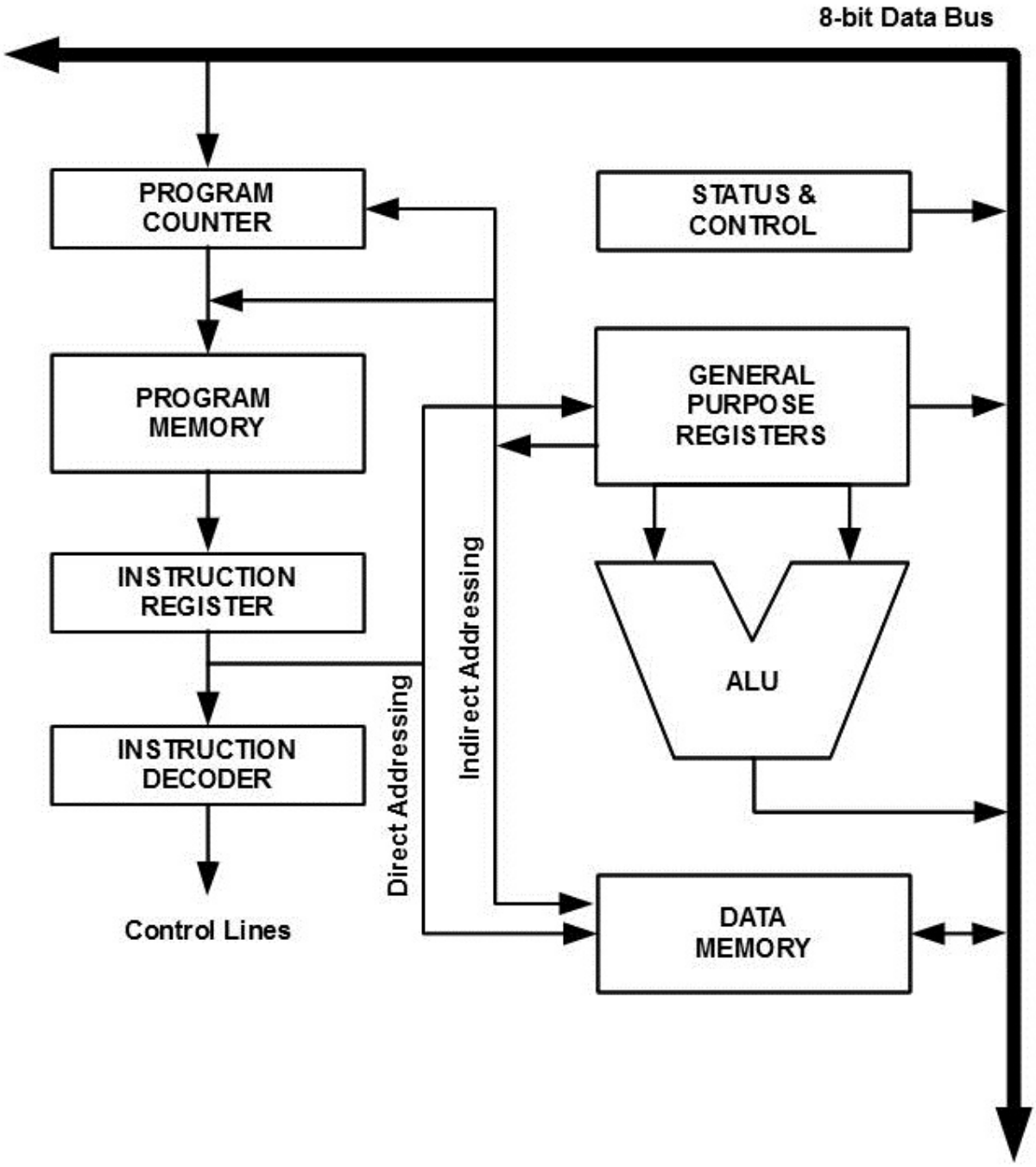

Figure 4 - Architecture of AVR core

The 32 registers in the core are able to be accessed quickly within a single clock cycle. Two registers from the register file may be accessed by the ALU simultaneously, perform an operation (like an add) and the result will be placed back in the register file all in a single clock cycle. The ALU supports arithmetic and logic operations between registers or between a constant and a register. The stack is allocated in SRAM, and must be initialized in the reset routine.

The operations that the ALU can perform are divided into three categories: arithmetic, logical and bit functions. Arithmetic functions include add and subtract. Logical functions include OR, AND, XOR, NOT and one's and two's complement. Bit operations include set, clear, shift and rotate.

Program execution, whether the program is written in assembly or the C programming language, starts at the reset vector. The reset vector is at address 0. Whenever the device is powered up (or comes out of reset) the program counter is set to address 0 (which is the address of the reset vector) and program execution begins. The routine pointed to by the reset vector initializes the stack and branches (or jumps) to the main program. The main program in the C language is main().

The vector table is an address map at address 0 that contains branch (or jump) instructions for every interrupt that the device contains. Table 1 is a description of all interrupts supported by the ATtiny2313A device.

Vector No.	Program Address	Label	Interrupt Source
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	INT1	External Interrupt Request 1
4	0x0003	TIMER1 CAPT	Timer/Counter1 Capture Event
5	0x0004	TIMER1 COMPA	Timer/Counter1 Compare Match A
6	0x0005	TIMER1 OVF	Timer/Counter1 Overflow
7	0x0006	TIMER0 OVF	Timer/Counter0 Overflow
8	0x0007	USART0 RX	USART0 Rx Complete
9	0x0008	USART0 UDRE	USART0 Data Register Empty
10	0x0009	USART0 TX	USART0 Tx Complete
11	0x000A	ANALOG COMP	Analog Comparator
12	0x000B	PCINT0	Pin Change Interrupt Request 0
13	0x000C	TIMER1 COMPB	Timer/Counter1 Compare Match B
14	0x000D	TIMER0 COMPA	Timer/Counter0 Compare Match A
15	0x000E	TIMER0 COMPB	Timer/Counter0 Compare Match B
16	0x000F	USI START	USI Start Condition
17	0x0010	USI OVERFLOW	USI Overflow
18	0x0011	EE READY	EEPROM Ready
19	0x0012	WDT OVERFLOW	Watchdog Timer Overflow

20	0x0013	PCINT1	Pin Change Interrupt Request 1
21	0x0014	PCINT2	Pin Change Interrupt Request 2

Table 1 - Interrupt Vectors for the ATtiny2313A

The vector table is resident in program memory at address 0. Table 1 shows an example of the vector table. Each instruction is a jump instruction. For example, when a reset condition occurs, the program counter is loaded with the address 0x0000 and a jump is made to the RESET routine.

```
0x0000 rjmp RESET
0x0001 rjmp INT0
0x0002 rjmp INT1
0x0003 rjmp TIM1_CAPT
0x0004 rjmp TIM1_COMPA
0x0005 rjmp TIM1_OVF
0x0006 rjmp TIM0_OVF
0x0007 rjmp USART0_RXC
0x0008 rjmp USART0_DRE
0x0009 rjmp USART0_TXC
0x000A rjmp ANA_COMP
0x000B rjmp PCINT0
0x000C rjmp TIMER1_COMPB
0x000D rjmp TIMER0_COMPA
0x000E rjmp TIMER0_COMPB
0x000F rjmp USI_START
0x0010 rjmp USI_OVERFLOW
0x0011 rjmp EE_READY
0x0012 rjmp WDT_OVERFLOW
0x0013 rjmp PCINT1
0x0014 rjmp PCINT2
```

In this course, the following interrupts will be utilized: Timer0 Compare A and External Interrupt 0.

Interrupts

An interrupt is an asynchronous event that happens outside of the main program loop that requires immediate attention. The processor is usually interfaced to a number of peripherals. Each peripheral needs periodic servicing, but not always at a predetermined interval. One solution to solve this problem is for the processor to poll each peripheral. During a polling operation, the processor queries the peripheral about its current state. The processor can then read and process the incoming bytes. This method is wasteful in terms of processing power. The

practical solution is for each peripheral to interface to an interrupt controller which will send a signal to the processor when a peripheral needs to be serviced.

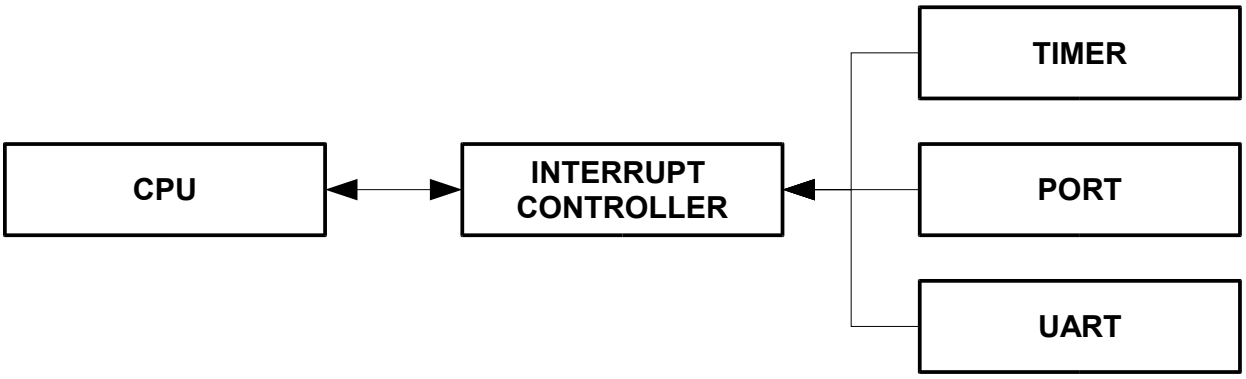

Figure 5 - Interrupt Controller, Processor and Peripherals

The interrupt controller sits between hardware peripheral devices and the processor. Its responsibility is to alert the processor when one of the hardware devices needs its immediate attention. When the hardware signals the interrupt controller that it needs attention, the processor stops its current activity, saves its current state (the program counter is saved and the stack contents are saved) and jumps to execute an interrupt service routine (or ISR) or interrupt handler. When the interrupt is serviced, the previously saved processor state is restored - the stack is restored and the previous value of the program counter is restored.

A peripheral such as a UART or a timer sends a signal to the interrupt controller when it needs to be serviced. The interrupt controller sends a signal to the processor which interrupts its current execution after executing its current instruction. When the processor is interrupted, it executes the appropriate interrupt handler. When an interrupt is signaled from the interrupt controller to the processor, the processor will jump to the set of instructions pointed to by the appropriate interrupt vector in the vector table.

An example of an interrupt is a receive character interrupt in a UART when a character is received. The interrupt service routine reads the character from the UART hardware and stores it in a buffer to be processed at a later time outside of the interrupt handler.

Other examples of interrupts are the timer overflow and timer compare match. A timer overflow interrupt occurs when the timer counter register is filled up and overflows. For example if the timer is an 8-bit timer, the counter register will overflow when the timer increments past a count of 255. When this happens, an interrupt to the processor occurs. Similarly, a timer compare match interrupt occurs when the timer counter register matches the timer compare match register. The timer compare match register is set by the programmer in another function.

Another example of an interrupt (exemplified in this course) is the external interrupt. An external interrupt is triggered by a change of state of a particular port pin. Depending on how the interrupt is initialized, an external interrupt will trigger when a port pin changes from low to high or from

high to low. In this course a button is used to trigger the external interrupt to signal the processor to turn on or off an LED.

Peripherals: Ports

In a sense the general purpose input/output (I/O) ports of a microcontroller are the hands and fingers of the processor. Just as fingers touch and sense their surroundings around them, an I/O pin when configured as an input can detect inputs from the outside world via pushbuttons and switches. In one of the course examples a pushbutton is used to activate an LED (which is connected to another I/O pin).

Just as humans use their hands to manipulate and control the world around them, microcontrollers use I/O port pins to control circuits that interact with the outside world. Port pins can be used to control LEDs directly. They can be used to turn switches on and off by using a FET (field effect transistor) directly or by using a FET interfaced to a relay or a solid-state switch. Port pins may also control motors through a circuit controlled by one or more FETs.

Port pins are usually bi-directional which means they can be configured either as an input or an output. The controller for the I/O ports usually consists of several registers. As for the Atmel AVR series, there are three registers: a data direction register (DDRx), an output register (PORTx) and an input register (PINx).

The data direction register (DDRx) is used to configure the port as an input or an output. Each bit in the register corresponds to a physical pin on the device. If a bit is written logic one, the corresponding pin is configured as an output. If a bit is written logic zero, the corresponding pin is configured as an input.

When configured as an output, the PORTx register is used to write a zero or one to the output pins. When configured as an input, the PINx register is used to read the input pins.

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0x38	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x37	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x36	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0

Table 2 - Register Description of PortB

Peripherals: Timer

A timer within a microcontroller is a peripheral that counts clock cycles. Timers within a microcontroller are used for many purposes. For example, a timer may be used within an application that keeps an LED on for a period of time after a button is pushed and switches the LED off after the preset time has expired.

A timer may be used as the basis of a task manager within a program. In this application the timer fires off an interrupt at preset rates (100ms, 200ms, 500ms, etc.). A separate task may be

serviced within these time intervals, such as to update a display, to service a state machine for a user interface, or to read or write data to another interface.

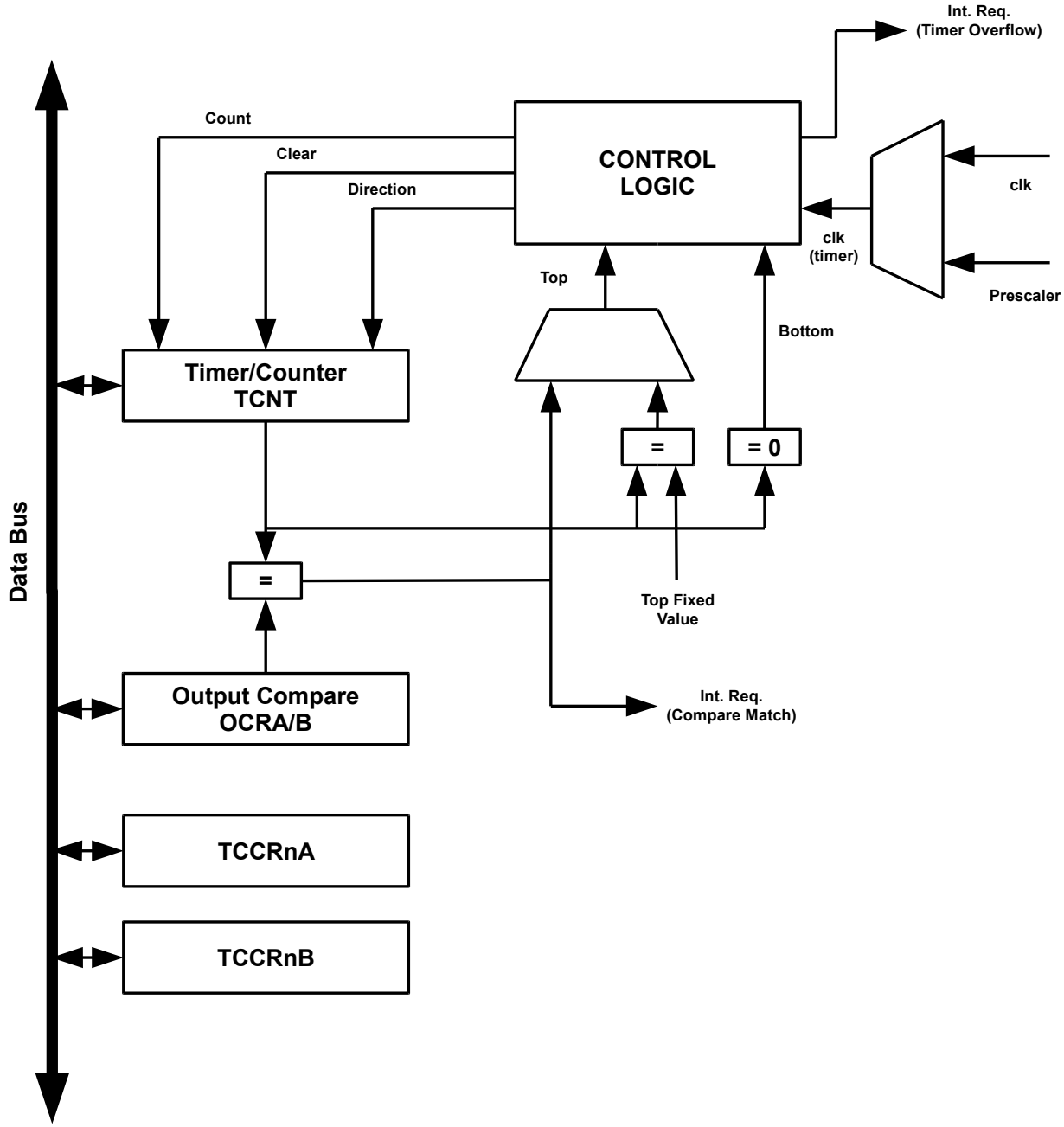

Figure 6 - Block Diagram of Timer Peripheral

Description

Simplified, the timer peripheral is just a counter. A counter is a device that counts up (or down) until it reaches its maximum value and then starts counting from zero again. When the counter

reaches it maximum value and one more tick is added, this condition is called overflow. The counter simply starts counting over again from zero. By keeping track of these overflow conditions, the counter then becomes a simple timer (or even a clock).

In clocks on the wall or on our desk that tell us the time, there is a counter in the clock that ticks away and another circuit (or mechanical mechanism for older clocks) that keeps track of the ticks and when a certain period of ticks lapse (like 60), then either a hand on the clock will advance by one tick, or a digital display will display the next number of minutes.

The timer peripheral counts clock cycles. The clock is connected to the system clock usually through a prescaler or a PLL (phase locked loop). The clock frequency is selected (via the prescaler) with the application in mind so that the desired time intervals are evenly divisible into the clock frequency. For example 32767 Hz is the perfect frequency for a real time clock because the counter register (if 16-bit) will count to 0x8000 exactly every second. This is easily counted and no floating point math is necessary.

The timer/counter register (TCNT0) is the heart of the timer peripheral; it is the register that stores the count value of the circuit. For 8-bit timers, this register is 8 bits in length, and for 16-bit timers, this register is 16 bits in length. The control logic circuitry checks the timer/counter register every cycle for an overflow condition or a match with one of the compare registers. The output compare registers (OCR0A/B) are registers that the programmer can set usually during initialization that will cause the timer circuit to send an interrupt signal to the processor every time the timer/counter register matches the compare register. The compare register is the same size as the timer/counter register (16-bits, for example). The compare register can be set to any value from 0 to the maximum value of the register (0xFFFF, for example for a 16-bit counter). When this condition occurs, the timer/counter register is cleared and resumes its count from zero.

Compare Mode

All timer circuits in any microcontroller will have a timer/counter register, and most will have one or more compare registers. The 8-bit or 16-bit comparator continuously compares the timer/counter register with the output compare register. If they are equal, then the comparator signals a match, and an interrupt signal is sent to the processor. In the Atmel AVR series of microcontrollers, the compare match circuit incorporates a mode that will either set or clear an I/O pin when the compare match criteria are met. This mode is used for generating waveforms including pulse-width-modulation (PWM) waveforms.

In the examples in this course, the timer peripheral is used to flash an LED. It will switch on an LED on for a period of time and switch off the LED for a period of time. The period of time is determined by the timer compare register. When the timer counter register reaches the value stored in the compare register, the timer signals an interrupt to the processor.

Input Capture Mode

The input capture circuit can capture an external event (detected by a high-to-low or a low-to-high transition on a pin) and give the event a time-stamp indicating time of occurrence. The time-stamps can be used to calculate frequency, duty-cycle, and other features of the signal applied. The time-stamps an also be used for creating a log of the events.

Register Description

The following register description applies to the Atmel AVR series of microcontrollers

Register	Description
TCCR0A	Timer/Counter0 Control Register A
TCCR0B	Timer/Counter0 Control Register B
TCCR0C	Timer/Counter0 Control Register C
TCNT0	Timer/Counter0
OCCR0A	Output Compare Register 0 A
OCCR0B	Output Compare Register 0 B
ICR0	Input Capture Register 0
TIMSK0	Timer/Counter Interrupt Mask Register 0
TIFR0	Timer/Counter Interrupt Flag Register 0
GTCCR	General Timer/Counter Control Register

Table 3 - Register Description of Timer Peripheral

Program Structure

The examples in this course are written in the C programming language. The programming code could also be written in assembly; however the programs would have to be written using the Atmel AVR instruction set. Although assembly language is similar from one processor family to the next, each processor family has its own native instruction set, each with different instructions and mnemonics.

Using a higher level language (such as C) allows the code to be relatively portable to a different target processor for all of the code that is not register-specific. It allows a developer to use the same programming language for most any processor family. It provides for ease of design and allows complex programs to be easily maintained. Also, the C programming language is very well known by programmers and software engineers in the industry. It is powerful, yet efficient in producing machine code which the processor executes.

All of the programs have a main() function and multiple other functions. The interrupt service routines (or ISR) are specific functions that are called when an interrupt signal is received by the core. When this occurs, the stack and program counter register values are saved and the particular ISR function is called. When the ISR returns, the stack and program counter register values are restored to their state before the interrupt occurred. The ISR functions are differentiated by the WinAVR compiler from other functions by the "ISR" and "__vector_ ## N" keywords (where N is the vector or interrupt number).

Projects

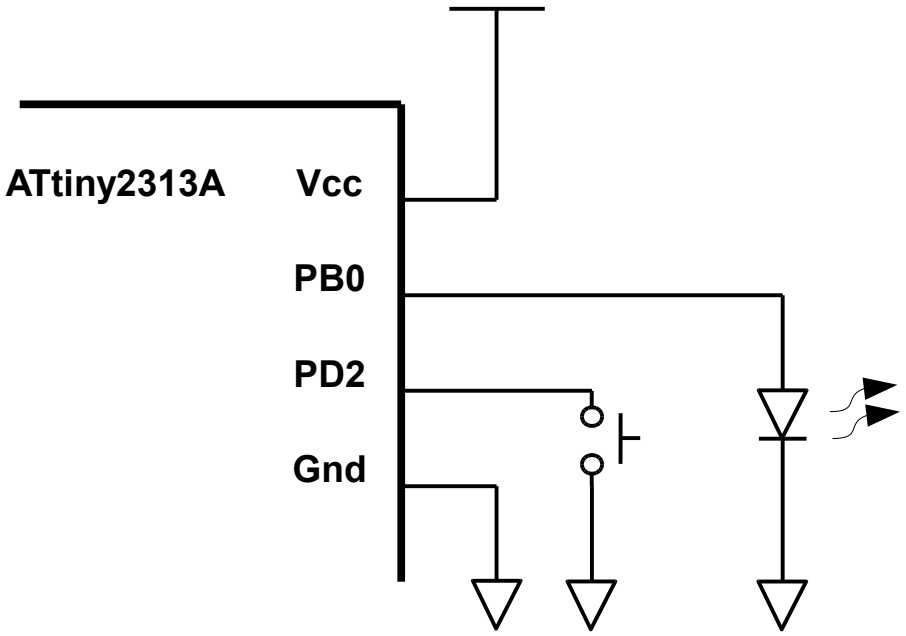

Figure 7 - Schematic Diagram of Project

Header File

The following is the “register.h” header file. The header file is included in the following projects to define certain register addresses and bit locations within the registers. The purpose of using a header file such as this is so that meaningful names can be used for variables (such as register names and bit locations) instead us using confusing numbers such as 0x38 for the PORTB register and 0x08 for pin PINB3.

```
#define DDRD   *(volatile unsigned char *)0x31
#define DDD0   0x01
#define DDD1   0x02
#define DDD2   0x04
#define DDD3   0x08
#define DDD4   0x10
#define DDD5   0x20
#define DDD6   0x40
#define DDD7   0x80

#define PIND   *(volatile unsigned char *)0x30
#define PIND0  0x01
#define PIND1  0x02
#define PIND2  0x04
#define PIND3  0x08
#define PIND4  0x10
#define PIND5  0x20
#define PIND6  0x40
#define PIND7  0x80

#define PORTD  *(volatile unsigned char *)0x32
#define PORTD0 0x01
```

```
#define PORTD1 0x02
#define PORTD2 0x04
#define PORTD3 0x08
#define PORTD4 0x10
#define PORTD5 0x20
#define PORTD6 0x40
#define PORTD7 0x80

#define PINB   *(volatile unsigned char *)0x36
#define PINB0  0x01
#define PINB1  0x02
#define PINB2  0x04
#define PINB3  0x08
#define PINB4  0x10
#define PINB5  0x20
#define PINB6  0x40
#define PINB7  0x80

#define DDRB   *(volatile unsigned char *)0x37
#define DDB0   0x01
#define DDB1   0x02
#define DDB2   0x04
#define DDB3   0x08
#define DDB4   0x10
#define DDB5   0x20
#define DDB6   0x40
#define DDB7   0x80

#define PORTB  *(volatile unsigned char *)0x38
#define PORTB0 0x01
#define PORTB1 0x02
#define PORTB2 0x04
#define PORTB3 0x08
#define PORTB4 0x10
#define PORTB5 0x20
#define PORTB6 0x40
#define PORTB7 0x80


#define TCCR0A *(volatile unsigned char *)0x50
#define WGM00  0x01
#define WGM01  0x02
#define COM0B0 0x10
#define COM0B1 0x20
#define COM0A0 0x40
#define COM0A1 0x80

#define TCNT0  *(volatile unsigned char *)0x52

#define TCCR0B *(volatile unsigned char *)0x53
#define CS00   0x01
#define CS01   0x02
#define CS02   0x04
#define WGM02  0x08
#define FOC0B  0x40
#define FOC0A  0x80

#define MCUCR  *(volatile unsigned char *)0x55
#define ISC00  0x01
#define ISC01  0x02
#define ISC10  0x04
#define ISC11  0x08
#define SM0    0x10
#define SE     0x20
#define SM1    0x40
#define PUD    0x80

#define OCR0A  *(volatile unsigned char *)0x56

#define TIMSK  *(volatile unsigned char *)0x59
#define OCIE0A 0x01
```

```
#define TOIE0  0x02
#define OCIE0B 0x04
#define ICIE1  0x08
#define OCIE1B 0x20
#define OCIE1A 0x40
#define TOIE1  0x80

#define GIMSK  *(volatile unsigned char *)0x5B
#define PCIE   0x20
#define INT0   0x40
#define INT1   0x80
```

LED Flasher with Delay Loop

The first project blinks an LED on and off in an infinite loop. There is a short delay between turning the LED on and turning it off. The simplest form of delay is a loop that counts to a certain value and then exits the loop. In many cases, the delay loop must contain some instruction (such as setting a port pin) instead of just an empty loop. The empty delay loop is often times optimized out by the compiler.

The processor is running off the internal 8MHz oscillator as the system clock for all projects in this course. Pin PORTB0 is connected to an anode of the LED and the cathode is connected to ground. The pin PORTB0 is initialized as an output by setting the DDB0 bit to a one in the DDRB (data direction) register. There are no interrupts utilized in this example.

As with most embedded systems, the main program loop is an infinite loop that never exits (as long as the processor is powered up). The main() function first calls the Initialize() function which calls the InitializePorts() function which initializes PORTB. Program execution then enters the main program loop. The program loop toggles the variable t and depending on whether t = 0 or t = 1, the pin PORTB0 is either cleared (turning off the LED) or set (turning on the LED). The Delay() function is then called that counts to a value (16834 in this example) which gives the flasher about a 200 millisecond (0.2s) on/off duty cycle with the processor running at 8MHz.

```
#include "register.h"

void Initialize(void);
void InitializePorts(void);
void Delay(void);

/*****************************************************************************
 *
 * Function: Main
 *
 * This function is the main program loop.
 *
 *****************************************************************************/
void main()
{
    unsigned char t = 1;

    Initialize();

    // main program loop (infinite)
    while(1)
    {
        if (t == 1)
        {
```

```
            // turn port pin PB0 off
            PORTB = 0;
        }
        else
        {
            // turn port pin PB0 on
            PORTB |= PORTB0;
        }

        t ^= 1;
        Delay();
    }
}


/*****************************************************************************
 *
 * Function: Initialize
 *
 * This function initializes the system.
 *
 *****************************************************************************/
void Initialize(void)
{
    InitializePorts();
}


/*****************************************************************************
 *
 * Function: InitializePorts
 *
 * This function initializes the input/output ports.
 *
 *****************************************************************************/
void InitializePorts(void)
{
    // set direction of PORTB pins
    DDRB = DDB0 | DDB1;
}


/*****************************************************************************
 *
 * Function: Delay
 *
 * This function introduces a delay. The port is written to so that the code
 * will not be optimized out.
 *
 *****************************************************************************/
void Delay(void)
{
    unsigned int i = 0;

    for(i = 0; i < 16834; i++)
    {
        PORTB |= PORTB1;
    }
}
```

LED Flasher Using Timer

The second project in this course is similar to the first project in that it blinks an LED in an infinite loop. The difference between this project and Project #1 is that this one utilizes the timer for the delay between turning the LED on and off. The advantage of using a timer instead of a

delay loop as in Project #1 is that the timer gives a more exact (and consistent) approach for time keeping.

The main program loop is empty because the LED is serviced in the interrupt service routine for Timer0. The timer is configured for compare mode (or compare match mode). As in the first project the main program loop is an infinite loop that never exits, since this would terminate all execution.

The main() function calls the Initialize() function which calls the functions InitializePorts() and InitializeTimer(). The Initialize() function also calls sei() which is a macro that sets the global interrupt enable bit. InitializePorts() sets the pins PORTB0 and PORTB1 as outputs by setting the corresponding bits in the DDRB (data direction) register.

The InitializeTimer() function initializes Timer0. The WGM01 bit is set in the TCCR0A register to configure Timer0 to clear the counter register on compare match (compare mode). The prescaler is set in the TCCR0B register using bits CS00, CS01 and CS02. The prescaler in this example is set to 1024, which will divide the timer frequency by 1024. This sets the timer clock frequency to 8MHz / 1024 = 8kHz. The compare value is loaded with a value of 244 which causes an compare match interrupt to occur when the timer counter register counts to a value of 244. The compare match interrupt is enabled by setting the OCIE0A bit in the TIMSK register.

The interrupt service routine (ISR) for the timer compare match interrupt is preceded by the notation ISR(_VECTOR(13)). This notation is specific to the WinAVR compiler used in this course to designate the ISR for vector 13 (or Timer/Counter 0 Output Compare Match A Interrupt). When this interrupt fires, the program counter is loaded with the value 0x000D which is a jump instruction (rjmp) to the timer compare match interrupt service routine. Table 1 shows the corresponding vector number associated with interrupt. The value of "13" that the compiler uses is actually the 14th vector since the compiler starts numbering the vectors starting with 0.

In the timer compare match ISR, the LED connected to pin PORTB0 is turned on and off. The LED is toggled each time it executes the ISR. This creates a 50% duty cycle for the LED (equal amount of time on and off) at a rate of 500 milliseconds (0.5s).

```
#include "register.h"

#define TIMER_COMPARE_VALUE   244


#define _VECTOR(N) __vector_ ## N
#define sei()  __asm__ __volatile__ ("sei" ::)

#define ISR(vector, ...)            \
    void vector (void) __attribute__ ((signal,__INTR_ATTRS)) __VA_ARGS__; \
    void vector (void)

void Initialize(void);
void InitializePorts(void);
void InitializeTimer(void);


/************************************************************************
 *
 * Function: Main
 *
 * This function is the main program loop.
 *
 ************************************************************************/
```

```
void main()
{
    Initialize();

    // main program loop (infinite)
    while(1)
    {
    }
}


/*************************************************************************
 *
 * Function: Initialize
 *
 * This function initializes the system.
 *
 *************************************************************************/
void Initialize(void)
{
    InitializePorts();
    InitializeTimer();

    // global interrupt enable
    sei();
}


/*************************************************************************
 *
 * Function: InitializePorts
 *
 * This function initializes the input/output ports.
 *
 *************************************************************************/
void InitializePorts(void)
{
    // set direction of PORTB pins
    DDRB = DDB0 | DDB1;
}


/*************************************************************************
 *
 * Function: InitializeTimer
 *
 * This function initializes the timer.
 *
 *************************************************************************/
void InitializeTimer(void)
{
    // set Timer0 for clear timer on compare match (CTC) mode
    TCCR0A = WGM01;

    // set timer frequency to clkIO/1024 (From prescaler)
    TCCR0B = CS02 | CS00;

    // load compare value
    OCR0A = TIMER_COMPARE_VALUE;

    // enable the Timer/Counter0 Output Compare Match A Interrupt
    TIMSK = OCIE0A;
}


/*************************************************************************
 *
 * Function: ISR (TIMER0 COMPA)
 *
 * This function is the interrupt service routine for the Timer0 Compare A.
 *
```

```
 ***************************************************************************/
ISR(_VECTOR(13))
{
    static unsigned char t = 1;

    // if the t is set - turn PB0 off, if t is clear - turn PB0 on
    if (t == 1)
    {
        // turn port pin PB0 off
        PORTB &= ~PORTB0;
    }
    else
    {
        // turn port pin PB0 on
        PORTB |= PORTB0;
    }

    // toggle for next time
    t ^= 1;
}
```

LED Flasher with Push Button

The third project in this course is similar to the second project in that it blinks an LED. However, the difference here is that the LED does not begin to flash until a pushbutton (attached to INT0 - or pin PORTD2) is pressed and released. When the button is pressed, the LED will blink on and off three times (still at a 50% duty cycle) and then turn off.

The main() function calls the Initialize() function which calls the InitializePorts(), InitializeExtInt() and InitializeTimer() functions. As in the second project, the Initialize() function calls sei() to set the global interrupt enable bit. The InitializePorts() function sets the PORTB0 and PORTB1 pins as an output by setting the appropriate pins in the data direction register (DDRB) to a one, and sets the PORTD2 pin as an input by setting the data direction register (DDRD) to a zero. The InitializeExtInt() function sets the pin PORTD2 (also known as INT0 - or external interrupt 0) to trigger on a rising edge. The external interrupt is set in the GIMSK register.

The InitializeTimer() function initializes the timer in the same way as in the second project. The timer compare match ISR is preceded by the notation ISR(_VECTOR(13)) and the external interrupt is preceded by the notation ISR(_VECTOR(1)).

In this example, the external interrupt (triggered by a low-to-high transition on pin PORTD2) enables the timer compare match interrupt. When the timer compare match interrupt is enabled the ISR triggers after a timer match and turns the LED on and off three times then disables the timer interrupt. The cycle repeats itself when the button is pushed again.

One thing to note here is that without some debounce circuitry on the pushbutton attached to the PORTD2 pin the interrupt may fire several times in a row in a rapid sequence. A better approach for an actual product would be to add a series resister and a capacitor in parallel to act as a low-pass filter on the line. Another approach would be to add some software logic to debounce the switch.

```
#include "register.h"

#define TIMER_COMPARE_VALUE   244
```

```
#define _VECTOR(N) __vector_ ## N
#define sei()  __asm__ __volatile__ ("sei" ::)

#define ISR(vector, ...)            \
    void vector (void) __attribute__ ((signal,__INTR_ATTRS)) __VA_ARGS__; \
    void vector (void)


void Initialize(void);
void InitializePorts(void);
void InitializeExtInt(void);
void InitializeTimer(void);


/************************************************************************
 *
 * Function: Main
 *
 * This function is the main program loop.
 *
 ************************************************************************/
void main()
{
    Initialize();

    // main program loop (infinite)
    while(1)
    {
    }
}


/************************************************************************
 *
 * Function: Initialize
 *
 * This function initializes the system.
 *
 ************************************************************************/
void Initialize(void)
{
    InitializePorts();
    InitializeExtInt();
    InitializeTimer();

    // global interrupt enable
    sei();
}


/************************************************************************
 *
 * Function: InitializePorts
 *
 * This function initializes the input/output ports.
 *
 ************************************************************************/
void InitializePorts(void)
{
    // set direction of PORTB pins
    DDRB = DDB0 | DDB1;

    // set direction of PORTD pins
    DDRD = 0;
}


/************************************************************************
 *
```

```
 * Function: InitializeExtInt
 *
 * This function initializes the external interrupts.
 *
 ******************************************************************************/
void InitializeExtInt(void)
{
    // set rising edge of INT0 to generate an interrupt request
    MCUCR = ISC01 | ISC00;

    // enable External Interrupt Request 0
    GIMSK = INT0;
}


/******************************************************************************
 *
 * Function: InitializeTimer
 *
 * This function initializes the timer.
 *
 ******************************************************************************/
void InitializeTimer(void)
{
    // set Timer0 for clear timer on compare match (CTC) mode
    TCCR0A = WGM01;

    // set timer frequency to clkIO/1024 (From prescaler)
    TCCR0B = CS02 | CS00;

    // load compare value
    OCR0A = TIMER_COMPARE_VALUE;
}


/******************************************************************************
 *
 * Function: ISR (TIMER0 COMPA)
 *
 * This function is the interrupt service routine for the Timer0 Compare A.
 *
 ******************************************************************************/
ISR(_VECTOR(13))
{
    static unsigned char t = 1;
    static unsigned char cnt = 0;

    if (cnt <= 8)
    {
        // if the t is set - turn PB0 on, if t is clear - turn PB0 off
        if (t == 1)
        {
            // turn port pin PB0 on
            PORTB |= PORTB0;
        }
        else
        {
            // turn port pin PB0 off
            PORTB &= ~PORTB0;
        }

        // toggle for next time
        t ^= 1;

        // increment counter
        cnt++;
    }
    else
    {
        // disable timer interrupt
        TIMSK = 0;
```

```
        // turn port pin PB0 off
        PORTB &= ~PORTB0;

        // reset counter
        cnt = 0;
    }
}


/*****************************************************************************
 *
 * Function: ISR (INT0)
 *
 * This function is the interrupt service routine for external interrupt 0.
 *
 *****************************************************************************/
ISR(_VECTOR(1))
{
    // enable the Timer/Counter0 Output Compare Match A Interrupt
    TIMSK = OCIE0A;
}
```

Summary

The Atmel ATtiny2313A microcontroller used for these exercises is packaged in a 20-pin DIP package. It is small compared with other microprocessors available, but there are other smaller devices available. For example the Atmel ATtiny10 is packaged in a 6-pin SOT-23 package, making this device ideal for small projects such as an electronic candle to simulate the flicker of the flame or a child's shoe that has LEDs that light up when he or she walks.

Although the Atmel AVR series of microcontrollers is a great product offering many features with a powerful core, this course is not meant to be an advertisement for the Atmel AVR series of microcontrollers. There are many similar microcontrollers available on the market, such as the Microchip PIC, Texas Instruments MSP430, Intel 8051, STMicroelectronics STM8, Freescale 68HC11, and multiple versions of the ARM core from many vendors.

Microcontrollers today can be designed and programmed to control and monitor almost anything. They have become an integrated part of our society, industry and culture.

References

1. "Atmel 8-bit AVR® Microcontroller with 2/4k Bytes In-System Programmable Flash: ATtiny2313A/ATtiny4313" September 2011 <http://www.atmel.com/Images/doc8246.pdf>

2. "Atmel 8-bit AVR® Microcontroller with 512/1024 Bytes In-System Programmable Flash: ATtiny4/5/9/10" November 2011 <http://www.atmel.com/Images/doc8127.pdf>

3. “ProtoStack – USBASP driver for Windows 7 and Windows Vista x64” visited 1 September 2012 <http://www.protostack.com/blog/2011/05/usbasp-driver-for-windows-7-and-windows-vista-x64/>

4. “WinAVR: WinAVR-20100110” January 2010 <http://winavr.sourceforge.net/>

Final Exam - Designing with Microcontrollers

1. The following components are all design considerations when designing a microcontroller-based system: the microcontroller, the compiler, and the device programmer.
    a. true
    b. false

2. The following parameters need to be considered when selecting a microcontroller for a system of any complexity: power consumption, processor bus width, peripheral selection, processor speed, and _____.
    a. temperature
    b. humidity
    c. the amount of program memory and data memory
    d. barometric pressure

3. The purpose of using a compiler (such as one that compiles programs written in the C programming language) is _____.
    a. to make the software code as small as possible
    b. to complicate program maintenance issues
    c. to prevent the developer from having to use processor-specific assembly code instructions
    d. to make the processor run faster

4. The program counter is the register that contains the program address of the current instruction being executed. Using the interrupt vector table for the ATtiny2313A in the course the value of the program counter immediately after a "Timer/Counter0 Compare Match A" interrupt fires is _____.
    a. 0x0000
    b. 0x0001
    c. 0x000C
    d. 0x000D

5. The term reduced instruction set computer (RISC) when compared to an x86-based processor implies that it _____.
    a. has fewer yet more powerful instructions
    b. is not as powerful as an x86-based processor (running at the same speed)
    c. is smaller in size
    d. consumes more power

6. The Harvard architecture employs a computer architecture _____.
    a. in which program memory and data memory exist in the same memory space
    b. that uses more registers
    c. in which program memory and data memory can be accessed simultaneously
    d. that employs longer bus lines

7. Polling a peripheral is much faster than using an interrupt to signal the processor that a peripheral needs servicing.
    a. true
    b. false

8. The purpose of an interrupt controller is to _____.
    a. allow two or more peripherals to communicate with one another
    b. copy the stack
    c. signal the processor that a peripheral needs servicing
    d. save memory

9. When an interrupt event occurs _____.
    a. the peripheral sends a signal to the interrupt controller
    b. the interrupt controller signals the processor that one of the peripherals needs to be serviced
    c. the current state of the processor (program counter and stack) is saved
    d. all of the above

10. The register that sets the direction of the port pins to either input or output is the _____.
    a. special function register
    b. program counter register
    c. data direction register
    d. timer counter register

11. The highest value that the counter register can hold in a 16-bit timer is _____.
    a. 16
    b. 255
    c. 65535
    d. 32767

12. A timer peripheral in a microcontroller can be used as the time basis for the following functions except a(n) _____.
    a. pulse width modulation circuit
    b. task manager for an operating system
    c. periodic servicing of a particular function
    d. LED driver to drive an LED with 100mA of current

13. In the timer compare match mode, when the compare match interrupt is set and the counter register equals the compare register _____.
    a. a compare match interrupt occurs and the counter register is set to zero and resumes counting from zero
    b. a compare match interrupt occurs and the counter register keeps counting from the same value
    c. a compare match interrupt occurs and the counter stops counting

d. a timer overflow interrupt occurs

14. In the C programming language header files such as the "register.h" header file used in the course projects are used _____.
    a. to make the overall length of the program shorter
    b. to use the same name for constants from one module to another
    c. to assign meaningful names to constants and registers
    d. all of the above

15. An external interrupt in a microcontroller is used to _____.
    a. signal the processor that a timer overflow condition has occurred
    b. signal the processor that an external event has occurred on a port pin such as a button press
    c. signal the processor that the external temperature is over its design limit
    d. all of the above

16. In the second project of the course a timer compare match interrupt will occur when the counter register reaches the value of _____.
    a. 65535
    b. 240
    c. 244
    d. 0

17. The bits in the data direction register are set to a one to correspond to an output on a port.
    a. true
    b. false

18. In the third project in the course the external interrupt service routine is used to _____.
    a. enable the timer output compare match interrupt
    b. initialize the port pins
    c. define the interrupt vector locations
    d. initialize the system

19. A typical bus width (or the number of bits that the core can process simultaneously) for a microcontroller is _____.
    a. 10 bits
    b. 67 bits
    c. 8 bits
    d. 1 bit

20. A processor architecture in which program memory and data memory share the same bus (so they cannot be accessed simultaneously) is called the _____.
    a. Harvard architecture
    b. von Neumann architecture
    c. Carnegie architecture

d. Morgan architecture

21. The processor's core (or central processing unit) is responsible for _____.
    a. performing calculations
    b. handling interrupts
    c. accessing program and data memories
    d. all of the above

22. The arithmetic logic unit _____.
    a. performs add, subtract, AND, OR, NOT, and bit shift operations
    b. copies blocks of memory from one location to another
    c. controls the clock
    d. handles interrupts

23. The first vector (or address) within the interrupt vector table is usually the _____.
    a. time overflow interrupt
    b. port interrupt
    c. reset vector
    d. UART receive interrupt

24. A microcontroller can be clocked by either an internal clock (or oscillator) or an external clock.
    a. false
    b. true

25. The address map that is used when an interrupt occurs is called the _____.
    a. register association
    b. instruction decoder
    c. program counter
    d. vector table