ECE 353 Lab 2

General MIDI Explorer

Professor Daniel Holcomb
UMass Amherst
Fall 2018
Intro to Instructor

- UMass prof since 2015
  - (BS/MS@UMass, PhD@Berkeley, Researcher@UMichigan)
- Research Interests
  - Modeling and Verification of Embedded Systems: 622/597MB
  - VLSI Circuit Design: 558/559
  - Hardware Security: 591CF
  - Formal Verification
- Past projects
  - Atmospheric research balloon payload
  - IC fingerprinting
  - QoS proofs for on-chip networks
- Current research
  - Chip design for ultra-low energy encryption
  - Circuit camouflaging and reverse engineering
  - FPGA Security — attacks and defenses
  - Security of IC Supply Chain

with P. Voss, Smith College
Schedule

- 10/8: Lab 1 due (Prof. Krishna)
- 10/29: Lab 2 due on Moodle, will post signups for check-off

- Lab component of the course does not have regular lectures
- Two mandatory lectures for introducing lab 2 (10/2 and 10/4)
  - We don’t take attendance
  - We will introduce everything about the lab in these lectures, so it is important to attend
  - Lab kits signed out in Duda Hall after lecture on 10/4
- After that will have optional discussion sections during class time
Class Information for Labs 2 and 4

- [http://ece353.ecs.umass.edu](http://ece353.ecs.umass.edu)

- Getting help
  - TA office hours in Duda Hall: M/W/F 7-9pm, Tu/Th 6-9pm.
  - My office hours: Tu/Th 10AM-11AM in KEB 309H
  - Discussion sections in lab Tu/Th 1-2:30 (after the lab lectures finish)
    - No set material to cover
    - Opportunity for Q&A
    - Clearing up points of confusion
    - In-depth discussions on topics from lab
  - Email me for appointment to meet at other times

- Grading
  - Lab 2 and Lab 4 equal weight (each 17.5% of final grade)
  - Final exam (30% of grade) – 50% of questions from Lab 2 and 4

- Lab Kits contain **everything** needed for lab

- Duda Hall
  - Suggested workspace but not required
  - You have access 24 hrs/day
  - **No food or beverages in Lab!**
Grading for Labs 2 and 4

- **Demo**
  - Schedule will be posted online, sign up for slots
  - Lab assessment (on course webpage)
    - Assessment shows exactly what we will test at demo
    - Complete by you and submitted with report (to help you prepare)
    - Completed by us at demo (this one counts for your grade)

- **Report**
  - Submitted on Moodle by 11:55pm on due date
  - Include commented code as separate file
  - See assignment for details
  - Be concise in report

- Changing code between submission and demo not allowed (we check)

- Lab 2 due 10/29 on Moodle, will post signups for demo slots after deadline

- **Late submissions:** 20 points per day penalty
**Computer Systems *Lab***

- Focus is on integrating hardware and software
- Prototyping systems on a breadboard
  - First step in building embedded systems
  - PCB design comes later (not in this course)
- Fundamental skill for career and future courses

- Building systems means confronting reality
  - This is a feature of lab, not a bug
  - Learning to debug with support

- Some well known examples of pesky bugs in real systems
  - Challenger Explosion (1986): o-ring failure in cold weather
  - Faster-than-light neutrino anomaly (2011-2012): bad connection on fiber connection that carries laser pulse

---

**Time spent on Lab**

- **Green**: Testing and Debugging
- **Blue**: Writing Report
- **Cyan**: Planning/Writing code
Teaching Philosophy for Labs

- Each group will encounter and resolve unique bugs
- Debugging is detective work

“An LED driven by pin A1 is not lighting up. I’ve verified that the program executes the line of code that sets PORTA, and can see the signal go high in the watch window. However, the output pin on the board only goes to 1.4V even when no LED is attached, and we expect it to go to 5.0V.”

“LEDs are not lighting up”
Technical Introduction to Lab Assignment
Lab Objectives

- Building a complete system (General MIDI Explorer)
- Record MIDI packets from computer when prompted
- Store packets on breadboard
- Show notes on LEDs
- Playback (with modification) to computer

![Diagram of sensor and LED connections to an ATmega32 microcontroller, with MIDI and USB connections to a computer.]
Lab Objectives

- Exposure to microcontroller programming
  - Lower abstraction than first lab
  - Programming for a specific processor
  - Focus is on interaction with hardware components
  - The program logic is the easy part

- Introduction to MIDI
  - Serial communication
  - Notes and instruments
  - Will use MIDI in 4th lab as well

- Intended to be a FUN project
Schematic

PDF of schematic on course website
On Breadboard

Tip: when building your design, use schematic and don’t try to copy circuit photograph
ATmega32 AVR

- ≤16MHz CMOS 8-bit microcontroller with AVR RISC instruction set
- Harvard architecture (separate program and data memory)
- Less than $5 (in large quantity)
- 32 general purpose registers
- 32KB of Flash (program memory)
- 2Kb internal SRAM
- 1KB EEPROM – 100k erase cycles
- 8 and 16 bit counters
- USART/SPI/I2C
- 8-channel 10-bit ADC
- JTAG
- 4 8-bit I/O ports
- Less than 10mW when active
Flexible Tool. Many Configurations. Use Carefully
Special-purpose Registers

- AVR uses memory-mapped architecture
  - Program interacts with the hardware modules by writing/reading to/from registers at specific addresses
  - These registers have designated meanings
  - We refer to these by names instead of addresses
  - Translation from names to addresses is accomplished by macro definitions in a header file

- Important concept: named registers are not variables
  - Some are writable, some are readable, some are readable and writeable
  - Values can get modified by the hardware
A Simple C Program

```c
#include <avr/io.h>    // Standard AVR header
#include <avr/delay.h> // Delay loop functions

int main(void) {
    DDRA = 0xFF;    // set Port A direction as output
    while (1) {
        for (int i=1; i<=128; i=i*2) {
            PORTA = i; // write to Port A
            _delay_loop_2(30000);
        }
    }
}
```

The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Read/Write</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero).

<table>
<thead>
<tr>
<th>Bit</th>
<th>Read/Write</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>
**Special-purpose Registers**

- When assigning values to special registers, what happens on the processor?
  - (e.g. `DDRA = 0xFF`)
  - The program writes a value to a designated memory address
  - The hardware module that controls Port A is configured by that register

- When reading special register values, what happens on the processor?
  - (e.g. `foo = PINA`)
  - The program reads a value from designated memory address
  - The hardware module that controls Port A knows to make pin values accessible in that address

---

From ATmega32 datasheet
Configuring Special-purpose Registers

- Included header file `<io.h>` will further include the appropriate definitions for the selected processor.
- `<iom32.h>` defines macros that replace names with addresses in the program.
Overview of Lab 2

- Write C program for AVR ATmega32
- Use AVR Studio and AVR gcc compiler to compile
- Program and debug AVR through JTAG
- Program controls hardware modules on microcontroller and causes project to record and playback music as intended
Hardware Modules for lab
Hardware Components

1. Communication using USART and opto-isolation circuit
   • To send and receive MIDI notes

2. Counter and Interrupts
   • To keep track of timing between notes for recording and playback

3. Storage with EEPROM
   • To store notes and timing on microcontroller

4. I/O
   • LEDs to show notes
   • Switches to choose record or playback mode
   • ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
Hardware Components

- **1. Hardware Components**
  - Communication using USART and opto-isolation circuit
  - ADC (Analog to Digital Converter) to digitize the amount of light
  - Switches to choose record or playback mode
  - LEDs to show notes
  - To send and receive MIDI notes
  - 4MHz CLK
  - NC
  - GND
  - OUT
  - VCC

- **2. Diagram**
  - IC1
  - (ADC7)PA7
  - (ADC6)PA6
  - (ADC5)PA5
  - (ADC4)PA4
  - (ADC3)PA3
  - (ADC2)PA2
  - (ADC1)PA1
  - (ADC0)PA0
  - (SCK)PB7
  - (MISO)PB6
  - (MOSI)PB5
  - (SS)PB4
  - (AIN1/OC0)PB3
  - (AIN0/INT2)PB2
  - (T1)PB1
  - (TO/XCK)PB0
  - (TOSC2)PC7
  - (TOSC1)PC6
  - (TDI)PC5
  - (TDO)PC4
  - (TMS)PC3
  - (TCK)PC2
  - (SDA)PC1
  - (SCL)PC0
  - (OC2)PD7
  - (ICP)PD6
  - (OC1A)PD5
  - (OC1B)PD4
  - (INT1)PD3
  - (INT0)PD2
  - (TXD)PD1
  - (RXD)PD0

- **3. Additional Notes**
  - RESISTOR_BAR
  - SLIDER_1
  - GND
  - VCC
  - MEGA256
  - OPITO
  - 6N138
  - MIDJACK
  - PC TO BOARD

- **4. Diagram Elements**
  - MEGA256
  - OPITO
  - 6N138
  - MIDJACK
  - PC TO BOARD
Hardware Components
MIDI

- Musical Instrument Digital Interface
- Common hardware interface and protocol
- Allows electronic musical devices to communicate with each other
- MIDI messages are transmitted asynchronously (no shared clock)
- We will send/receive MIDI messages using microcontroller USART
  - USART – Universal Synchronous/Asynchronous Receiver Transmitter
  - Flexible module with configurable data format, baud rates, etc.

Special effects at Universal Studios orchestrated using MIDI show control
MIDI Specification

- Start and end of note each signified by three transmitted bytes (6 total bytes to push and release a key)
- Byte 1: Status byte (MSB is on)
  - Byte 1 contains code for Note On, Off, other ctrl, ChID
- Bytes 2,3: Data bytes (MSB is off)
  - Byte 2 contains the note number (128 different notes, 10 octaves)
  - Byte 3 contains velocity of note (how hard is key/instrument pressed)
Decoding of a MIDI Message

- 31,250 bits/s fixed baud rate
- Bit time (BT) is 32µs
- With START & STOP bits, a MIDI frame is 10BT, 320µs
  - You will see this on scope or logic analyzer
- For lab 2, the microcontroller’s USART handles receive/transmit
- For lab 4, will be sampling MIDI values with your own hardware design
Microcontroller USART

- USART – Universal Synchronous/Asynchronous Receiver Transmitter
- Flexible hardware module that can implement many different serial communication protocols
- After configuration, USART handles the low-level communication details
- We interact with communication by write/read to/from USART registers
- What USART settings for MIDI?
  - USART Baud rate = 31,250
  - USART mode = Asynchronous
  - 8 data bits, 1 stop bit, no parity bits
  - Receive and Transmit both on
- Your job is to translate these settings into appropriate register values using datasheet as a guide
Using the USART

- **UCSRA/UCSRB/UCSRC** (USART Control Status Register A/B/C)
- **UBRRH/UBRRL** (USART Baud Rate Register High/Low)
- Will have to use datasheet to figure out details of how to use these registers to control and configure USART settings. Datasheet includes code examples.

### USART

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RXC</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TXC</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UDRE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOR</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PE</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U2X</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPCM</td>
<td>R/W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Read/Write**
- RXC: Read
- TXC: Read/Write
- UDRE: Read
- FE: Read
- DOR: Read
- PE: Read
- U2X: Read/Write
- MPCM: Read/Write

**Initial Value**
- RXC: 0
- TXC: 0
- UDRE: 1
- FE: 0
- DOR: 0
- PE: 0
- U2X: 0
- MPCM: 0

The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a highly flexible serial communication device. The main features are:
- Full Duplex Operation (Independent Serial Receive and Transmit Registers)
- Asynchronous or Synchronous Operation
- Master or Slave Clocked Synchronous Operation
- High Resolution Baud Rate Generator
- Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits
- Odd or Even Parity Generation and Parity Check Supported by Hardware
- Data OverRun Detection
- Framing Error Detection
- Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter
- Three Separate Interrupts on TX Complete, TX Data Register Empty, and RX Complete
Communication Circuit: Opto-isolator

- Voltage-based communication only works if devices share a reference (ground)
  - PC and microcontroller may not
- Solution: opto-isolator
- Opto-isolator relays signals from inputs to outputs, but with no electrical connection between input circuit and output circuit
- How does that work?
  - Light-based signals inside chip package
  - LED in transmitter circuit
  - Photosensor in receiver circuit
- Communication between isolated circuits

**Circuit 1**
with own power/ground
e.g. PC

**Circuit 2**
with own power/ground
e.g. breadboard circuit
Opto-isolator

- Your breadboard will only have opto-isolator on the MIDI input to the microcontroller
- Why not on the microcontroller output (i.e. PC input)?
- This opto-isolator is in MIDI-to-USB converter
The two MIDI ports are the same part. The pins are numbered the same, because a MIDI cable would cross the signals internally. If you hooked up a MIDI cable from one port to another, the numbers would match up as shown at right (note that OUT is flipped).
Hardware Components

1. Communication using USART and opto-isolation circuit
   • To send and receive MIDI notes

2. Counter and Interrupts
   • To keep track of timing between notes for recording and playback

3. Storage with EEPROM
   • To store notes and timing on microcontroller

4. I/O
   • LEDs to show notes
   • Switches to choose record or playback mode
   • ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
Hardware Components

1. Communication using USART and opto-isolation circuit
   - To send and receive MIDI notes

2. Counter and Interrupts
   - To keep track of timing between notes

3. Storage with EEPROM
   - To store notes and timing on microcontroller

4. I/O
   - LEDs to show notes
   - Switches to choose record or playback mode
   - ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
**Timer 1**

- Timer1 is 16-bit counter implemented in hardware on the microcontroller
- Runs in parallel to the CPU
- Can interact with timer by resetting and reading its value (i.e. to measure an interval of time)
- Can configure timer to cause interrupts on certain events
## Timer 1

You will use Timer1 module for two purposes in this lab

1. To measure the time between MIDI messages
   - i.e. the time between one group of three bytes and the next group of three bytes
   - Measure interval by resetting timer on first event and reading value on second event

2. To know when 800ms have elapsed since last note was received
   - LEDs are light up on each MIDI message but must be turned off after 800ms
   - Configure an interrupt to occur when Timer1 reaches a certain value
**Timer 1 Prescaler**

- How often does 16-bit timer overflow if it increments every cycle at 4MHz clock?
- Prescaler scales down the frequency of timer input to slow down the incrementing of **TCNT1** and increase time before counter overflows.
- You’ll need to use prescaler when counting the time between notes because notes may be separated by as much as 4 seconds and don’t want overflow.
- Configuration register for pre-scaler (**TCCR1B**)
Timer 1 Interrupts

- Example code (blink.c) uses interrupts when timer overflows (this is not your task for lab 2, but is similar)
- ISR (interrupt service routine)
  - Code that processor jumps to when an interrupt event occurs
  - A large number of events can trigger interrupts
  - Configure microcontroller registers to enable desired interrupt triggers

```c
#include<io.h>
#include <avr/interrupt.h>
ISR(TIMER1_OVF_vect)
{
    PORTB ^= 0xFF; //toggle PORTB values
}
int main(void)
{
    DDRB = 0xFF; //Set PORTB as output
    PORTB = 0;   //Clear PORTB bits (turn LEDs off)
    TCCR1B |= (1 << CS11); //prescaler = 8
    TCNT1 = 0;    //Clear count
    TIMSK = (1 << TOIE1); //Enable timer1 overflow interrupt(TOIE1)
    sei();        //Enable global interrupts
    while(1);      //Do nothing
}```
Timer1 Comparison Interrupt

- Your program will need interrupt 500ms after counter gets reset
- Configure an interrupt to occur when the timer reaches a specific value
- You’ll have to decide the value according to your choice of prescaler
- See datasheet for details on how to set this up
- (Similar to the code shown on previous slide)
Hardware Components

1. Communication using USART and opto-isolation circuit
   • To send and receive MIDI notes

2. Counter and Interrupts
   • To keep track of timing between notes for recording and playback

3. Storage with EEPROM
   • To store notes and timing on microcontroller

4. I/O
   • LEDs to show notes
   • Switches to choose record or playback mode
   • ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
Hardware Components

1. Communication using USART and opto-isolation circuit
   • To send and receive MIDI notes

2. Counter and Interrupts
   • To keep track of timing between notes

3. Storage with EEPROM
   • To store notes and timing on microcontroller

4. I/O
   • LEDs to show notes
   • Switches to choose record or playback mode
   • ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
EEPROM

- Non-volatile data memory (retains value after chip is powered off)
- 1k bytes of memory
- How to write to EEPROM?
  - Check control register to see whether writes are enabled (EECR)
  - Set data and address registers (EEDR, EEAR)
  - Enable write and then write data (EECR)
- How to read EEPROM?
  - Set address register (EEAR)
  - Start a read operation (EECR)
- What should be stored in EEPROM for each msg?
  - Time elapsed since last msg
  - The value of the 3 bytes of the message
  - Real system would probably compress msgs, but you don’t need to
  - Must also store in EEPROM a variable to indicate the last address in EEPROM that holds valid data. This will enable initial playback after system reset without overrunning end of valid data.
Hardware Components

1. Communication using USART and opto-isolation circuit
   • To send and receive MIDI notes

2. Counter and Interrupts
   • To keep track of timing between notes for recording and playback

3. Storage with EEPROM
   • To store notes and timing on microcontroller

4. I/O
   • LEDs to show notes
   • Switches to choose record or playback mode
   • ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
Hardware Components

1. Communication
   - To send and receive MIDI notes

2. Counter and Interrupts
   - To keep track of timing between notes for recording and playback

3. Storage with EEPROM
   - To store notes and timing on microcontroller

4. I/O
   - LEDs to show status
   - Switches to choose record or playback mode
   - ADC (Analog to Digital Converter) to digitize the amount of light falling on sensors
Digital Outputs (LEDs)

- Configure port direction to be outputs (DDRA/DDRB/etc).
- Write appropriate register (PORTA/PORTB/etc), values appear at outputs.
- Use resistor bar in series with LEDs
  - One end of resistor bar is ground.
- What happens without resistors?
  - Bright LED
  - Current limit of ATmega outputs
  - Voltage drops on ATmega outputs.

**ATmega pin properties**

**LED properties (typical)**
Digital Inputs (switches)

- Configure port directions to be inputs (e.g. DDRA for port A)
- Then, a register (e.g. PINA) will contain the input values from pins
- Read register to get pin values
- Playback and record switches connect pins to either $V_{CC}$ or gnd
- Reset uses a button and pull-up circuit (shown below)

![Digital Inputs Diagram]

- Pull-up resistor $r = 1k$
- Push-button open or shorted
- Reset $V_{PIN} \approx V_{CC}$ or $V_{PIN} \approx 0$
- ATmega input pins
Analog Input from Photocell

- Analog inputs will adjust the playback if the modify switch is on during playback
  - This allows you to “play” the GME like an instrument
  - Analog inputs are ignored if modify switch is off
- Required: one analog input must speed up and slow down playback
- Optional: use the 2nd analog input for anything you like
  - Repeat notes during playback
  - Change notes
  - Change instrument
Photocell

- Essentially a light-dependent resistor
- Why the squiggly shape?

For a good overview: https://cdn-learn.adafruit.com/assets/assets/000/010/129/original/APP_PhotocellIntroduction.pdf
Analog Input from Photocell

- Hardware setup translating photocell resistance to analog voltage
- Describe analog voltage as function of cell resistance
- This setup is called a single-ended conversion, we are measuring the analog input with respect to ground of microcontroller

CELL RESISTANCE VS. ILLUMINANCE

Analog to Digital Conversion

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}}$$

- What are min and max ADC outputs for photo sensor?
- Does bright light cause high output or low output?

$V_{REF}$ is configurable; can use external voltage or internal 2.56 V source (see datasheet: ADMUX).
How to decide which to use?
Analog to Digital Conversion

- Configure chosen pin as analog input (e.g. DDRA)
- Select the pin that should be used for ADC (ADMUX)
- Start analog-to-digital conversion by setting appropriate bit of status register (ADCSRA)
- Monitor appropriate bit of ADCSRA to wait for conversion to complete
  - Could also set an interrupt to occur when conversion finishes
- Read out the 10-bit digitized value from ADCL and ADCH after conversion is complete

- This particular ADC uses successive approximation for conversion
  - Come see me if you want to know more about it
Programming Tips
Pseudocode of a Possible Implementation

Configure timers/USART/interrupts/etc
while (1) {
    if (recording) {
        USART_Read()
        EEPROM_Write()
    }
    if (playing) {
        EEPROM_Read()
        if (modifying) {
            //modify
        }
        USART_Write()
    }
}
Suggested Timing of Program

**Record Mode**
- reset timer value
- polling for msg
- receive msg (3 bytes)
- read timer value, calculate interval
- write msg/interval to EEPROM
- polling for msg
- reset timer value
- timer reaches interval
- transmit msg (3 bytes)
- polling for msg

**Playback Mode**
- read msg/interval from EEPROM
- reset timer value
- polling timer
- timer reaches interval
- transmit msg (3 bytes)
- polling timer
Programming Tips: Manipulating bits

- Microcontroller programming requires bit-level manipulations
- Recall bitwise operators in C
  - $a = b \& c$
  - $a = b | c$
  - $a = b << c$ //shift “b” left by “c” positions
  - $a = \sim b$  //bitwise negation
- Compound assignment operators
  - $a |= b$  //bitwise or assignment
    - equivalent to $a = a | b$
  - $a &= b$  //bitwise and assignment
    - equivalent to $a = a & b$
- Bit-level manipulations that you will need
  1. Check value of a bit
  2. Set bit to 1, leaving other bits unchanged
  3. Set bit to 0, leaving other bits unchanged
Use Bitmasks to check value of a bit

- Goal: to check whether a single bit of some register is high/low
- Technique: Use “bitmask” to mask out the bits we don’t care about

- Example: check bit 3 of PINA register
  - \((1 << 3)\) is the value \(0b00001000\)
  - This value is the bitmask
  - \(\text{PINA} \& (1 << 3)\)
    - ... is \(0b00001000\) if bit 3 of PINA is 1
    - ... is \(0b00000000\) if bit 3 of PINA is 0
  - Can use “if (PINA \& (1 \ll 3))” as branching condition
    - Branch taken if bit 3 of PINA is 1 //”if(0b00001000)”
    - Branch not taken if bit 3 of PINA is 0  //”if(0b00000000)”
    - All other bits of PINA are irrelevant

- What about if(PINA \&\& (1 \ll 3))?
Use Bitmasks to check value of a bit

- Recall that register names such as `PINA` are defined as specific addresses in the memory space.
- Names are also defined for individual bits of registers:
  - e.g. `PINA3` is defined as the number 3.
  - Use it to check whether bit 3 of `PINA` has been set high.
  - if (PINA & (1 << PINA3))...
  - if (PINA & (1 << PINB3))... // what does this do?

- Having names for individual bits is very useful in control registers, since the bits there have entirely different meanings from each other.
  - if (UCSRA & (1<<RXC)) {
    // USART is done receiving
  }
Use Bitmasks to set bits

- Goal: set bits to 1 without changing other bits of register

- Example: set bit 3 of PORTB:
  - PORTB |= (1 << PORTB3)
  - PORTB = PORTB | (1 << PORTB3)
  - PORTB = PORTB | (1 << 3)
  - PORTB = PORTB | (0b00001000)

- Can combine masks using bitwise or
  - PORTB |= (1 << PORTB3) | (1 << PORTB2)

- What does this do?
  - PORTB |= (0 << PORTA3)
Use Bitmasks to clear bits

- Goal: clear bits to 0 without changing other bits of register

- Example: clear bit 3 of port B:
  - `PORTB &= ~(1 << PORTB3)`
  - `PORTB = PORTB & ~(1 << PORTB3)`
  - `PORTB = PORTB & ~(1 << 3)`
  - `PORTB = PORTB & ~(0b00001000)`
  - `PORTB = PORTB & 0b11110111`

- Can you clear some bits of register and set other bits of same register using a single bitmask operation??
Programming Tips: Appropriate Datatypes

- 8-bit processor
- int is 16 bits
- long is 32 bits

- May want to use “uint8_t” for 8-bit integer values
- May want to use “uint16_t” for 16-bit integer values
  - e.g. ADC outputs, timer values

- How are 16-bit types handled on 8-bit processors?
Suggestion for Building Project Incrementally

- Try example code (blink.c) from course website
  - Verify that you can program microcontroller and step through program as shown in the AVR studio demo video

- Write simple skeleton of your code
  - Verify that playback/record switches lead to the expected points in code

- Verify that USART receives MIDI correctly
  - Use watch window or light up LEDs to show received notes
  - Use oscilloscope to check that outputs of opto-coupler look right

- Verify that USART correctly sends MIDI to PC
  - Start by sending hard-coded constants
  - Use scope to verify that bit timing is correct

- Verify that ADC works correctly
  - Use watch windows or light up LEDs to see ADC output
  - Check whether ADC value matches applied voltage

- Verify functions for reading and writing EEPROM
  - Could write a loop to read/write values

- Test your interrupt routine to verify 800ms timing
General Advice

- Don’t build complete system without first testing components:
- Consult the Atmega32 datasheet
  - Pin layouts
  - Code examples
  - Everything you need to manipulate the microcontroller for project is there!
- Use the AVR studio debugger
  - Check the value of pins and registers
  - Single step through code to find source of problems
- Debug with logic analyzer and oscilloscope
  - Check frequency of USART outputs on scope
- Use internet resources
  - avrfreaks.net is a good resource for examples
- Expect that you will face issues: start early and get help early if stuck
  - E.g. if you break a part on the night the project is due, you may not be able to get a replacement before deadline
All the software needed for the lab is available on the lab computers in directory:

```
c:\!_ECE_353_2018_Lab_Software
```

There is an excel spread sheet with the list of software, version numbers and links to the source so you can install it on your own personal computers

**Tool flow for Putting code onto ATMega32**
WinAVR

- WinAVR is a set of development tools for Atmel AVR RISC microprocessors
- Programs written in C, compiled with GCC and avr-libc
- Open source, can be obtained at: winavr.sourceforge.net
Build Process

- Board Assembly
- C Programming
  - WinAVR
  - AVR Studio version 4 (IDE)
  - JTAG Programmer
- Testing
  - In-Circuit Emulator
  - MIDI-OX
  - Saleae Logic Analyzer
    - Instructions on saleae.com
  - Oscilloscope in lab
AVR Studio Demo: Getting Started

- Open AVR Studio
- Create new project
- Select AVR GCC for the project type
- Type in a project title and choose project directory if necessary
- Click next and select the debugging platform (JTAG ICE) used to program the MCU as well as the MCU that you are using (ATmega32)
- Finish
AVR Studio Demo: Configure Project

1. Choose the appropriate clock frequency for your part
2. Under custom options, should see WinAVR for avr-gcc and make
3. May need to add include directories and libraries if not seeing any external dependency files listed in project after building in next step
AVR Studio Demo: Program Device

Write code, then then…

1. Build your project

2. Connect to the MCU, and then
   A. Make sure fuse selects ext. clock
   B. Erase the device
   C. Load to flash memory the .hex file created in your project directory (location of hex file is set through project configuration window)

Now program will start executing!
In Circuit Emulator

- Use ICE to debug code as it runs on ATmega32
- Critical for completing project and diagnosing bugs
- Click on triangle to start debug
- Use controls to move through code
  - Step through code one instruction at a time
  - Set breakpoints in code
- Watch windows to see values of variables changing as program runs
ICE Watch Windows

Note: this was compiled with option -o0 to turn off compiler optimization (otherwise var x gets removed by compiler and can’t be added to watch window)
MIDI-OX Software

- Open MIDI device in options menu
  - Select Yamaha UX16 (if not available, need to install driver)
- Click on the keyboard button, press key on computer keyboard
- Hear sound, and key appears in output monitor
- This note was sent to the GME (can see it on scope)
  - Note on is a 3-byte message
  - Note off is a 3-byte message (shortly after the first)
- Note will appear on input monitor later when played back from GME