Using the ATmega328P/PB chips with Microchip Studio (in a non-Arduino world)

Index

Overview

The ATmega328P/PB chips have many features similar to those you have explored with the 9S12X chip. These include:

- 16-bit timer with output compare, with interrupts
- UART (can be configured to operate just like the SCI)
- I2C
- Interrupts (pins and modules)
- 10-bit A/D
- GPIO
- PWM

There are many other features, but the ATmega328P/PB is a useful chip to build a design around as it:

- Is cheap (when we aren't experiencing a global chip shortage)
- Comes in DIP (P version) and QFP (PB version) formats
- Requires minimal external parts
- May be programmed with a reliable and cheap programmer

This chip is popular with Arduino designs, but you will be using the chip without Arduino libraries. You may still use Arduino compatible peripherals, but you will be responsible for writing the code that operates the chip, and those peripherals.

These devices contain an internal 8MHz oscillator that is fuse-configured to operate the device at 1MHz by default. This is the out-of-the-box configuration, and provides a reliable starting point for the device. Your design will include a 16MHz** external crystal that will permit the device, with the correct fuse options, to run as fast as 16MHz; 16x the original configuration. Note: running the device this fast is not necessarily required for your project, nor will it be stable at all supply voltages at this rate, but as this is a PCB design course including these parts will provide some artwork that has specific constraints.

In order to introduce you to PCB design and software development in this realm, one or more projects will be examined to provide whole-project scope. You will be required to come up with your own unique project of a similar scope, using similar technology.

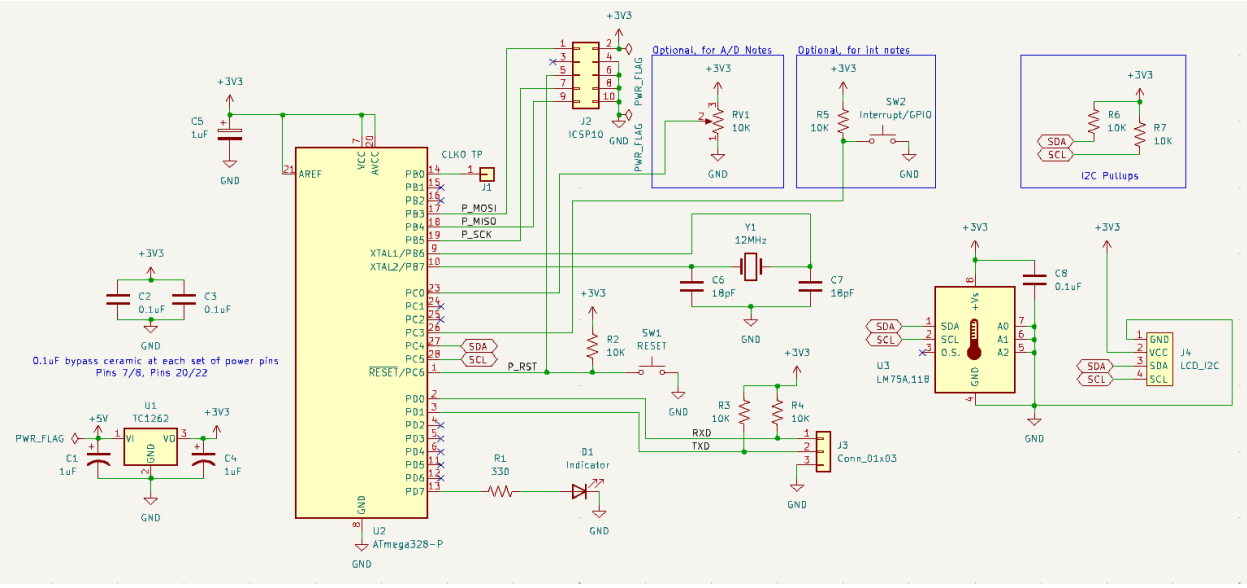
You will initially build a breadboard version of your project using the DIP version of the device, and use this to develop the necessary software for your project, and prove out the design. You will then design a PCB version, using the QFP version of the chip*. You will order PCBs for this matured design, create a Bill of Materials (BOM), order parts not available from your instructor, assemble and program a prototype, then present (demonstrate) your final project.

***If availability makes this possible, otherwise your final PCB will feature a DIP 328.**

****5V operation, max 16MHz, 3.3V operation, max 12MHz**

Initial breadboard design

The base design for your prototype will be a breadboard version of the DIP version of the ATmega328P. You may purchase these devices from J105. The following circuit diagram shows what is required as a minimum, for 3.3V operation, externally sourced:



Version 1.0 – Basic ATmega328P (DIP) Breadboard Circuit

This circuit contains:

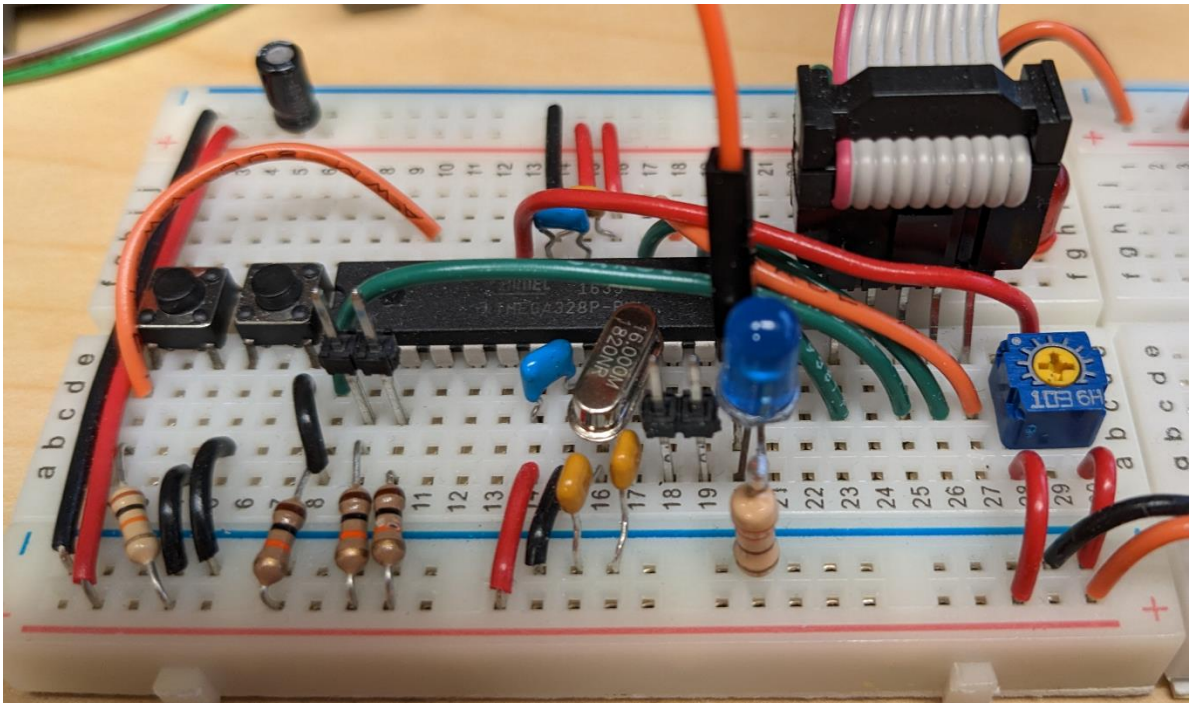
- a reset switch with pull-up resistor to correctly manage the level of the RESET line
- decoupling/filtering capacitors for the chip power supply (0.1uF cap for *each set* of power pins)
- a 16MHz/12MHz crystal with matching capacitors for external crystal usage
- a test point to validate that the chip is correctly using the crystal circuit (optional)
- an ICSP10 interface header for connecting to the device programmer
- a header for the UART TX and RX pins to use for debugging purposes (optional, but not really)
- an optional LED indicator to use for debugging purposes

NOTE: Pin 22 needs to be tied to ground. Some library parts hide redundant pins in the schematic (but will appear when you make the PCB). The schematic tool has a show/hide button for hidden pins. If you create your own symbols, make all the pins visible! Your instructor will comment on this!

You will be provided the crystal (possibly in your year two kit, or in the lab), the associated capacitors, and the reset switch. All other components you should have in your kit, or you may purchase them from J105.

Note: once you have validated that you have programmatic control over the device, you are free to reassign the **PB0** and **PD7** pins to the needs of your project.

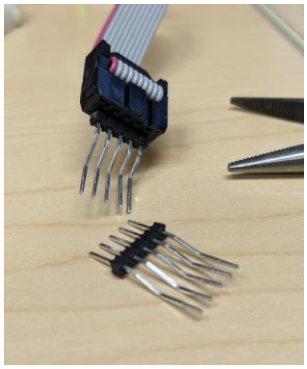
When assembled, the final breadboard should look *something* like this:



Breadboard Version of Starter Circuit (ATmega328P)

Tidy wiring and short jumpers become increasingly important with higher frequencies. You'll note that the crystal and its two capacitors are as close as possible to their associated pins – moving the crystal away from this location could well result in oscillator failure due to extra capacitance and resistance, and the system will be more susceptible to, and will also generate, radio-frequency noise.

You will need to do some creative mashing of some wire-wrap tail header pins to create the ICSP10 header, as it needs to span the breadboard device gap, but still form up to a standard 2.54 mm spacing 2x5 header connector. Your instructor will provide a demonstration on how to do this.

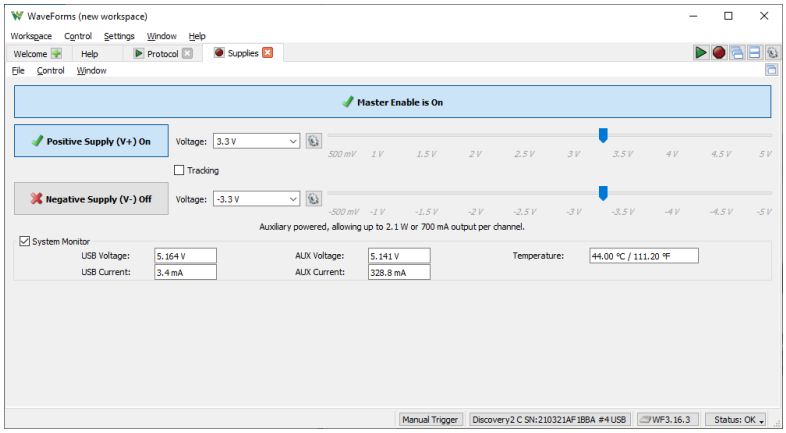


Header strip "adjustment" to form connector

Do not perform lead bending with the header strip plugged into the programmer cable – this could damage the programmer cable.

Note: it is difficult to remove the ICSP10 header connector (device programmer end) from the header. Because of this, you may want to consider purchasing your own programmer ($29.48 from Mouser in term 1232). This will also permit you to develop code at home. Programmers will be made available in the lab, during lab time, if you don't want to purchase one.

Sensor/peripheral choices will drive the decision for system voltage selection. For the now, you may power your board at 3.3V or 5V with a bench power supply, the programmer, or with your trusty AD2:



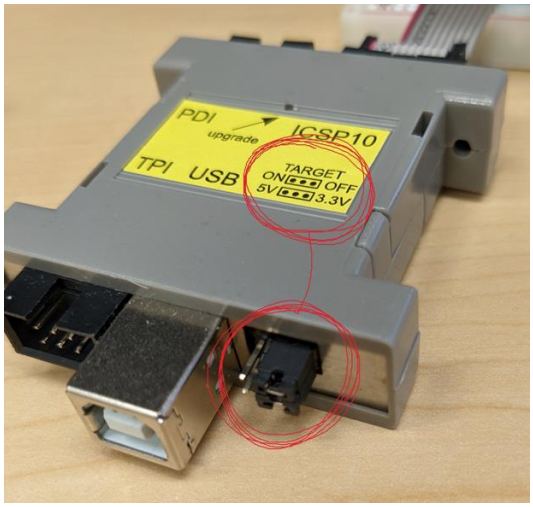
Waveforms 'Supplies' Tab with power configuration for ATmega328P Breadboard

Your device programmer is capable of supplying power to the target (the chip, and your circuit). To use the programmer as a power source, put the upper target jumper in the ON position; otherwise, put it in the OFF position to allow you to use your own power supply.

Check the programmer, regardless, before you connect it to your circuit to ensure that it is in the right power configuration. **Do this every time you borrow a programmer!**

You certainly wouldn't want to have it powered up, in 5V mode if you had 3.3V devices in your circuit. Note: the programmer can only supply so much current to your circuit!

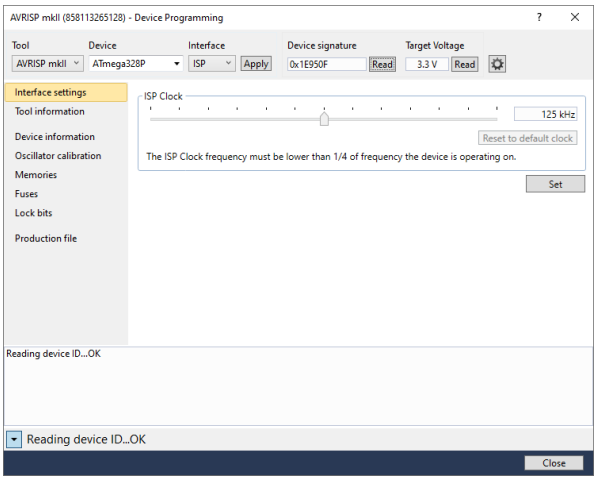
The red stripe on the programmer's cable indicates the pin 1 end.



Olimex AVR-ISP-MK2 Programmer with Target Option Jumpers

Once you are satisfied that your circuit is correct, you are ready to test the programming connection with Microchip Studio. Microchip Studio (MCS) is installed in the labs, but you will need to download it at home, should you wish to do development at home. A link is provided for you in Moodle.

MCS contains a device programming tool, found under the 'Tools' menu that can independently test the device:

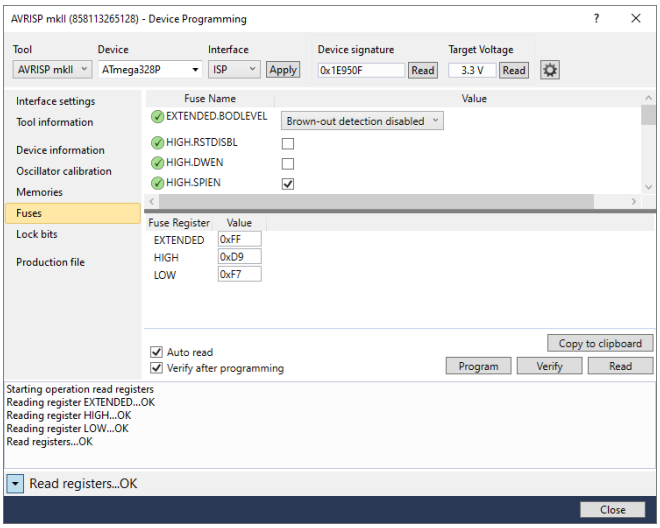


Device Programming Dialog in Microchip Studio

When the device programmer is correctly connected to your circuit, the circuit is correctly powered, and the programmer is plugged into a USB port on the PC, you should be able to read the device signature from the ATmega328P chip.

If you are unable to do this, power down the circuit and have your instructor inspect your work before you continue.

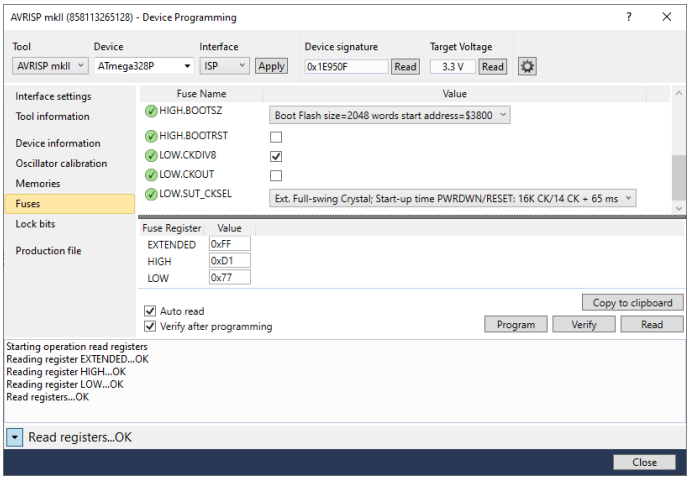
In addition to reading the device signature, the device programming tool can also read the device configuration. This is presented as a series of 'fuse' options that control the major configuration options for the device:



'Fuses' page of the device programming tool

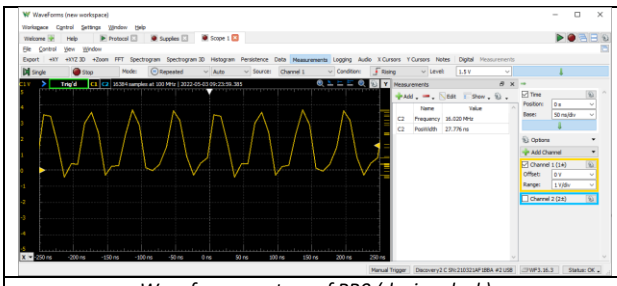
The device ships from the manufacturer with the internal RC oscillator selected, with a divide by 8 option turned on. When you are ready, you can take a deep breath and program the fuses to use the external crystal, and *optionally* remove the divide by 8 option, *if running the device at 5V*. This will cause the chip to run at either 2MHz or 16MHz, depending on the divide by 8 option. **3.3V designs should not operate above 12MHz.** You may also turn on the option LOW.CKOUT to have the clock expressed on the **PB0** pin (**J1** in the sample schematic above). Note: there are **a lot** of options for the external crystal; ensure you pick the correct one!

Once you have selected the appropriate Fuse settings, hit 'Program' to send the settings to your chip. 'Verify' that the fuse settings are correct – if all went well, you should get a copy of the settings you chose and no error messages. As a final check, you can also 'Read' the fuse settings, with the same results as 'Verify'. Reading the device signature, at any time, is also a good way to ensure that you can communicate with the chip.

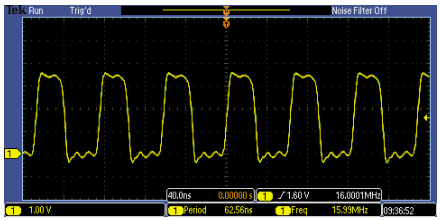


Fuse options for initial circuit tests

You may use a scope or your AD2 to measure the device clock on **PB0** after programming the fuses with these options, although 16MHz is at the upper limit of what your AD2 can capture (w/configuration 2):



Waveforms capture of PB0 (device clock)



MSO2014B capture of PB0 (device clock)

Measuring the device clock will be useful if:

- you are having trouble seeing device activity
- you are using an odd crystal (something at low frequency)
- you are attempting to characterize the internal RC oscillator

Initial Programming

To create projects for the ATmega328P or ATmega328PB devices, there are a number of possible IDEs to select from. In our labs, we will use Microchip Studio. The simplest way to create code is to create a new *GCC C Executable Project*. There are other techniques for creating projects, and some that automate the process of code generation. You may experiment with these other methods, but the notes for this course will focus on simple code generation. Libraries of code for common tasks will be provided on Moodle. *Avoid techniques/tools that inject code you don't understand.*

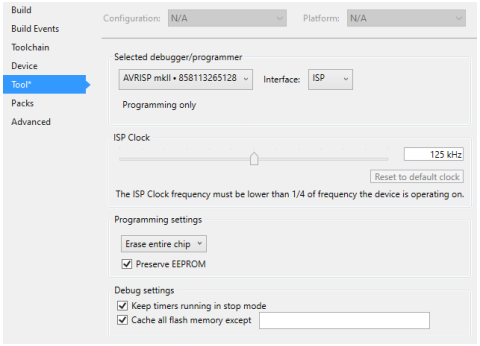
The starter code for this type of project is a little thin. The sample below is a slightly more fleshed-out version that includes the necessary code to toggle the LED shown in the sample schematic at 1ms intervals. As in CMPE1250/2250, you should build up templates/libraries to manage your code.

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)

int main(void)
{
  // one-time initialization section

  // sample schematic shows a LED on pin PD7
  // set pin 7 of port D to be an output
  // note: this is just like with our micro!
  //  but the derivative file is a little different...
  DDRD |= 1 << PORTD7; // make portd pin 7 an output (PD7)
  // same as DDRD |= 0b10000000;

  // main program loop - don't exit
  while(1)
  {
    //_delay_ms(1);
    //PORTD |= 0b10000000;      // PD7 on
    //_delay_ms(1);
    //PORTD &= ~(0b10000000);   // PD7 off

    // same as
    _delay_ms(1);
    PIND = 0b10000000;         // toggle PD7
  }
}
```

Like our micro, unassigned pins on the ATmega328P may be used for GPIO. There is a data direction register for each port that works exactly the same way as you have seen previously. Unlike our micro, every pin on this chip is capable of generating an interrupt (more on this later), and each pin may be configured with an internal pull-up resistor (set by setting/clearing the bit in the port register when the pin is an output), and there is a pin toggling register!

If you intend to do any GPIO operations with your project, you should read the relevant section(s) of the datasheet for the ATmega328.

If the device is connected to a programmer, you may now hit F5 to compile, build, and program the code on the device. Initially, you will need to pick a programmer, and the IDE will be happy to tell you that you don't have one set up:



Selection of Device Programmer in Microchip Studio

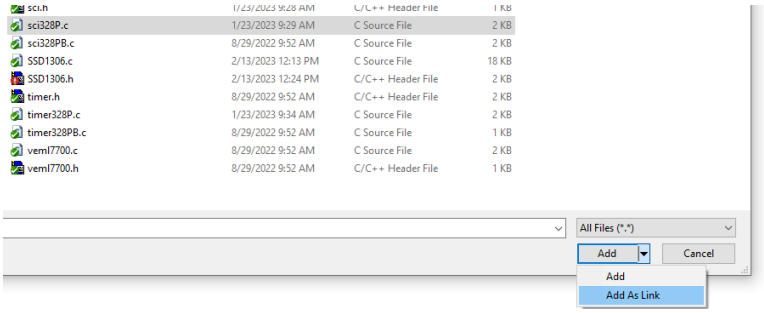
You should be able to select the 'Selected debugger/programmer' from the dropdown – your programmer should appear in the list if it is plugged into a USB port.

After this is set up, hitting F5 should program the device. Note: this is typically astonishingly fast compared to programming the 9S12 chip in micro; in fact, you may become confused initially, unsure if anything actually happened.

With the sample code above, we would expect to see a 500Hz 50% duty square wave on the LED pin. The LED will be blinking too quickly to notice, and will appear to simply be on.

Because you won't have sophisticated display devices (at least initially), you will find that using the device UART and your AD2 as a handy terminal can be instrumental in providing debugging information. A starter library for the UART (similar to the SCI from the micro) will be provided in GitHub. You can use this library to send/receive data with the AD2. You may connect the digital I/O pins to the TXD and RXD pins from the sample circuit and configure the AD2 protocol analyzer to send/receive data.

You will need to add the provided library files (implementation files) to your project folder. Sadly, when you include library files in Microchip Studio, you need to be careful how you do it. For example, to add a file to your project, you would right-click on the ***project*** in the solution explorer, and select 'add existing item'. Now for the counter-intuitive part. When you select the implementation file to include, you will need to take the 'Add as link' option or the file will be copied.

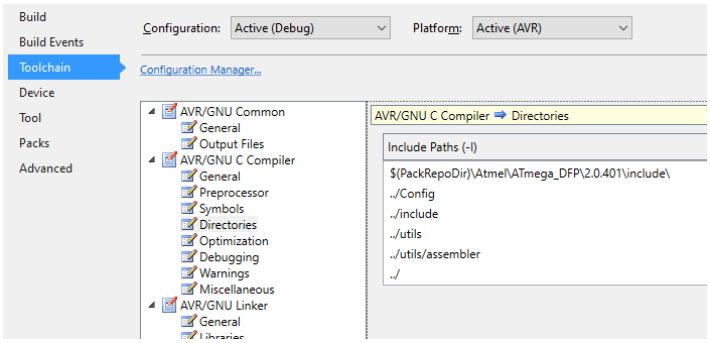


The not-so-intuitive mechanism for adding existing items as links in Microchip Studio

You want links, so that as you update the libraries, the changes will apply to all links.

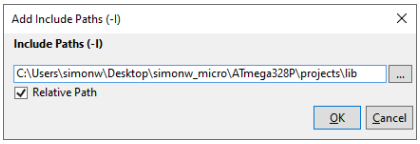
Note: there are implementation files for each derivative, so pick the one that is for your P/PB device. Also, you do not need to add the headers to project, but you can if you like.

Microchip Studio will have no idea, oddly, where to find these files, so you will need to update the search path to find them. This is done in the 'Project' / '<your project> Properties' option, under the 'Toolchain' tab, under the 'AVR/GNU C Compiler' option, under the 'Directories' item:



Adding a new search path in Microchip Studio

Pick the 'Add Item' button on the right, and navigate to your library folder. Ensure the 'Relative Path' option is checked, and click OK:



This should add a new path entry under '../../../lib', or whatever your folder is called.

Finally, done. The good news is, you only need to do this once per project.

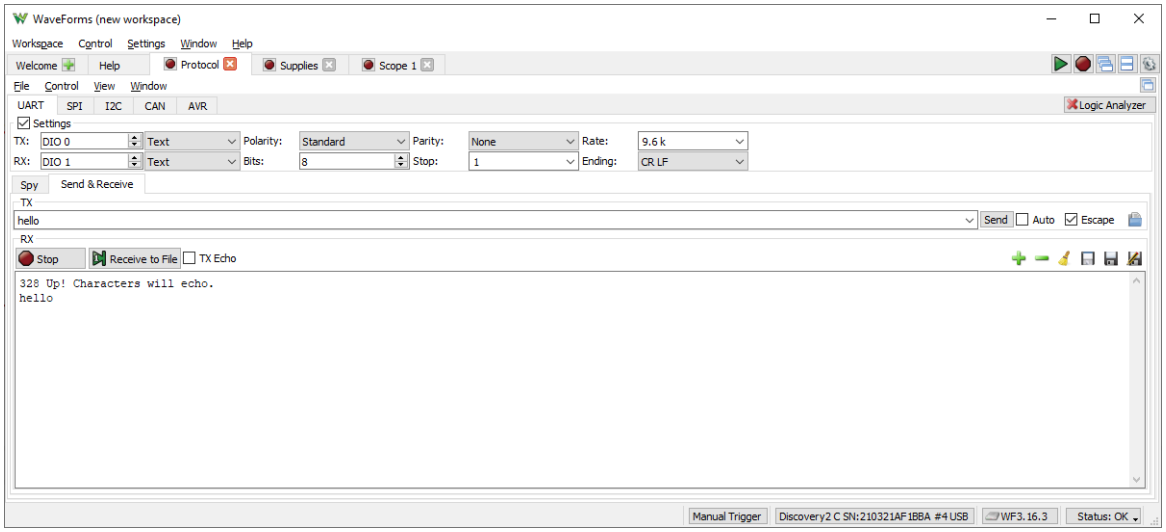
If, by mistake, you've 'Added' a library file instead of linking to it, you'll need to go to the project folder and remove (delete) the copy of the library file you added. If you don't remove the copy, MCS will preferentially select the copy instead of any link, so changes you've made in your master library won't appear in your project.

Once you've removed the copy you may add the link you intended to originally add.

On the code side, things will look remarkably similar to what you are familiar with in micro:

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include "sci.h"

int main(void)
{
  // one-time initialization section

  // set up serial for 2MHz bus, 9600 BAUD, no interrupts
  SCI0_Init(F_CPU, 9600, 0);

  // send a welcome message
  SCI0_TxString("Hello from the ATmega328P!");

  // make portd pin 7 an output (PD7)
  DDRD |= 0b10000000;

  // main program loop - don't exit
  while(1)
  {
    // toggle PD7 at 1ms intervals
    _delay_ms(1);
    PIND = 0b10000000;          // toggle PD7
  }
}
```

To check your communication, use the 'Send & Receive' mode in the AD2 Waveforms Protocol component for UART. Make sure the protocol settings for the AD2 match the settings of the ATmega328P code.



Sending and Receiving characters with WaveForms and the AD2

Your instructor is certainly going to demonstrate all of this, so don't panic.

NOTE: Don't attempt to use 9600 BAUD when the device is running at 1MHz – the BAUD rate won't work out very well, and you won't be able to communicate with the chip.

Coordinating Activities with the Timer

Many coding activities need to be controlled with specific timing, or at the very least, not in a CPU-bound loop.

The ATmega328 chip has a timer that is not entirely unlike what you have worked with on the 9S12X chip. Once again, a starter library will be provided that is able to set up an output compare operation, with interrupt, that may be used to coordinate program tasks that require it.

Interrupts are a little simpler on this platform to code. The ISR may be used to increment a global counter that will act as a reference for timing operations; this includes time projection and modulus operations:

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include "sci.h"
#include "timer.h"

// global variables

// constant for timer output compare offset, init and ISR rearm
const unsigned int _Timer_OC_Offset = 250; // 1 / (2000000 / 8 / 250) = 1ms (prescale 8)

// global counter for timer ISR, used as reference to coordinate activities
volatile unsigned int _Ticks = 0;

int main(void)
{
  // constant for how many ticks (for 1ms in this case)
  //  to schedule the blink event of the indicator LED
  const unsigned int cuiBlinkEventCount = 10;

  // variable to hold projected time for next toggle
  // note: using a variable to keep track of projected event times
  //  is superior to using modulus, as modulus could miss an event
  //  if polling was too slow relative to the counter
  //  or the natural type has a non-even factor boundary (65535 % 10 is a problem)
  // projection would not (but the event could be delayed until the next poll)
  // unless the code is using a wait state, or can guarantee execution in one tick time
  unsigned int uiBlinkEventNext = cuiBlinkEventCount;

  // one-time initialization section

  // set up serial for 2MHz bus, 9600 BAUD, no interrupts
  SCI0_Init(F_CPU, 9600, 0);

  // welcome message, so we know it booted OK
  SCI0_TxString("\n328 Up! Now with timer interrupts!\n");

  // bring up the timer, requires ISR!
  Timer_Init(Timer_Prescale_8, _Timer_OC_Offset); // 1ms intervals

  // enable sleep mode, for idle, sort of similar to WAI on 9S12X (13.2)
  //SMCR = 0b00000001;
  sleep_enable();

  // set the global interrupt flag (enable interrupts)
  // this is backwards from the 9S12
  sei();
```

```
  // make portd pin 7 an output (PD7)
  DDRD |= 0b10000000;

  // main program loop - don't exit
  while(1)
  {
    // storage for transient character received/sent
    unsigned char crx;

    // go idle!
    // note: this implies that the main loop can execute in less
    //  than 1ms, or, is tolerant of being interrupted if it can't
    sleep_cpu();

    // blink the indicator LED every 10ms
    // note: using subtraction with the unsigned type will
    //  ensure that boundary issues are not a factor
    // are we past the scheduled event?
    if (uiBlinkEventNext - _Ticks > cuiBlinkEventCount)
    {
      // yep, so rearm for next event
      uiBlinkEventNext += cuiBlinkEventCount; // rearm for next blink event from last time

      // do the event action
      PIND = 0b10000000;         // toggle PD7
    }

    // this code is now constrained by the 1ms sleep action (probably OK for 9600 BAUD)
    // if this needed to occur at a higher rate, well, we would use SCI interrupts
    //  or use a faster tick rate

    // if we got a byte...
    if (!SCI0_RxByte(&crx))
    {
      // and echo the received character back
      SCI0_BSend(crx);
    }
  }
}

// output compare A interrupt
ISR(TIMER1_COMPA_vect)
{
  // rearm the output compare operation
  OCR1A += _Timer_OC_Offset; // 1ms intervals

  // up the global tick count
  ++_Ticks;
}
```

Interrupts with the UART, and program loop control

The following code sample demonstrates how interrupts may be used with the UART to receive characters. Consideration must be made on how received characters are processed; insufficient responsiveness will still cause problems. You considered this in some circumstances in the micro courses, and the situation is the same with any code, on any platform. The comments in the code sample explain some of the things you may wish to consider when planning your design.

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "sci.h"
#include "timer.h"

// global variables

// constant for timer output compare offset, init and ISR rearm
const unsigned int _Timer_OC_Offset = 250; // 1 / (2000000 / 8 / 250) = 1ms (prescale 8)

// global counter for timer ISR, used as reference to coordinate activities
volatile unsigned int _Ticks = 0;

// byte received by UART RX interrupt
volatile unsigned char _RXByte = 0;

// flag (counter) that indicates if a byte has been received by the interrupt
//  this will normally be 1 when a byte is received, but the ISR increases
//  it by one on each reception. The owning code should clear it, and possibly
//  look for >1 to identify overrun.
// realistically, the ISR should be placing received values in a queue
//  or circular buffer
volatile int _bRXFlag = 0;

// wall time counter to show up time (normalized)
//  on the UART
// added to timer interrupt (not tolerant of main loop latency)
volatile unsigned long _ulWallTime = 0;

// local prototypes
void TimeFromMS (char * buff, unsigned long ticks);


int main(void)
{
  // constant for how many ticks (for 1ms in this case)
  //  to schedule the blink event of the indicator LED
  const unsigned int cuiBlinkEventCount = 10;

  // variable to hold projected time for next toggle
  // note: using a variable to keep track of projected event times
  //  is superior to using modulus, as modulus could miss an event
  //  if polling was too slow relative to the counter
  //  or the natural type has a non-even factor boundary (65535 % 10 is a problem)
  // projection would not (but the event could be delayed until the next poll)
  // unless the code is using a wait state, or can guarantee execution in one tick time
  unsigned int uiBlinkEventNext = cuiBlinkEventCount;

  // one-time initialization section
```

```
// set up serial for 2MHz bus, 9600 BAUD, interrupts please!
SCI0_Init(F_CPU, 9600, 1);

// welcome message, so we know it booted OK
SCI0_TxString("\n328 Up! Now with timer and UART interrupts!\n");

// bring up the timer, requires ISR!
Timer_Init(Timer_Prescale_8, _Timer_OC_Offset); // 1ms intervals

// enable sleep mode, for idle, sort of similar to WAI on 9S12X (13.2)
//SMCR = 0b00000001;
sleep_enable();

// set the global interrupt flag (enable interrupts)
// this is backwards from the 9S12
sei();

// make portd pin 7 an output (PD7)
DDRD |= 0b10000000;

// main program loop - don't exit
while(1)
{
  // go idle!
  // note: this implies that the main loop can execute in less
  //  than 1ms, or, is tolerant of being interrupted if it can't
  sleep_cpu();

  // blink the indicator LED every 10ms
  // note: using subtraction with the unsigned type will
  //  ensure that boundary issues are not a factor
  // are we past the scheduled event?
  if (uiBlinkEventNext - _Ticks > cuiBlinkEventCount)
  {
    // yep, so rearm for next event
    uiBlinkEventNext += cuiBlinkEventCount; // rearm for next blink event from last time

    // do the event action
    PIND = 0b10000000;         // toggle PD7
  }

  // this is a lazy 10s update, where wall time is being strictly
  //  updated by the ISR, but updates can be missed
  // updates are only for display and don't have anything
  //  to do with accuracy of the wall time
  // this illustrates boundary issues where updates
  //  will occur at 0, 10000, 20000, 30000, 40000, 50000, 60000, and wrap
  //  leaving the 5535 in 65535 without contributing an update
  // time is accurate, update frequency is not aligned rationally with time
  if (!(_Ticks % 10000)) // this would depend on sleep per loop
  {
    char buff[50] = { 0 };
    TimeFromMS (buff, _ulWallTime);
    SCI0_TxString (buff);
    SCI0_TxString ("\n");
  }

  // this code is now constrained by the 1ms sleep action (probably OK for 9600 BAUD)
  // if this needed to occur at a higher rate, well, we would use interrupts
  //  wouldn't we... or use a faster tick rate
```

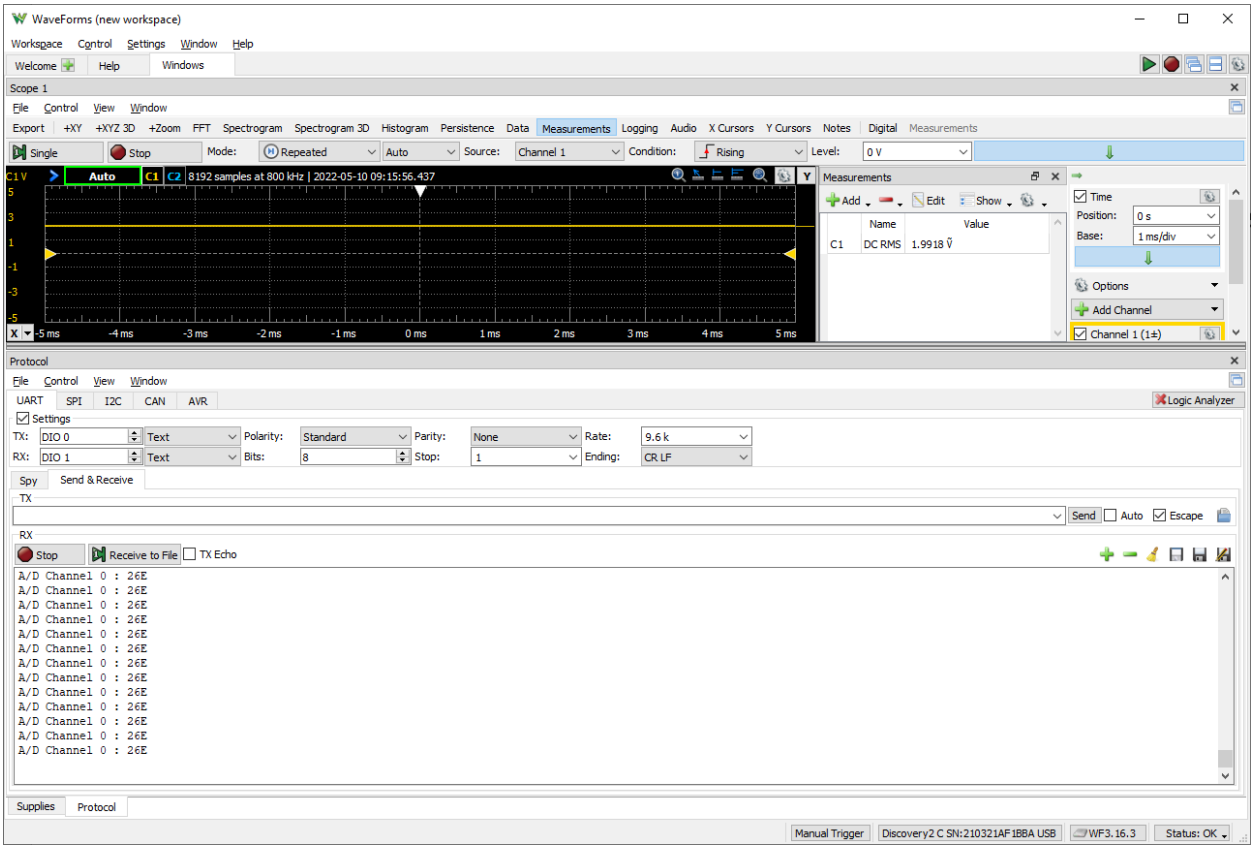
```
    // if we got a byte via interrupt...
    if (_bRXFlag)
    {
      // clear flag (could do overrun test here)
      _bRXFlag = 0;

      // and echo the received character back to prove round-trip
      SCI0_BSend(_RXByte);
    }
  }
}

// support function to turn ms ticks into wall time
void TimeFromMS (char * buff, unsigned long ticks)
{
  (void)sprintf (buff, "%2.2d:%2.2d:%2.2d:%2.2d.%1.1d", // DD:HH:MM:SS:T
  (unsigned int)(ticks / (1000l * 60 * 60 * 24)),       // DD
  (char)((ticks / (1000l * 60 * 60)) % 24),             // HH
  (char)((ticks / (1000l * 60)) % 60),                  // MM
  (char)((ticks / 1000l) % 60),                         // SS
  (char)(ticks / 100 % 10)                              // H
  );
}

// output compare A interrupt
ISR(TIMER1_COMPA_vect)
{
  // rearm the output compare operation
  OCR1A += _Timer_OC_Offset; // 1ms intervals

  // up the global tick count
  ++_Ticks;

  // up the wall time ms count
  ++_ulWallTime;
}

ISR (USART_RX_vect)
{
  // indicate byte has been received (and can detect overrun too)
  ++_bRXFlag;

  // pull the byte (clears the interrupt too)
  _RXByte = UDR0;
}
```

Note: This code now overruns the main loop, meaning the main loop is no longer able to execute in 1ms when the time string is sent to the UART. The overrun will cause the output waveform on the LED pin to hiccup, as the timing is disturbed. If you *require* critical timing, you would not perform waveform generation in the main loop, as you can't, generally, guarantee that the main loop is always executing within the timer interrupt interval. It would be better to do this with PWM, or in the timer ISR, even though lengthy ISRs are something to avoid.

Adding A/D

The ATmega328P chip has a 10-bit A/D module that is suitable for general-purpose measuring of both external voltages, and some interesting internal ones as well.

In the sample circuit, the reference voltage (AREF) is tied to the positive supply (3.3 V), so the A/D range is 3.3 V / 2^{10}, or approximately 3.223 mV/step. You could (and certainly should) put more effort into creating the A/D circuitry if you want additional stability, useful step size, and noise rejection. Anything beyond basic A/D operation is beyond the scope of this course, but your instructor can assist with this, as long as Ross is your instructor.



*A/D Measurement with 2V input, with 3.3V Reference (0x26E/0x400 * 3.3 = 2.0V)*

Once again, a base library will be provided that you can modify to suit your needs. The following sample code shows how an A/D reading could be taken and displayed on the UART:

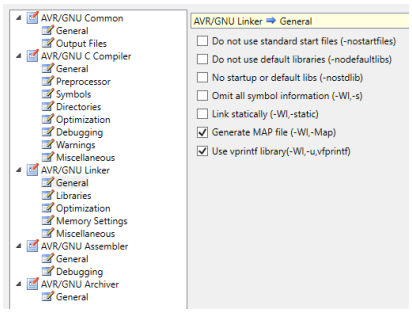
```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "sci.h"
#include "timer.h"
#include "atd.h"

// global variables

// constant for timer output compare offset, init and ISR rearm
const unsigned int _Timer_OC_Offset = 250; // 1 / (2000000 / 8 / 250) = 1ms (prescale 8)

// global counter for timer ISR, used as reference to coordinate activities
volatile unsigned int _Ticks = 0;

// byte received by UART RX interrupt
volatile unsigned char _RXByte = 0;

// flag (counter) that indicates if a byte has been received by the interrupt
//  this will normally be 1 when a byte is received, but the ISR increases
//  it by one on each reception. The owning code should clear it, and possibly
//  look for >1 to identify overrun.
// realistically, the ISR should be placing received values in a queue
//  or circular buffer
volatile int _bRXFlag = 0;

int main(void)
{
  // variable for managing the A/D update
  const unsigned int cuiAtoDEventCount = 500; // every 1/2 second
  unsigned int uiAtoDEventNext = cuiAtoDEventCount;

  // one-time initialization section

  // set up serial for 2MHz bus, 9600 BAUD, interrupts please!
  SCI0_Init(F_CPU, 9600, 1);

  // welcome message, so we know it booted OK
  SCI0_TxString("\n328 Up! Now with A/D!\n");

  // bring up the timer, requires ISR!
  Timer_Init(Timer_Prescale_8, _Timer_OC_Offset); // 1ms intervals

  // enable sleep mode, for idle, sort of similar to WAI on 9S12X (13.2)
  sleep_enable();

  AtoD_Init(AtoD_Channel_0);

  // set the global interrupt flag (enable interrupts)
  // this is backwards from the 9S12
  sei();

  // make portd pin 7 an output (PD7)
  DDRD |= 0b10000000;
```

```
  // main program loop - don't exit
  while(1)
  {
    // go idle!
    sleep_cpu();

    // are we past the scheduled event?
    if (uiAtoDEventNext - _Ticks > cuiAtoDEventCount)
    {
      uiAtoDEventNext += cuiAtoDEventCount; // rearm

      unsigned char AD_low = ADCL;       // must be read first
      unsigned char AD_high = ADCH;

      char buff [50] = { 0 };
      (void)sprintf (buff, "A/D Channel 0 : %3.3X\r\n", AD_low + AD_high * 256);
      SCI0_TxString(buff);
    }

    // if we got a byte via interrupt...
    if (_bRXFlag)
    {
      // clear flag (could do overrun test here)
      _bRXFlag = 0;

      // and echo the received character back to prove round-trip
      SCI0_BSend(_RXByte);
    }
  }
}

// output compare A interrupt
ISR(TIMER1_COMPA_vect)
{
  // rearm the output compare operation
  OCR1A += _Timer_OC_Offset; // 1ms intervals

  // up the global tick count
  ++_Ticks;
}

ISR (USART_RX_vect)
{
  // indicate byte has been received (and can detect overrun too)
  ++_bRXFlag;

  // pull the byte (clears the interrupt too)
  _RXByte = UDR0;
}
```

Adding Floating-point Output

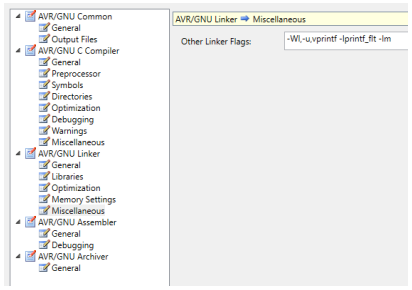
By default, the projects you create in Microchip Studio will not link the necessary libraries to perform floating-point output. The argument is that adding the additional overhead is undesirable if it won't be used, so it's not included by default.

To turn this on, there are a couple of changes that you need to make.



Turn on the 'Use vprintf library' option in the "project / properties" page.

Next, (somewhat redundantly) add the following text to the "Other Linker Flags" in the "Miscellaneous" option:



For cut-and-paste benefit, that line is: **`-Wl,-u,vprintf -lprintf_flt -lm`**

This will permit standard **`%f`** format specifier use with **`sprintf`**, so that you can display floating-point values:

```
unsigned char AD_low = ADCL;     // must be read first
unsigned char AD_high = ADCH;
unsigned int AD_Raw = AD_low + AD_high * 256;

char buff [50] = { 0 };
(void)sprintf (buff, "A/D Channel 0 : %3.3X (%0.2fV)\r\n", AD_Raw, AD_Raw / 1024.0 * 3.3);
SCI0_TxString(buff);
```

```
A/D Channel 0 : 0AB (0.55V)
A/D Channel 0 : 295 (2.13V)
A/D Channel 0 : 09A (0.50V)
A/D Channel 0 : 22D (1.80V)
A/D Channel 0 : 38A (2.92V)
A/D Channel 0 : 076 (0.38V)
A/D Channel 0 : 044 (0.22V)
A/D Channel 0 : 391 (2.94V)
A/D Channel 0 : 3FF (3.30V)
A/D Channel 0 : 29E (2.16V)
A/D Channel 0 : 0DD (0.71V)
A/D Channel 0 : 00E (0.05V)
```

The Timers and PWM

There are three timers on the 328 chip; one 16-bit and two 8-bit. Timer/Counter1 was already discussed in the timer section, and is 16-bit. Each timer is able to operate in PWM mode. When used for PWM, the timers are very configurable, and have multiple modes of operation. A complete examination of all PWM configurations is beyond the scope of this course, but a simple example will be used to outline basic operation. If you have specific waveform generation needs, you may need to research and operate one of the timers in a configuration not covered here.

The pins that are available to your project should be carefully considered, as pins may need to be committed to particular activities. For example, the ICSP interface has already consumed one of the PWM-capable outputs from Timer/Counter2.

The following example and supporting library code will operate Timer/Counter0 in 8-bit fast PWM mode. In fast mode, the timer counts up from 0 and resets after 255. In the most basic configuration this defines the period of the generated waveform. You may set the output compare register for the channel used to define the point at which the output pin is set (or cleared, depending on settings). In this configuration, you are determining the duty of the waveform by setting the output compare register value. Because the counters are only 8-bit, you have limited control over the duty. For simple timing, this would suffice. If you need higher resolution, you would use the 16-bit Timer/Counter1.

Since the period is determined by the 8-bit register size, and the clock fed to the timer, you have very little control over the period of the waveform. Timer/Counter0 allows selection of a prescale value that is one of **1**, **8**, **64**, **256**, or **1024**. This configuration of the timer significantly limits your options for period. Many applications of waveform generation are sensitive to duty, not period, so if this is your situation, this configuration will be appropriate.

Consider, for example, that the divide by **64** prescale is used. At 16MHz, the prescale will bring the clock down by a factor of **64**, and it takes **256** counts to make a full period. The output frequency then, at this rate is about 976.6Hz. If roughly 1kHz is what you are after then you are in luck. The fastest and slowest frequencies in this regime are 62.5kHz (prescale: 1) and just over 61Hz (prescale: 1024) respectively.

At 2MHz, it would probably make sense to use a lower prescale. To match the example above, a prescale of 8 with a 2MHz clock would yield the same results on the PWM clock.

The following code demonstrates how an A/D sample may be used to set the PWM duty:

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "sci.h"
#include "timer.h"
#include "atd.h"

// global variables

// constant for timer output compare offset, init and ISR rearm
const unsigned int _Timer_OC_Offset = 250; // 1 / (2000000 / 8 / 250) = 1ms (prescale 8)

// value to program into the OCR0A register (essentially sets duty)
//  at the start of the next cycle
volatile unsigned char _PWM_DutyVal = 0;

int main(void)
{
  // one-time initialization section

  // set up serial for 2MHz bus, 9600 BAUD, NO interrupts please!
  SCI0_Init(F_CPU, 9600, 0);

  // welcome message, so we know it booted OK
  SCI0_TxString("\n328 Up! Now with A/D and PWM!\n");

  // bring up the timer, requires ISR!
  Timer_Init(Timer_Prescale_8, _Timer_OC_Offset); // 1ms intervals

  AtoD_Init(AtoD_Channel_0);

  // bring up pwm (debug)
  Timer_F_PWM0(Timer_PWM_Channel_OC0A, Timer_PWM_ClockSel_Div8, Timer_PWM_Pol_NonInverting);

  // enable sleep mode, for idle, sort of similar to WAI on 9S12X (13.2)
  sleep_enable();

  // set the global interrupt flag (enable interrupts)
  // this is backwards from the 9S12
  sei();

  // main program loop - don't exit
  while(1)
  {
    // go idle!
    sleep_cpu();

    // read the A/D value and set the PWM duty based on it
    unsigned char AD_low = ADCL; // must be read first
    unsigned char AD_high = ADCH;
    unsigned int AD_Raw = AD_low + AD_high * 256;
    _PWM_DutyVal = (unsigned char)((AD_Raw / 1024.0) * 256);
  }
}
```

```
// output compare A interrupt
ISR(TIMER1_COMPA_vect)
{
  // rearm the output compare operation
  OCR1A += _Timer_OC_Offset; // 1ms intervals
}

// timer overflow (means end of PWM cycle)
// you don't need to use an interrupt with PWM, but it
//  sure makes figuring out when to set the new value
//  easy...
ISR (TIMER0_OVF_vect)
{
  OCR0A = _PWM_DutyVal;
}
```

The timer in this example is using **OCR0A**, which corresponds to pin 12 (**PD6**), which is where the output waveform may be observed.

Of all the topics covered here, PWM is the one you would need to do the greatest amount of research on to make it match your particular project needs. The starter library only serves this demo.

If you require substantial control over your PWM output, or multiple channels, you may need to consider an external PWM chip.

Adding Pin Interrupts

Essentially all of the pins on the ATmega328P device are capable of generating external interrupts. There are two special pins (**INT0** and **INT1**) that are more configurable for interrupts, and you would use these pins if you needed to capture specific edge conditions.

The remaining pins generate interrupts on any change (rising or falling), so if you can tolerate all transitions, then virtually any available pin will work for you. Pins labeled as **PCINT** operate under this scheme, and other than **VCC**, **GND**, **AREF**, and **AVCC**, all pins fall under this category.

While each pin is able to generate an interrupt, the pins are clustered into three groups:

- **PCI2**: **PCINT23** through **PCINT16**
- **PCI1**: **PCINT14** through **PCINT8**
- **PCI0**: **PCINT7** through **PCINT0**

So each pin can interrupt, but the expression of these interrupts is grouped internally. This could alter your preference for which pins to use, as you may want simple resolution of the interrupt source. For example, you may wish to use a pin from each group, so they fire separate interrupts, should you need multiple interrupt sources. Pins within the same group would require that you check the pin state to determine which pin caused the interrupt.

Each pin is separately masked as an interrupt source, so when you set out to configure the pins as interrupts, you need to do three things:

- Enable the pin as an input (the default state) (**DDR** register for the appropriate port)
- Enable the interrupt for the individual pin (**PCMSK2:0** register)
- Enable interrupts for the group to which the pins belongs (**PCICR** register)

Each group of pins has a separate interrupt, as **PCINT0_vect**, **PCINT1_vect**, and **PCINT2_vect**.

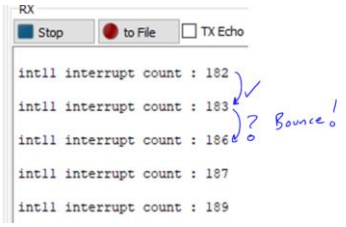
Pin **PC3** is shown in the sample schematic connected to a switch with a pull-up. The following code example shows how you could configure pin **PC3** (**PCINT11**) as an interrupt source:

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "sci.h"
#include "timer.h"

// global variables

// constant for timer output compare offset, init and ISR rearm
const unsigned int _Timer_OC_Offset = 6250; // 1 / (2000000 / 64 / 6250) = 200ms (prescale 64)

// variable to hold the number of interrupts seen on PC3 (int11)
volatile unsigned int _PinInterruptCount = 0;
```

```
int main(void)
{
  // one-time initialization section

  // set up serial for 2MHz bus, 9600 BAUD, NO interrupts please!
  SCI0_Init(F_CPU, 9600, 0);

  // welcome message, so we know it booted OK
  SCI0_TxString("\n328 Up! Pin interrupt demo!\n");

  // enable sleep mode, for idle, sort of similar to WAI on 9S12X (13.2)
  sleep_enable();

  // setup GPIO pin for interrupt
  // sample circuit uses PC3 (int11) on a MC switch
  // int11 is part of PCI1 group of pins
  PCMSK1 |= 0b00001000; // turn on PCINT11 pin mask (enable interrupts) (12.2.7)
  PCICR |= 0b00000010; // turn on interrupts for group 1 (12.2.4)

  // make pc3 an input (but should already be) (13.4.6)
  DDRC &= (~0b00001000);

  // set the global interrupt flag (enable interrupts)
  // this is backwards from the 9S12
  sei();

  // main program loop - don't exit
  while(1)
  {
    // go idle!
    sleep_cpu();

    char buff [50] = { 0 };
    (void)sprintf (buff, "\nint11 interrupt count : %d", _PinInterruptCount);
    SCI0_TxString (buff);
  }
}

ISR (PCINT1_vect)
{
  // flag is cleared automatically, so just need to take action
  ++_PinInterruptCount;
}
```

NOTE: Due to the high responsiveness of the interrupt to the pin state, any switch bounce is sure to be captured in all its glory. To mitigate this, you will need to use strategies learned in CMPE2250 to 'lockout' the switch behavior for a suitable period.

NOTE: The general interrupt pins cause an interrupt on rising and falling edge, so a button push and release will cause two interrupts.



Using I²C (TWI) on the ATmega328P

Due to the low pin count on the ATmega328 devices, you will probably find that connecting to external devices is best facilitated through either SPI (Serial Peripheral Interface) or TWI (two-wire interface, or I²C as you are familiar with from micro). Since the device programmer is connected on the SPI interface in the DIP version, you will be left with just TWI or UART, realistically, for connecting to external devices. This excludes, of course, simple analog or low pin count digital devices.

Note: the QFP version of the chip (the PB device) has more pins, and those pins expose an additional UART/SPI/TWI port. If your design requires SPI, the PB version could satisfy your needs, while still supporting the programmer. SPI will not be discussed in this document (yet).

As was the case in micro, TWI is sufficiently complex that a base driver library will be provided. As you did in micro, you will use the base library for all I²C calls. You will create device-specific libraries for each I²C device you need to use, and these device-specific libraries will all use the driver level I²C functions.

Samples of some common device-specific libraries may be included as a reference, so that you are able to create new libraries based on the datasheets for the devices you wish to use in your design.

Only a brief review of I²C will be discussed here, as you have had some exposure in micro. An overview of the I²C functions in the base library will be discussed, with a case example for communicating with the LM75AD,118 temperature sensor. Analysis of the device datasheet for this sensor will also be included.

Since you will likely be using devices that won't have libraries available, it will be your responsibility to research the communication method for each device from its datasheet, and create a new library to operate each device. You may reverse engineer what you find in other libraries (including ones provided for other platforms, as long as credit is given to the source).

An I²C transaction begins when the bus is not being actively used by another device. As discussed in micro, we substitute the words *Initiator/Responder* for *Master/Slave* respectively, that you will typically find in documentation. Your 328 chip will be the initiator, and all devices you connect to it will be responders. All communications are initiated and clocked* by the initiator.

Each external device will have a 7-bit address that is manufacturer-defined. You will need to look at the datasheet for each device to determine its address, and ensure that you don't select devices that will cause address conflicts.

Note: some devices have pins that you can tie high, tie low, or leave floating to alter the address. This scheme permits resolution of address conflicts, or operation of multiple identical devices on the same bus.

The LM75AD,118 datasheet indicates how the device address is determined, based on fixed address and the states of three external pins:

7.3 Slave address

The LM75A slave address on the I²C-bus is partially defined by the logic applied to the device address pins A2, A1 and A0. Each of them is typically connected either to GND for logic 0, or to V_{CC} for logic 1. These pins represent the three LSB bits of the device 7-bit address. The other four MSB bits of the address data are preset to '1001' by hard wiring inside the LM75A. Table 4 shows the device's complete address and indicates that up to 8 devices can be connected to the same bus without address conflict. Because the input pins, SCL, SDA and A2 to A0, are not internally biased, it is important that they should not be left floating in any application.

Table 4. Address table

1 = HIGH; 0 = LOW.

MSB						LSB
1	0	0	1	A2	A1	A0

Edited from LM75A datasheet (pages 5 and 6) – Address Determination

This means that the address for this sensor is **`0b1001000`** to **`0b1001111`** (**`0x48`** to **`0x4F`**) depending on how the address lines are connected. The datasheet also indicates that the address lines must not be left floating. These are all considerations for your design. Unless you have a reason to do otherwise, it is recommended that you tie address lines low to use the device 'base' address. This is the address a rational person would expect the device to be on, in the absence of other information.

Arguably the most confusing thing about the addressing in the I²C world is the appearance of the device address when in use. Device addresses are seven bits in size, but when used, the address appears shifted up one bit position, with the LSB being used to indicate desire to read or write. The byte that contains the 7-bit address will appear to have the address multiplied by two. It is important that you keep the 7-bit device address and the 8-bit announcement byte clear in your head.

Before communication with devices may commence, the TWI module must be initialized. By default, the TWI module will be powered off, and must be powered on. The timing characteristics for the clock must also be configured. The supplied library will do the necessary initialization, and only requires that you specify the device clock and desired I²C clock. A typical call that starts the I²C bus at 400 kHz would look something like this (in this case, using a 2MHz system clock):

```
// start I2C module
if (I2C_Init(2E6, I2CBus400))
       SCI0_TxString("Unable to bring up I2C module");
```

Transactions begin by grabbing the bus when idle, and announcing the target device address, along with intent to read from or write to the device. This process includes issuing a START condition. Most devices will perform an acknowledgement on announcement with read, and *all* devices will acknowledge *their* address on write. The driver-level function that starts the transaction will return various error codes if something in the process goes wrong.

From this point, the transaction may continue in the specified data direction. Devices that are receiving bytes (be it initiator or responder) may acknowledge or not acknowledge the byte they have received. In general, not acknowledging a received byte is a signal that the transaction should end (the sending device should stop sending bytes). When the initiator wishes to end the transaction, it should issue a STOP condition. Different devices have slightly different uses of the acknowledgment process, so you will need to look at the device datasheet to ensure that you are following the device protocol appropriately. The LM75A datasheet contains diagrams that explain the transaction timeline, and what the device expects or will do. Most devices have these diagrams in one form or another.

For example, the following diagram taken from page 12 of the LM75A datasheet, shows what a single byte register write transaction looks like:

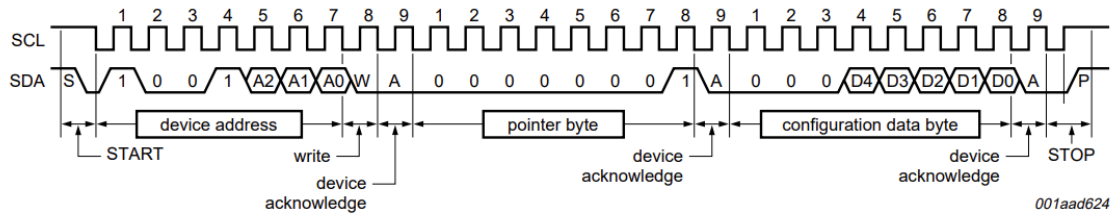


Fig 5. Write configuration register (1-byte data)

The start condition, followed by device address + intent to write is sent. The device *will* acknowledge this. The initiator continues with a write of the register (pointer byte) it intends to write to. Different devices have different registers, and similar to the micro, these registers mean something specific to the configuration of the device. Finally, the value for that register is written to the target device. The device promises to acknowledge all bytes it receives, so if no ACK is received, you know an error occurred. Once the last byte is sent, the initiator is free to send a STOP condition to release the bus.

The LM75A is configured with rational defaults, so you wouldn't typically need to write to any of its registers. It being a temperature sensor though, would mean that you certainly want to read the temperature from it.

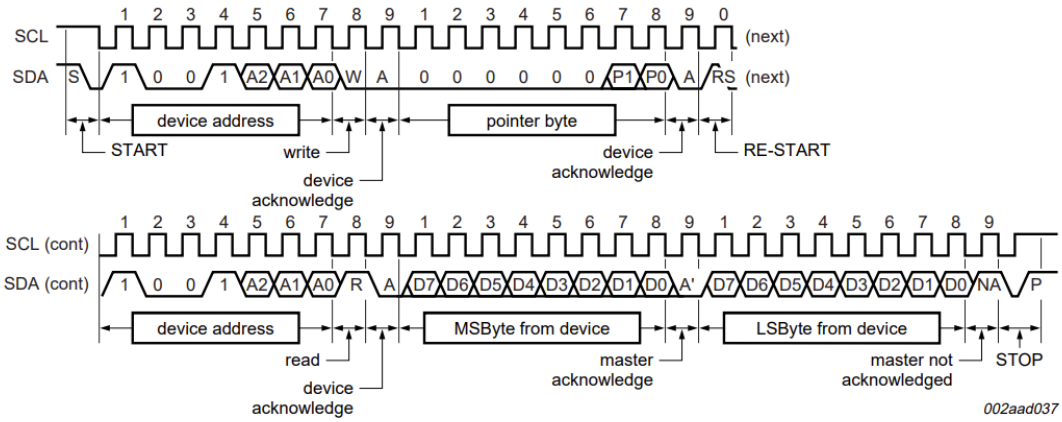


Fig 9. Read Temp, Tos or Thyst register including pointer byte (2-byte data)

To read the temperature, a transaction still needs to start, specifying the device address. The initial intent is to write, oddly, as the register to read needs to be written to the device. The pointer byte in this case would be set to the register that contains temperature information. Once acknowledged by the device, a RESTART condition would be sent to change the direction of the data to read. This is essentially a new START condition with intent to read. The device will now produce data when clocked, and it will continue to produce data until the received byte is not acknowledged. The temperature data requires two bytes, so the initiator should acknowledge the first byte, then not acknowledge the second byte. This tells the device to stop sending data, and the initiator may issue a STOP to end the transaction. *Note: For this device, it does not matter if the last byte is acknowledged by the initiator, as the STOP condition ends the transaction regardless. It is still a good idea to do what the datasheet defines as protocol, and standard ACK/NACK procedure might be more critical in other circumstances.*

In code, the *figure 9* transaction diagram shown above, is relatively easy to implement:

```
int LM75A_ReadTemp (unsigned int * uiTemp)
{
  unsigned char ucDataH, ucDataL;

  // start transaction with device address
  if (I2C_Start(0x48, I2C_Write))
    return -1;

  // want to read temperature (register zero)
  if (I2C_Write8(0, I2C_NOSTOP))
    return -2;

  // issue restart to change to read
  if (I2C_Start(0x48, I2C_READ))
    return -3;

  // read temperature data byte high, nostop, ack more data please
  if (I2C_Read8(&ucDataH, I2C_ACK, I2C_NOSTOP))
    return -4;

  // read temperature data byte low, stop, nack no more data please
  if (I2C_Read8(&ucDataL, I2C_NACK, I2C_STOP))
    return -5;

  // form up the raw data into a 16-bit raw value for return
  *uiTemp = ((unsigned int)ucDataH << 8) + ucDataL;
    return 0;
}
```

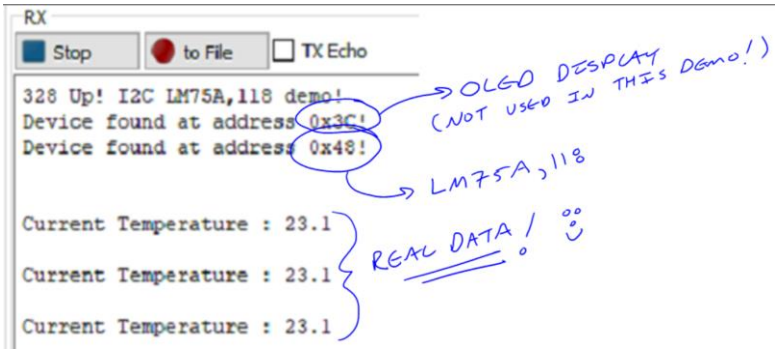
Interpretation of the data, what the registers in the device are, and other information about the target device are all found within the device datasheet. If you intend to use a particular device, it is imperative that you read all of the relevant sections of the datasheet and fully understand how the device operates before you select that part for use. The returned data in the example above, for example, only uses 11 of the 16 bits to represent the temperature, and the data is signed. Clearly, reading the datasheet to figure out how to interpret this is important!

You will be supplied with sample libraries for a number of devices. Ultimately, you would need to create your own libraries to communicate with other devices that you want to use. You should be able to apply what you see in the sample libraries to operate these new devices.

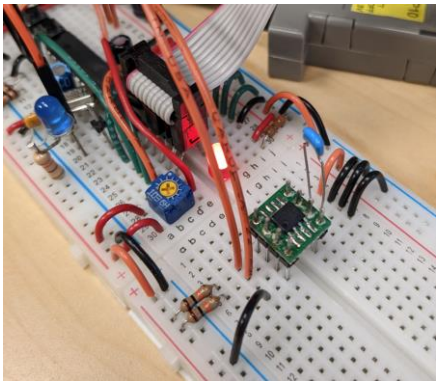
Remember: The **SCL** and **SDA** lines require pull-ups; don't forget this in your design!

```
#define F_CPU 2E6       // with external xtal enabled, and clock div/8, bus == 2MHz

#include <avr/io.h>
#include <util/delay.h> // have to add, has delay implementation (requires F_CPU to be defined)
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include "sci.h"
#include "timer.h"
#include "I2C.h"
#include "LM75A.h"

// global variables

// constant for timer output compare offset, init and ISR rearm
const unsigned int _Timer_OC_Offset = 3125; // 1 / (2000000 / 64 / 3125) = 100ms (prescale 64)

// global isr-modified 100ms tick counter
volatile unsigned long _Ticks = 0;

int main(void)
{
  // one-time initialization section

  // set up serial for 2MHz bus, 9600 BAUD, NO interrupts please!
  SCI0_Init(F_CPU, 9600, 0);

  // welcome message, so we know it booted OK
  SCI0_TxString("\n328 Up! I2C LM75A,118 demo!\n");

  // bring up the timer, requires ISR!
  Timer_Init(Timer_Prescale_64, _Timer_OC_Offset); // 200ms intervals

  // bring up the I2C bus, at 400kHz operation
  I2C_Init(2E6, I2CBus400);

  // test bus for devices, should find our trusty LM75A!
  unsigned char devidbuff[128] = { 0 };
  I2C_Scan(devidbuff);
  for (int i = 0; i < 128; ++i)
  {
    if (devidbuff[i])
    {
      char buff[50] = { 0 };
      (void)sprintf (buff, "Device found at address 0x%X!\n", i);
      SCI0_TxString(buff);
    }
  }

  // enable sleep mode, for idle, sort of similar to WAI on 9S12X (13.2)
  sleep_enable();

  // set the global interrupt flag (enable interrupts)
  // this is backwards from the 9S12
  sei();
```

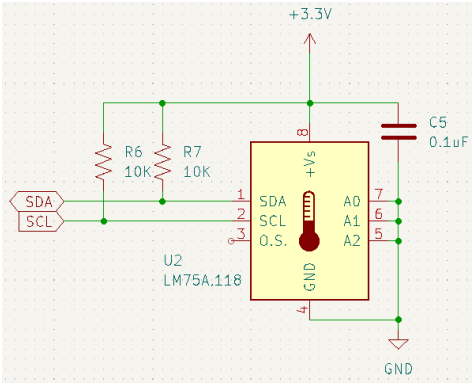
```
// main program loop - don't exit
  while(1)
  {
    // go idle!
    sleep_cpu();

    // sample and show temperature every 1s
    if (!(_Ticks % 10))
    {
      char buff [50] = { 0 };
      (void)sprintf (buff, "\nCurrent Temperature : %0.1f", LM75A_GetTempF());
      SCI0_TxString (buff);
    }
  }
}

// output compare A interrupt
ISR(TIMER1_COMPA_vect)
{
  // rearm the output compare operation
  OCR1A += _Timer_OC_Offset; // 100ms intervals @ prescale 64

  // count 100ms ticks!
  ++_Ticks;
}
```



Capture of UART output showing temperature



Updated breadboard



LM75A,118 Schematic Diagram used for example

Libraries for the base I2C functions, the LM75A,118, and other devices will be available on Moodle.