IntroToMC.odt
27jul24
09oct24

Beyond Arduino: Introduction to Industrial Micro-Controllers.
by Nels D. “Chip” Pearson (aka CmdrZin)

These tutorials will explore the Microchip tinyAVR 8bit families of Micro-controllers.
Most, if not all, of the hardware resources of these devices will be demonstrated using multiple software examples and a variety of programming techniques.
Numerous hardware interfacing examples to external devices will showcase the I/O capabilities of the devices.

The MPLAB X IDE system will be the main environment used for code development.

Target Devices:		FLASH	RAM	EEPROM	I/O	Price	
ATtiny212/412	8SIOC	2k/4k	128/256	64/128	6/6	\$0.49/25	\$0.53/25
ATtiny814/1614	14SIOC	8k/16k	512/2048	128/256	12/12	\$0.74/25	\$0.83/25
ATtiny1616/3216	20SIOC	16k/32k	2048/2048	256/256	18/18	\$1.05/25	\$1.17/25

Bare bones breakout boards for 8SIOC, 14SIOC, and 20SIOC device packages will be used with protoboards for demonstrations and experiments.

TODO:

Verify DAC section.
Verify TCB section.
Verify SPI – Host section.
Verify WDT section.

Table of Contents

Introduction

This project assumes an average level of experience with the Arduino development environment. Its goal is to free you from the constraints of the Arduino 'pre-determined hardware use' environment and allow the hardware resources of the micro-controller to be used as you see fit. Code examples will show the user how to replicate many of the Arduino support functions without being limited to the ATmega324 device used on most Arduino boards.

The Microchip tinyAVR 1-series family was selected due to its straight forward architecture, low cost, and variety of hardware resources in such small packages. Although these devices are SIOC packages (break-out boards are available), similar Atmel devices from the prior generation are still available in DIP packages and can be substituted with minimal effort.

The processes followed in this document to explore this family of Microchip devices can be applied to the newer AVR EA, EB, DA, DB, and DU families as well. Use the link[1] to obtain more information.

Many code examples use macros for bit settings. These usually end in _bp, _bm, or _gc. The _gc is a group code macro and represents a group of bits positioned for a specific area in the register. For example, `ADC_RESSEL_8BIT_gc` is a pattern 00000100 which was generated by shifting a 01 pattern two bits to the left with a `(0x01<<2)` statement and is used to set the `RESSEL` area of the `ADCn.CTRLA` register. (see iotn412.h for more examples). The _bm is a bit mask for an individual bit at a specific position. Its position is generated with a bit shift statement like `(1<<3)` which will generate the pattern 00001000. For example if `A=(1<<3)`, then to set the bit, use

```
reg |= A;
```

to OR the bit with the reg contents and to clear that bit position, use

```
reg &= ~A;
```

to AND the inverse of the pattern (11110111) with the reg contents. The _bp is just the number of the bit position. In the prior example, if we used `#define X_bp 3`, then we could use `A=(1<<X_bp)` to get the same results. X could be replaced with a meaningful name to make the code more readable.

SECTION I - Setup

1. Introduction and where to find additional information.

a. Microchip's website.[1][3]

Use the link[3] to access the Microchip website, download the MPLAB X IDE, and install it. The IDE is available for Windows, Linux, and macOS systems. Be sure to install the XC8 compiler.

b. UPDI Programmer

You will also need a UPDI capable programmer to program/debug the target device. (see the **UPDI Programmers** section for suggestions)

2. Establishing a Development Environment: Hardware and Software Integration.

a. Hardware target.

Simple solder-less breadboard or protoboard with a DC power supply between 3.0 and 5.0 vdc able to provide a minimum of 100ma of current.

A three cell AA or AAA battery pack (4.5v with standard batteries, 3.6v with rechargeable) works well as would a LiPo battery (3.7v). The devices can use supply voltages from 1.8v to 5.5v.[2] The target MCU will be an 8SIOC device mounted on a breakout board.

Figure 1 shows how the power supply, MCU board, and SNAP UPDI header are wired together.

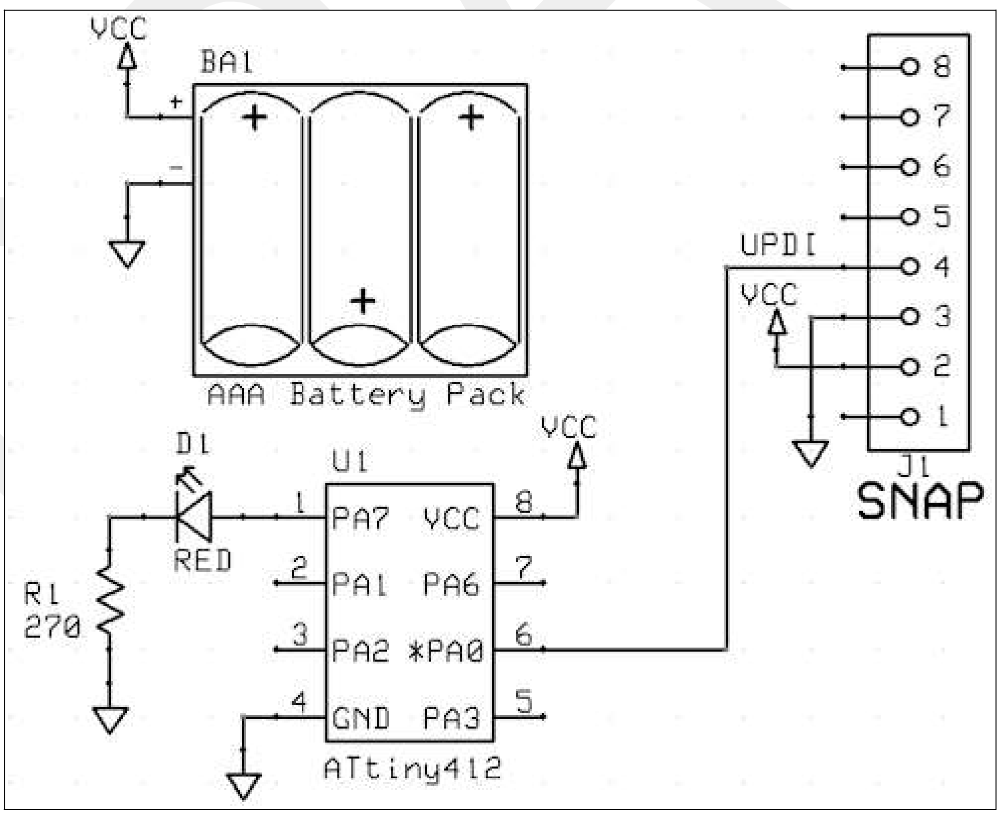


Figure 1

b. MPLAB X setup and UPDI programmers.

Download and install MPLAB X IDE from the Microchip website.[3]
Open MPLAB and connect the SNAP module to the computer.
Select File > New Project Microchip Embedded > Standalone Project then Next >.
In the Select Device pop-up, use
Family: 8-bit AVR MCUs (XMega/Mega/Tiny/AVR)
Device: ATtiny412
Tool: SNAP or Simulator if you do not have a programmer tool.
Next >
In the Select Compiler, pick XC8. Next >
Select a Project Name and Folder to store the project into. Finish.
Once the project is set up, RMC Source File in the Projects list then New > avr-main.c
Change File Name to main. Finish.
NOTE: The SNAP programmer has to be modified per ETN #36 "MPLAB SNAP AVR UPDI/PDI/TPI Interface Modification engineering technical note". It may also need to have its software updated using the MPLAB IPE. This is evident if the connection window shows no S/N for the SNAP when trying to connect.
If MPLAB X will not program with the SNAP, use the MPLAB IPE and the .hex file from the ../dist/default/production directory in the project folder.

c. Blink the LED.

Replace the main(){ } code in the main.c file with this code.

```
int main(void)
{
   uint16_t count = 0;
   /* set PA7 of PORT A as an output pin */
   PORTA.DIR |= PIN7_bm;

   // Toggle pin every 65536 counts.
   while (1) {
      if( count == 0) {
         PORTA_OUTTGL = PIN7_bm;
      }
      ++count;
   }
}
```

Compile the code and download it into the device.

The complete project, ITMC_I_2.X.zip, can be downloaded from "https://www.gameactive.org/dist/".

SECTION II - Exploration of Basic Hardware

1. Digital Input/Output (I/O)

Now for an explanation of what it took to control the LED. A typical red LED requires about 1ma of current flowing through it to generate detectable light. Almost all of them are rated for a maximum of 20ma continuous current and will tolerate short pulses of 40ma or more.

A little electronics theory will explain this common LED circuit seen in many examples. It consists of a resistor in series with an LED and is designed to result in a nominal 5ma to 10ma to light up the LED. The resistor has to be adjusted based on the power source.

Two circuits are shown; one using a 5v source (typical of many micro-controllers) and another using a 3.7v source (a system using a LiPo battery as its power source). A typical RED LED has a voltage drop of about 1.7v. The resistor has to cause a voltage drop such that the total of voltage changes in the circuit is zero.

Using Ohm's law, E = I*R, or E/R = I to calculate the resistor value for a nominal current value, we have.

```
+5 --------/\/\/\-------------|>|-------GND   +3.7 --------/\/\/\-------------|>|-------GND
              470               D                              270               D
     3.3v/470 = 7.0ma       1.7v                       2.0v/270 = 7.4ma       1.7v
```

So, +5 - 3.3v - 1.7v = 0 and +3.7 - 2.0v - 1.7v = 0.

a. Pick a pin. Configure the PORT.

Now that the external circuit parameters are known, 5v @ 7.0ma or 3.7v @ 7.4ma, to light the LED, you need to determine if this micro-controller can meet these requirements.
We will use the ATtiny412 device and a 3.6 AAA battery pack with NiCad rechargeable batteries and PA7 as the control pin.

Section 35.10 Table 35-16 states the I/O Pin Characteristics. V_{OH} gives the I/O pin drive strength with V_{DD} = 3.0v and an output current of 7.5ma, the pin will maintain a minimum level of 2.4v. So for a V_{DD} of 3.6v, it should maintain about 3.2v or so which is still high enough to light the LED.
Note that higher currents are available with higher V_{DD} supply voltages. i.e. use +5v V_{DD} and a smaller resistor (allowing more current to flow) for brighter LEDs!

b. Optional PORT Control

Section 16.3.2.2[2] states that the Virtual PORT register map method should be used to control I/O port. This provides some benefits, as explained, but we will use the standard name PORT. Table 16-1[2] shows that to access the regular port registers, just use VPORT instead of PORT.

c. Output Options.

A. Source and Sink current considerations.

The example schematic in Figure 1 is using the MCU pin to source current to the LED. The current flows out of the pin, though the LED and resistor, and into ground (the negative side of the battery).
The LED could have been connected to VCC and then the MCU pin would act as a current sink.
The current would flow from the battery plus terminal, through the LED and resistor, and into the MCU pin.
Whether the circuit is configured for a source or a sink, the calculations for the resistor still have to result in the total voltage changes being zero and the MCU pin has to be able to handle the source current or the sink current. Always check the data sheet! Some MCUs can sink more current than they can source.
B. Control Logic

To use an I/O PORT pin as an output, source or sink, the bit in the direction register PORTA.DIR associated to the pin must be set to a '1'. The example code line

```
PORTA.DIR |= PIN7_bm;
```

does this. The 'x' in PORTx has been replaced with the port being used; an 'A' for this device. Some devices have many ports: PORTA, PORTB, PORTC, etc. Just use the appropriate letter for the port to be controlled.
The `PIN7_bm` is a defined literal for the bit pattern `1000 0000`. Looking at 16.5.1[2], the DIR register, this pattern OR'd into the register will 'set' bit 7 to a '1' and cause PA7 to be an output pin with its output driver enabled.
The output pin can be set HIGH or LOW by writing a '1' or a '0' respectively using the

```
PORTA.OUT = <pattern>
```

command. It the example code to toggle the bit, the

```
PORTA_OUTTGL = PIN7_bm;
```

command is used with the bit pattern for pin 7.
Other control registers are discussed in section 16.5.1[2] and you may want to experiment with them.

d. Input Options.

A. Pull-ups considerations.
The example schematic in Figure 2 has added a push-button switch to circuit from Figure 1.
This is a common configuration used for simple user inputs. Note that is does not have an external pull-up resistor to pull the line HIGH when the button is not pressed. To minimize part count, most MCUs have optional internal pull-up resistors. Section 36.10[2] shows this internal resistor to be about 35k, when activated.

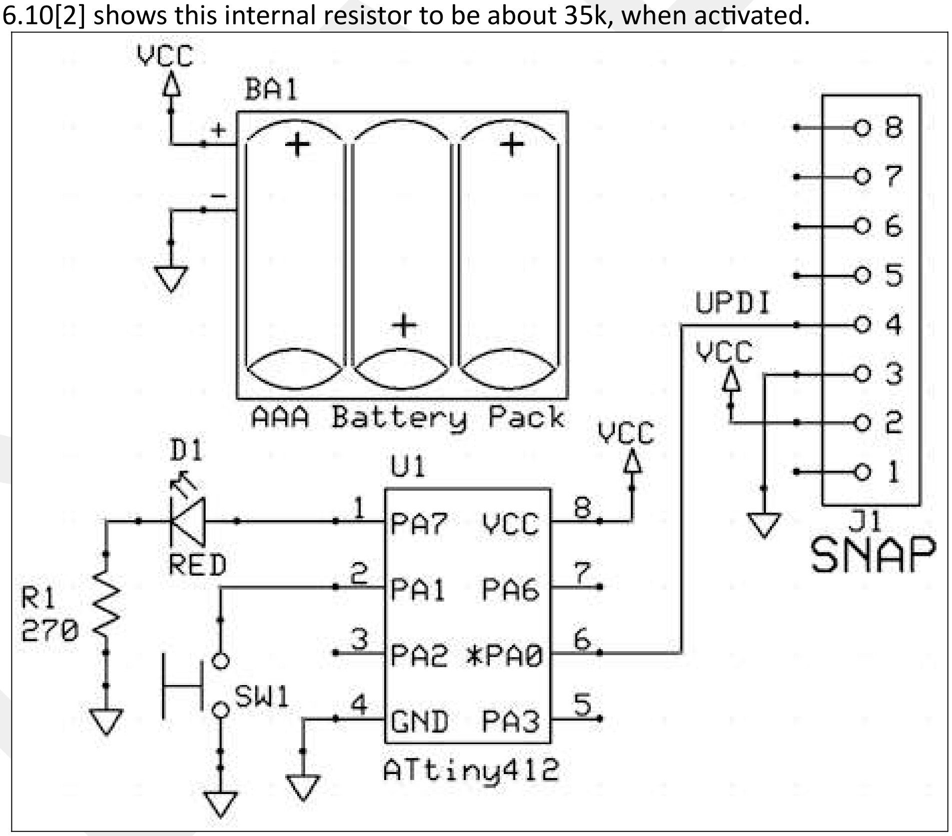


Figure 2

To use an I/O PORT pin as an input, the bit in the direction register PORTA.DIR associated to the pin must be set to a '0'. Since this is the reset condition of the bit (see section 16.5.1[2], nothing needs to be done. If you need to clear this bit due to other code operations, this example line of code could be used.

```
PORTA.DIR &= ~PIN1_bm;
```

It inverts the bit mask `0000 0010` to `1111 1101` and ANDs it to the DIR register contents.

Another register, PORTA.PINnCTRL (see 16.5.11 [2]) also affects the operation of this pin. The 'n' refers to the pin number, so 'n=1' in this case.
The reset (default) settings for all pins is Non-Inverted, Pull-up disabled, and INTDISABLE.
Since the pull-up for pin 1 will be used set bit 3 to '1' with

```
      PORTA.PIN1CTRL |= (1<<3);  // set PULLUPEN = 1
```

Read the state of the pins into a variable. For example, use

```
      portA_pins = PORTA.IN;     // read all the pins of PORTA.
```

To test this new configuration, replace the main(void) code (main02.c) with the following:

```
    int main(void) {
        uint8_t portA_pins = 0;

        /* set PA7 of PORT A as an OUTPUT pin. The other bits are left as '0'
         * so that their associated pins will be INPUT pins. */
        PORTA.DIR |= PIN7_bm;

        /* enable the internal Pull-up resistor for PORTA pin 1 */
        PORTA.PIN1CTRL |= (1<<3);   // shift the '1' into the bit3 position.

        // Control the LED with the push button switch.
        while (1) {
            portA_pins = PORTA.IN;          // read all the pins of PORTA.
            portA_pins &= PIN1_bm;          // set all bits except bit.1 to '0'.
            if( portA_pins == 0 ) {
                /* turn ON the LED if bit.1 (pin1) is LOW (i.e. switch is closed) */
                PORTA_OUTSET = PIN7_bm;
            } else {
                /* turn OFF the LED if pin1 is HIGH (i.e. switch is open) */
                PORTA_OUTCLR = PIN7_bm;
            }
        }
    }
```

Compile the code and download it into the device. The LED can now be controlled by the switch.

The complete project, ITMC_II_1234.X.zip, can be downloaded from "https://www.gameactive.org/dist/".

Now that basic INPUT and OUTPUT control is available, try adding more LEDs and switches to do more complex operations. A simple system might use two LEDs, one RED and one GREEN, and three switches; A, B, and C. When a majority of switches are pressed the GREEN LED is turned ON. If not, the RED LED is lit instead.

2. Timing - Part 1 - Simple Timer - TCA

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3217-Getting-Started-with-TCA-DS90003217.pdf

Section 10[2] describes the clock control registers. On reset, these registers default to generating a 3.3 MHz peripheral clock (CLK_PER) that is used as the base clock for all peripherals. This will be the starting point for exploring Timer A; a 16-bit Timer/Counter described in Section 20[2]. The goal is to blink the LED in Figure 1 at a constant rate.

a. Configuring the TCA to count up and set a bit on overflow.

The TCA has many features and operating mode. The ref above explores them and provides example code to experiment with. This section will look at the most basic of timer operations: counting internal clock pulses (CLK_PER).

The counter will be used in the NORMAL mode (see 20.3.3.1[2]). This is a 16-bit counter and will overflow back to 0 every 65356 counts. The CLK_PER is CLK_MAIN (usually 20MHz) divided by 6 resulting in a 3.3 MHz (3.333333×10^6) clock which would cause the counter to overflow every

$20 \times 10^6 / 6 / 65356 = 51.0$ Hz

Too fast to see a blinking LED. Setting the TCA.CTRLA to divide the incoming CLK_PER by 64 will slow the overflow rate down to about 0.8 seconds.

The counter also must be enabled to operate. The code below can be used to do these tasks.

```
/* set Normal mode */
TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_NORMAL_gc;

TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV64_gc  /* set CLK_PER/64) */
                  | TCA_SINGLE_ENABLE_bm;      /* start timer */
```

When it overflows, it will set the OVF bit is the INTFLAGS register (see 20.5.11[2]). This bit is not cleared automatically and must be cleared by writing a '1' to its bit location to allow it to bet set on the next overflow. The following example code can be used to do this.

```
/* The OVF flag has to be cleared manually */
TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
```

The following example code (main03.c) will use this information to toggle the LED in Figure 1 on or off approximately every 1.3 seconds.

```
int main(void) {
    /* set PA7 of PORT A as an OUTPUT pin. The other bits are left as '0'
     * so that their associated pins will be INPUT pins. */
    PORTA.DIR |= PIN7_bm;

    /* *** TCA Configuration as NORMAL counter *** */
    /* set Normal mode */
    TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_NORMAL_gc;

    TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV64_gc  /* set CLK_PER/64) */
                      | TCA_SINGLE_ENABLE_bm;      /* start timer */

    // Blink the LED at a rate set by the TCA overflow rate.
    while (1) {
        /* Test for overflow */
        if( TCA0.SINGLE.INTFLAGS & TCA_SINGLE_OVF_bm) {
            /* reset the flag */
            /* The OVF flag has to be cleared manually */
```

```
            TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
            /* and toggle the LED state. */
            PORTA_OUTTGL = PIN7_bm;
        }
    }
}
```
Compile the code and download it into the device. The LED is now being controlled by TCA timer.

This modified code is in the main03.c file. The complete project, ITMC_II_1234.X.zip, can be downloaded from "https://www.gameactive.org/dist/".

3. Timing - Part 2 - Simple Timer - TCA using its interrupt

Configuring the TCA for NORMAL operation is the same as in the previous section. The additional step is to enable the interrupt on overflow bit in register `TCA0.INTCTRL`. The following code can do that.

```
/* enable overflow interrupt */
TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;
```

Since an interrupt is being active, it must have an interrupt handler (ISR) to take control when the MCU jumps to the interrupts vector. The following example code will do that. This code will also toggle the LED.

```
ISR(TCA0_OVF_vect)
{
    /* toggle the LED state. */
    PORTA_OUTTGL = PIN7_bm;
    /* The interrupt flag has to be cleared manually */
    TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
}
```

Additionally, the interrupt.h header is needed and the command to enable global interrupts. This is the modified main.c file (main04.c) code for this example. The changes are in **BOLD**.

```
#include <avr/io.h>
#include <avr/interrupt.h>          // to support the use of interrupts

/* TCA interrupt service routine. */
ISR(TCA0_OVF_vect)
{
    /* toggle the LED state. */
    PORTA_OUTTGL = PIN7_bm;
    /* The interrupt flag has to be cleared manually */
    TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;
}

int main(void) {
   /* set PA7 of PORT A as an OUTPUT pin. The other bits are left as '0'
   * so that their associated pins will be INPUT pins. */
   PORTA.DIR |= PIN7_bm;

   /* *** TCA Configuration as NORMAL counter with interrupt *** */
   /* enable overflow interrupt */
   TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;

   /* set Normal mode */
   TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_NORMAL_gc;

   TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV64_gc  /* set CLK_PER/64) */
                     | TCA_SINGLE_ENABLE_bm;    /* start timer */

   /* enable Global interrupts */
   sei();

   // Blink the LED at a rate set by the TCA overflow rate.
   while (1) {
       ;       // LED control is handled by the interrupt service.
   }
}
```

Compile the code and download it into the device. The LED is now being controlled by TCA timer interrupt service routine.
As a simple test, change `TCA_SINGLE_CLKSEL_DIV64_gc` to `TCA_SINGLE_CLKSEL_DIV16_gc` to make the LED flash 4 times faster.

This modified code is in the main04.c file. The complete project, ITMC_II_1234.X.zip, can be downloaded from "https://www.gameactive.org/dist/".

4. Timing - Part 3 - System Timer

A very common resource in many systems is the one millisecond time reference or millis() function. This function returns a value equal to the number of milliseconds that the system has been operating since reset. The value is normally an unsigned long or uint32_t and will overflow about every 50 days so, long enough for most projects.

TCA will be configured to generate an interrupt every 1ms and additional code will be added to provide the millis() function. The millis() function will then be used to control the LED in Figure 1.

The first task is to configure the TCA to interrupt every 1ms. Since the CLK_PER is at 3,333,333 Hz and if the prescaler is set to x1, the counter will be at 1ms after 3,333 counts. The TCA0_SINGLE_PER register sets the TOP value for the counter. This is the overflow point. At reset, this TOP value is 0xFFFF, so the full 16-bits of the counter are used. The configuration code for the TCA is now:

```
    /* *** TCA Configuration as NORMAL counter with interrupt *** */
    TCA0_SINGLE_PER = 0x0D04;    // 3,333 - 1 since the counter starts at 0.
    TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;
    TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_NORMAL_gc;
    TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV1_gc  /* set CLK_PER/1) */
                      | TCA_SINGLE_ENABLE_bm;      /* start timer */
```

The ISR will now update a global variable to accumulate 1m 'ticks'. A service function will allow other parts of the program to access the accumulated value in a reliable method. This is the additional code.

```
volatile static uint32_t totalMilliseconds;      // global variable

/* This interrupt service is called every 1 ms. */
ISR(TCA0_OVF_vect)
{
     ++totalMilliseconds;
     TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;  // clear interrupt flag.
}

/* Return the total number of milliseconds since the project started. */
uint32_t millis()
{
     uint32_t temp;              // make a holder for the counter.

     cli();                      // Turn OFF interrupts to avoid corruption
                                 // during a multi-byte read.
     temp = totalMilliseconds;   // get a copy while interrupts are disabled.
     sei();                      // Turn interrupts back ON.

     return temp;                // return a 'clean' copy of the counter.
}
```

The new main.c file (main05.c) should now look like this.

```
#include <avr/io.h>
#include <avr/interrupt.h>      // to support the use of interrupts

#define LED_DELAY     1000UL     // N * 1ms

volatile static uint32_t totalMilliseconds;      // global variable

uint32_t millis();               // prototype for function

/* TCA interrupt service routine. */
```

```
ISR(TCA0_OVF_vect)
{
   ++totalMilliseconds;
   TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;   // clear the interrupt flag
}

/* Return the total number of milliseconds since the project started. */
uint32_t millis()
{
   uint32_t temp;               // make a holder for the counter.

   cli();                  // Turn OFF interrupts to avoid corruption
                           // during a multi-byte read.
   temp = totalMilliseconds;    // get a copy while interrupts are disabled.
   sei();                  // Turn interrupts back ON.

   return temp;            // return a 'clean' copy of the counter.
}

int main(void) {
   uint32_t ledTime = 0UL;

   /* set PA7 of PORT A as an OUTPUT pin. The other bits are left as '0'
    * so that their associated pins will be INPUT pins. */
   PORTA.DIR |= PIN7_bm;

    /* *** TCA Configuration as NORMAL counter with interrupt for 1ms OVF ***
*/
   TCA0_SINGLE_PER = 0x0D04;    // 3,333 - 1 since the counter starts at 0.
   /* enable overflow interrupt */
   TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;
   /* set Normal mode */
   TCA0.SINGLE.CTRLB = TCA_SINGLE_WGMODE_NORMAL_gc;
   TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV1_gc  /* set CLK_PER/64) */
             | TCA_SINGLE_ENABLE_bm;    /* start timer */

    /* enable Global interrupts */
    sei();

    // Blink the LED at a rate set by the TCA overflow rate.
    while (1) {
       if( millis() > ledTime ) {
          ledTime = millis() + LED_DELAY;
          /* toggle the LED state. */
          PORTA_OUTTGL = PIN7_bm;
       }
   }
}
```
Compile the code and download it into the device. The LED is now being controlled by timing test in the while() loop using the millis() function based on the 1ms interrupt 'tick' from the TCA interrupt service routine.

As a simple test, change the value of LED_DELAY to change the LED flash rate.

This modified code is in the main05.c file. The complete project, ITMC_II_1234.X.zip, can be downloaded from "https://www.gameactive.org/dist/".

5. General Programming

With the basics of Input, Output, and Timing covered, the support code for these functions should be organized into their own files for better version and configuration control. This will be very helpful as the project grows and coding becomes more complex.

A common architecture may look like this:

a. main.c - Overall project control.
b. systime.c - time support and timer initialization functions.
   systime.h - header file to be added to main.c to access time functions.
c. io_ctrl.c - input/output control and initialization functions.
   io_ctrl.h - header file to be added to main.c to access i/o functions.
d. sysdefs.h - (optional) global definitions file.

Starting with main05.c having all of the source code, RMC on all of the other 'main' files and select "Remove from project" then rename main05.c to main.c. To create the two new files, use Source Files > New > avr-main.c > File Name: systime > Finish and Source Files > New > avr-main.c > File Name: io_ctrl > Finish

For the header files, use

Header Files > New > C Header File > File Name: systime > Finish and
Header Files > New > C Header File > File Name: io_ctrl > Finish and for the optional sysdefs.h file, use
Header Files > New > C Header File > File Name: sysdefs > Finish

The hierarchy tree should now show three Header Files and three Source Files.

Open main.c, systime.c, and systime.h open in the Editor by double clicking them if then are not already open.

In systime.c, delete all lines below the `#include <avr/io.h>` line.

Move, from main.c, the lines for

```
volatile static uint32_t totalMilliseconds;      // global variable
ISR(TCA0_OVF_vect)
{
  ...
}
uint32_t millis()
{
 ...
}
```

into the systime.c file. Also, add the code

```
#include "systime.h"
```

just under the `#include <avr/io.h>` line.

Add a function void init_systime(void) to the systime.c and move all of the TCA0 initialization code from main.c into it.

Put as prototype for it in systime.h and add `#include "systime.h"` to the main.c file.

Place a call to `init_systime()` where the TCA0 code was in main.c.

In the systime.h file, delete all lines of the __cplusplus `#if` conditional, unless you plan to code in C++ later.

Copy

#include <avr/interrupt.h>

from main.c and add it under the `#define SYSTIME_H` line.

Move the `uint32_t millis();` line from main.c to systime.h also.

For io_ctrl.c, remove the main() function code block. Add an `#include "io_ctrl.h"` statement.

Add a function called `void init_io(void){ }` and move the PORTA initialization code from main.c into it. Then add a call to `init_io()` where the code was.

In io_ctrl.h, remove the cplusplus code and add a prototype for `void init_io(void).`

Compile the code and download it into the device. The LED is being controlled by timing test in the while() loop using the millis() function based on the 1ms interrupt 'tick' from the TCA interrupt service routine as before.

The current project ITMC_II_5.X.zip, reorganized in this method, can be downloaded from "https://www.gameactive.org/dist/". This is one of many way to organize your projects.

Other things that can be done:

Move the _OUTTGL operation into a function called toggle_LED().
Make led_ON() and led_OFF() functions.
Make a blink_LED(int n) function to blink the LED n times at a 200ms rate.

SECTION III - Exploration of Unique Hardware

This section will look at the variety of hardware peripherals available in the AVR ATtiny family of MCUs and explore some of their features. These selections do not have to be reviewed in any particular order as they are essentially independent of each other.

1. Serial Communications - USART

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3216-Getting-Started-with-USART-DS90003216.pdf

A common interface between a project and a PC or laptop is through a USB-to-Serial board. These are available from Adafruit and Sparkfun and come in 5v and 3.3v versions. Using the AAA battery pack as a power source, use a 3.3v board if using rechargeable AAAs, use a 5v board if using standard or alkaline AAAs. Alternately, the USB board can supply power to replace the battery pack.

Figure III-1 shows an example circuit used to explore the Serial Communications port (USART) of an ATtiny412. It uses the USB serial board for power instead of the battery pack.

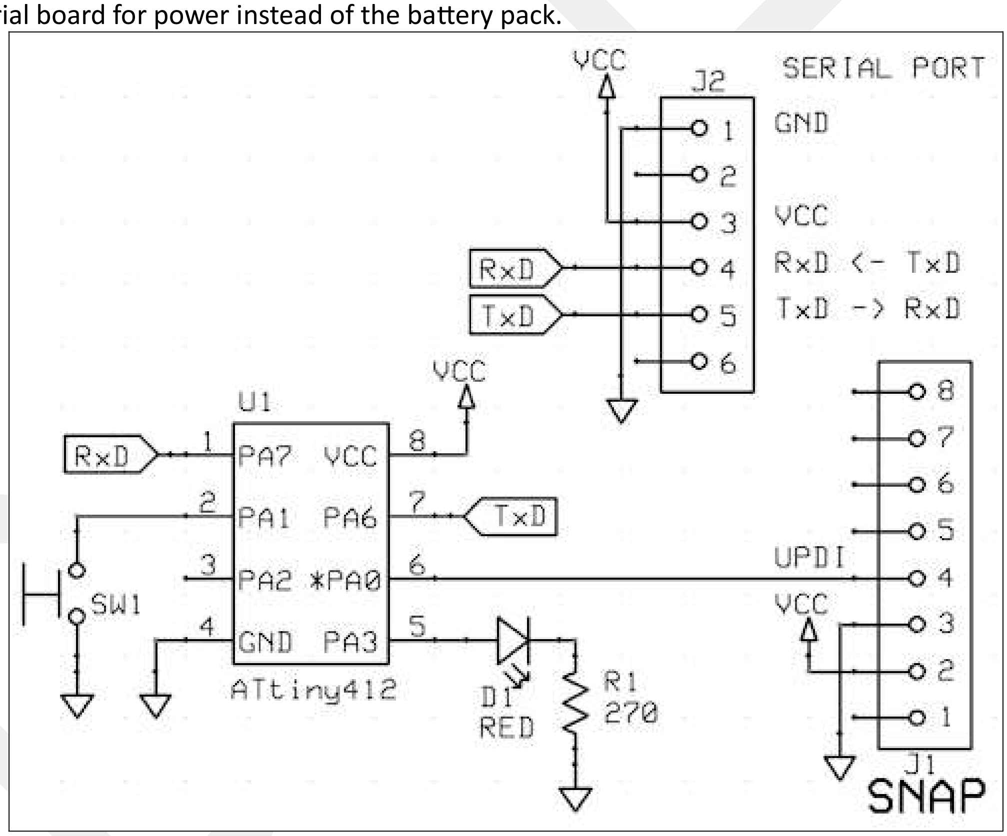


Figure III-1

The 6-pin serial port follows the FTDI standard and should be a male 0.1" header to match the female connector used on most USB-to-Serial boards.

The LED was moved to PA3 to free up PA7 since it is the receive (RxD) pin. PA6 is the transmit out (TxD) pin. (see Table 5.2 USART0)[2]

NOTE: Figure III-1 shows the SIOC BOARD pins numbers, NOT the ATtiny412 device pin numbers.

Connect the USB cable to the PC and in MPLAB X select Window > Debugging > Data Visualizer to bring up the MPLAB Data Visualizer window. A COM port should show up in the Serial Ports panel. Select the gear symbol to configure the port to 9600 8N1 if not already set that way. This tool can be used to display the serial output for this example.

Optionally, an Arduino board can be used as something to talk to. Connect RxD to Arduino TxD and TxD to Arduino RxD. Be sure to use the Arduino +5v power for the ATtiny412 instead of the battery pack.

Section 24[2] describes the USART operation and its registers. Two files are added to the project:

Header Files > SerialPoll.h
Source Files > serialPoll.c

These will contain the serial support code. A simple polling process will be used. See [ref] for ideas on how to implement an interrupt based system.

The USART will be set up for Full-Duplex, Asynchronous, 8N1 communications at 9600 baud. Section 24.3.1[2] lists the steps needed to initialize the USART for this mode.

a. Baud rate

USART0.BAUD sets the BAUD rate (i.e. the bit rate of communications). The following lines of code will calculate the baud rate based on the CPU clock rate.

```
#define F_CPU 3333333L            // CLK_PER is 3.3 MHz
#define USART_BAUD_RATE(BAUD_RATE) ((float)(F_CPU * 64)/(16 * (float)BAUD_RATE)+0.5)
```

and

```
USART0_BAUD = (uint16_t)USART_BAUD_RATE(baud);
```

will set the baud rate.

b. Frame format

USART0.CTRLC sets the frame format. The register is set to Asynchronous 8N1 with predefined bit group code (_gc) found in the iotn412h file.

```
USART0_CTRLC = USART_CHSIZE_8BIT_gc;
```

c. TxD as an output pin

Set the DIR register to make pin PA6 an output. PA7 (RxD) will be an input by default.

```
PORTA_DIR |= PIN6_bm;       // TxD as an OUTPUT
```

d. Enable transmitter and receiver

Use CTRLB to enable the receiver and transmitter.

```
USART0_CTRLB |= (USART_RXEN_bm | USART_TXEN_bm);
```

All of this can be gathered into an initialization subroutine as shown below.

```
/* Initialized USART0 registers and IO pins */
void USART0_init(uint16_t baud)
{
  /* Set IO port pin directions. */
  PORTA_DIR &= ~PIN7_bm;          // Rx as INPUT (default)
  PORTA_DIR |= PIN6_bm;           // Tx as OUTPUT

  /* Set baud rate */
  USART0_BAUD = (uint16_t)USART_BAUD_RATE(baud);

  /* Set frame format: 8bit Data, No Parity, 1stop bit (8N1) */
  USART0_CTRLC = USART_CHSIZE_8BIT_gc;  // default

  /* Enable receiver and transmitter */
```

```
    USART0_CTRLB |= (USART_RXEN_bm | USART_TXEN_bm);
  }
```

If a byte is written into the TXDATAL register, the USART will transmit it out the TxD line. The register should be checked to see if it's empty before writing to it to prevent an overwrite. The example code below does this.

```
  /* Blocking call to send one character. */
  void USART0_sendChar(char data)
  {
    /* Wait for empty transmit buffer */
    while ( !(USART0_STATUS & USART_DREIF_bm ) )      ;

    /* Put data into buffer, sends the data */
    USART0_TXDATAL = data;
  }
```

This is a 'blocking call' because the function does not immediately return and if the USART failed to send the last byte, it will wait forever. A way to mitigate this is to add a timeout to the wait operation.

To receive a byte, just check the USART_RXCIF bit in the STATUS register and read the RXDATAL register if the bit is set. The example code below shows one way to do this.

```
  /* Blocking call to receive one character.
   * USART0_isChar() should be called and return 'true' before calling this function.
   */
  char USART0_recvChar()
  {
    /* Wait for data to be received */
    while ( !(USART0_STATUS & USART_RXCIF_bm) );

    /* Get and return received data from buffer */
    return USART0_RXDATAL;
  }

  // Check for character in Receive register
  bool USART0_isChar()
  {
    bool result = false;

    if( USART0_STATUS & USART_RXCIF_bm ) {
      result = true;
    }

    return( result );
  }
```

All of this code is in the SerialPoll files included in the ITMC_III_1.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

2. Analog-to-Digital Converter - ADC

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3209-Getting-Started-with-ADC-DS90003209.pdf
NOTE: ADC input resistance is approximately 14k ohms and must be considered when attaching a sensor or other voltage generating circuit.

This family of devices provides a 10-bit Analog to Digital Converter (ADC) with a maximum sample rate of 115 kHz. The ADC generates a digital number relative to the voltage level applied. 0x000 for zero volts to 0x3FF for an input equal to the Vref voltage. A configuration is also provided to measure the internal temperature of the device. This example will measure the voltage from a variable resistor and use three LEDs to show its position. An adc.c. and an adc.h file will be added to the project to contain the ADC specific code.

The green LED will light if the voltage is less that 1/2 Vref. The yellow LED will light if the voltage is between 1/2 and 3/4 Vref. The red LED will light if the voltage is greater than 3/4 Vref.

Figure III-2a shows the circuit used in this example. The internal timer will trigger a voltage measurement once per second. The three colored LEDs will show the relative position of the wiper of the variable resistor. Figure III-2b shows a typical buffered ADC circuit using a single supply op-amp.

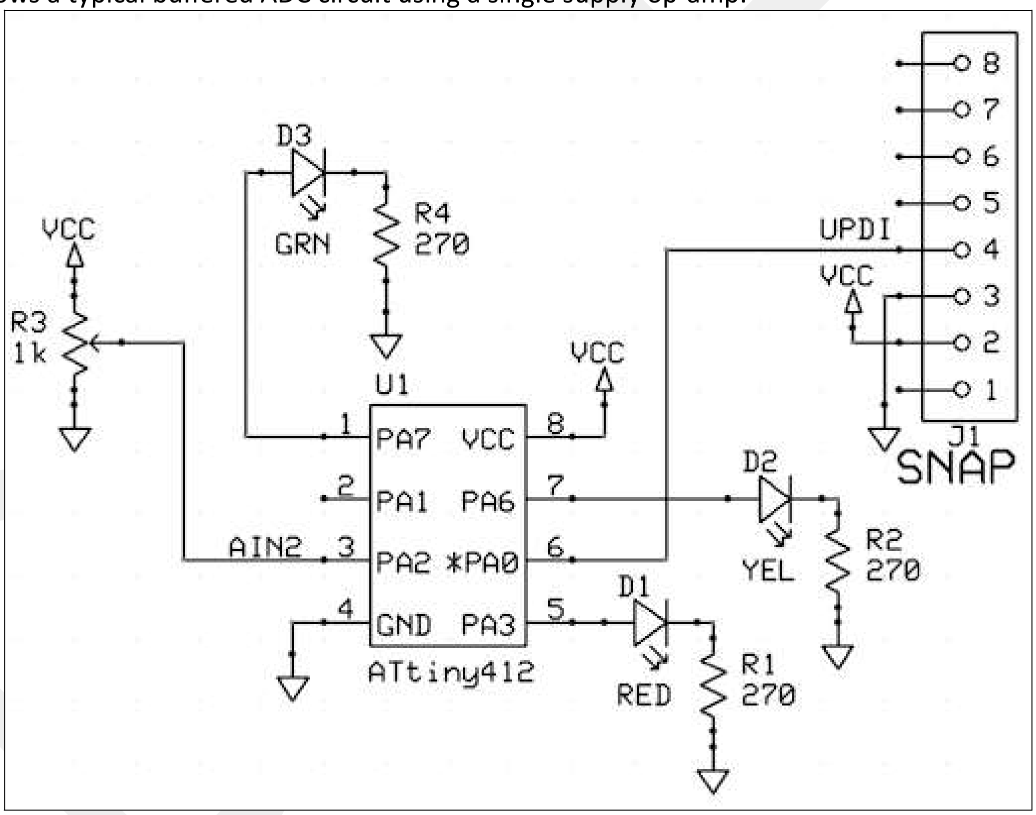


Figure III-2a

This circuit can be used to read the position of R3 and processor will determine which LED to light up based on the voltage measured.

a. Setting up the ADC

Section 30[2] describes the ADC. The ATtiny412 provides 6 channels for voltage measurement at a 10-bit resolution. Channel 2 (AIN2) is used in this example. Table 5-2[2] shows that AIN2 is connected to PA2.
To simplify things, only 8-bits of the ADC will be used. The CTRLA.RESSEL will be set to '1' for this mode.

```
ADC0.CTRLA = ADC_RESSEL_8BIT_gc;
```

The ADC will be used in free-running mode. No accumulation will be used, so the CTRLB register is not set.
This example will use 2.5V for Vref. This provides a wide range for both a Vdd of 3.6v or 4.5v. Set the VREF using the VREF.CTRLA register. The following code will do that.

```
VREF.CTRLA |= VREF_ADC0REFSEL_1_bm;        // set Vref to 2.5v.
```

Section 30.3.2.2[2] states that the maximum sampling frequency is 1.5 MHz for 10-bit resolution. With CLK_PER at 3.3 MHz (default), a DIV4 will be used. The SAMPCAP will also be set to '1'. This code example will do these settings. (see 30.5.3)[2]

```
ADC0.CTRLC = ADC0.CTRLC = 0x40 | ADC_REFSEL_VDDREF_gc | ADC_PRESC_DIV4_gc;
```

Next, set the MUXPOS register to select AN2 (PA2 pin) as the input to the ADC.

```
ADC0.MUXPOS = ADC_MUXPOS_AIN2_gc;
```

Lastly, enable the ADC by setting the Enable bit in the CTRLA register.

```
ADC0.CTRLA |= ADC_ENABLE_bm;        // enable the ADC.
```

The ADC is now configured to start measuring the input voltage when trigger by setting the STCONV bit in the COMMAND register.

```
ADC0.COMMAND = 0x01;
```

This bit will remain set while the ADC conversion process in in progress. Once the process sets it to '0', the data can be read from the Results register(s).

```
while (ADC0.COMMAND == 0x01);        // wait for completion.
voltage = ADC0.RESL;
```

This voltage is then tested to determine which LED to turn on. The following pseudo code shows one way to do this. Note: This is using 8-bit so maximum voltage is 0xFF.

If voltage < 0x80 then turn ON Green LED else turn it OFF.
If voltage >= 0x80 && voltage is < 0xC0 then turn ON YELLOW LED else turn it OFF.
If voltage >= 0xC0 then turn ON RED LED else turn it OFF.

Configure and control the two additional LEDs using the methods of Section II.3

To minimize the affects of the low input impedance of the ADC, a buffer can be used at the input as shown in Figure III-2b.

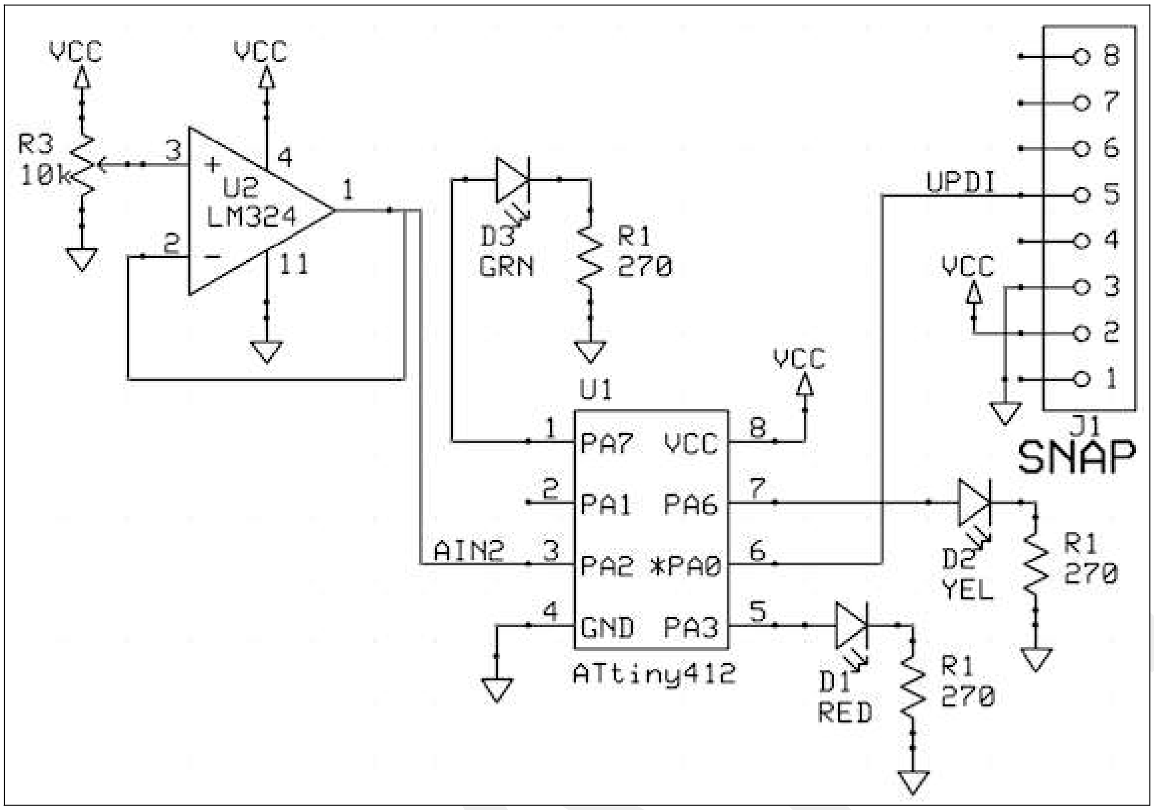


Figure III-2b

The LM324 is a single ended supply op-amp configured for unity gain and used as a buffer for the voltage from the resistor. It provides a low impedance drive for the low 14 k ohm input impedance of the ATtiny device so that the device will not affect the voltage coming from the resistor.

An example project using the code discussed is in the ITMC_III_2.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

3. Digital-to-Analog Converter - DAC

ref: https://ww1.microchip.com/downloads/en/DeviceDoc/TB3210-Getting-Started-with-DAC-90003210A.pdf

The DAC generates a voltage proportional to the digital value in its DATA register. This example configures it to provide a voltage to an output pin. Its output can also be used to provide a reference voltage to the Analog Comparator(AC) and the Analog-to-Digital Converter(ADC).

The DAC can only source 1ma of current. This example will require a VOM or DMM to read and display the DAC output voltage. The millis() function will be used to generate a sawtooth waveform by incrementing a counter and letting it roll over to zero. The counter value will be sent to the DAC for output. Figure III-2 will be used with the LED removed so that PA6 can be used as the DAC output. A dac.c and a dac.h file will be added to the project to contain the DAC specific code. Adjust the value of SAMPLE_DELAY to speed up or slow down the wave frequency.

a. Setting up the DAC

Section 31[2] describes the DAC. The tinyAVR 1-series devices provide an 8-bit DAC that supports an update rate of 350 kHz. The DAC can be used to provide a reference voltage to the Analog Comparator(AC) and the Analog-to-Digital Converter(ADC). This example configures it to provide a voltage to an output pin.
First, select the Voltage Reference(VREF) to set the upper output level of the DAC with the DAC0REFSEL bits in the VREF.CTRL0 register. The following code will do that.

```
// set both ADC and DAC Vref to 2.5v.
VREF.CTRLA |= VREF_ADC0REFSEL_1_bm | VREF_DAC0REFSEL_1_bm;
```

Now, enable the output pin and enable the DAC.

```
PORTA.DIR |= PIN6_bm;                    // set as OUTPUT
DAC0.CTRLA |= DAC_OUTEN_bm | DAC_ENABLE_bm;
```

These two lines should be put into an init_dac() function in the dac.c file.
The last step is to provide a function to set the DAC output value. The following code is a way to do that.

```
void set_dac_output(uint8_t value)
{
    DAC0.DATA = value;
}
```

An example of a simple count and output loop is shown below.

```
// Increment a counter and copy its value to the DAC to generate a voltage.
uint8_t count = 0;

while (1) {
    if( millis() > ledTime ) {
        ledTime = millis() + LED_DELAY;
        toggle_LED();
    }

    if( millis() > sampleTime ) {
        sampleTime = millis() + SAMPLE_DELAY;
        /* update the count. */
        ++count;
        /* update DAC value */
        set_dac_output(count);
    }
}
```

An example project using the code discussed is in the ITMC_III_3.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

4. Serial Peripheral Interface – SPI

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3215-Getting-Started-with-SPI-DS90003215.pdf

NOTE: HOST = MASTER = CONTROLLER and CLIENT = SLAVE = PERIPHERAL

4a. SPI – Controller

This example will require an Arduino board or similar device to act as a SPI Client device. The code will configure the ATtiny412 as a SPI Host and send ASCII characters to the Arduino to be printed out in the Serial Monitor window.

The SPI Host initiates a message transfer by asserting the /SS line for chip select. The Host then generates the required clock pulses on the SCK line while shifting data out though the MOSI line and/or receiving data through the MISO line. The Client SHOULD have its data ready to shift out before the /SS is asserted since the SCK pulses usually begin immediately after /SS is asserted. Figure 25-5 shows how the clock and data are related in each of the four possible SPI Data Transfer Modes. Mode 0 will be used in this example. The Host will also operate in NORMAL mode and will NOT be using interrupts.

Add a file spi.c to the Source File section of the project. New > newavr-main.c. Rename to spi. > Finish. Delete the int main(void) code block and add

```
void init_spi(void){ }
```

and

```
bool sendByte(uint8_t val){ }
```

functions to the file. Then add an spi.h file, New > C Header File, to the Header Files section to store the function prototypes.

Identify the MOSI, MISO, SCK, and /SS pins for the device being used. For the ATtiny412 SIOC14, these are:

Name	ID	Pin	Board Pin
MOSI	PA1	11	11
MISO	PA2	12	12
SCK	PA3	13	13
/SS	PA4	2	1

Configure the /SS pin as a OUTPUT. Place this code init_spi().

```
PORTA.DIR |= PIN4_bm;
```

In init_spi(), configure the device as a Master, set the pre-scaler to divide CLK_PER by 64 to get a 52 kHz SCK clock and enable the SPI interface. The following code should do this:

```
SPI0.CTRLA |= SPI_MASTER_bm | SPI_PRESC_1_bm | SPI_ENABLE_bm;
```

Also, disable Client Select so that /SS action does not change the SPI configuration.

```
SPI0.CTRLB |= SPI_SSD_bm;          // Default Mode 0.
```

Per 25.3.2.1.1[2], SPI0.INTFLAGS IF bit is set after transfer has completed. Test and reset this bit before attempting to send more data. Write data into the SPI0.DATA register. This will initiate data transfer IF the ENABLE bit is the SPI0.CTRLA register is set to ‘1’. This logic will be needed to send a byte. The following code, placed in sendByte(), should do this:

```
    bool result = false;

    if( SPI0.INTFLAGS & SPI_IF_bm ) {
        SPI0.INTFLAGS &= ~(SPI_IF_bm);      // clear IF flag
        SPI0.DATA = val;
        result = true;
    }

    return result;
```

Set up a simple loop to send an ASCII character to the Slave once a second and toggle the LED on PA7 each time a value is sent.

Configure the Arduino to act as a SPI Client to receive the data. (see Arduino Help > Reference > SPI)

The Arduino set up as a SPI Client will read its SPI Data In register and print the character received to the Serial Monitor window. An example of this is in the Arduino_SPI_Client folder, along with the code from this section in project ITMC_III_4a.X.zip, can be downloaded from "https://www.gameactive.org/dist/".

4b. SPI - Peripheral

This example will require an Arduino board or similar device to act as a SPI Host. The code will configure the ATtiny412 as a SPI Client and allow control of the LED and reading the switch state. Section 25[2] has information about the SPI – Serial Peripheral Interface for both Host and Client configurations.
NOTE: HOST = MASTER = CONTROLLER and CLIENT = SLAVE = PERIPHERAL

5. Two-Wire Interface - TWI (I2C)

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ApplicationNotes/ApplicationNotes/atmel-2565-using-the-twi-module-as-i2c-slave_applicationnote_avr311.pdf

This example will require an Arduino board or similar device to act as a Master I2C controller. The code will configure the ATtiny1614 as an I2C Slave at address 0x50 and allow control of the LED and reading the switch state. Figure 1 shows the connections with the Arduino supplying power.

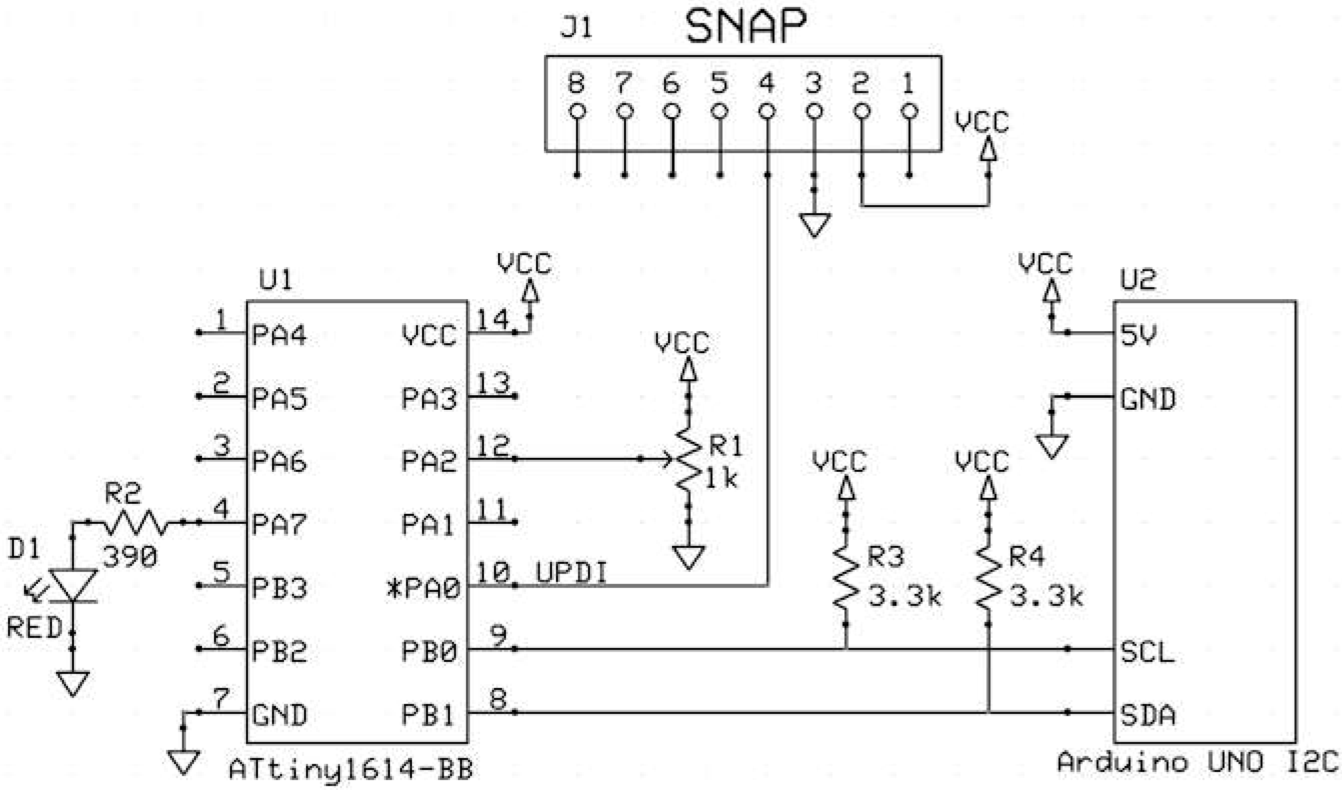


Figure 1

This example uses code for a register based I2C Slave. The source code is part of the ITMC_III_5.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

This code makes the device look like an I2C slave with 16 data registers. The I2C message format is to send the start register number with a Write (SDA_W) command followed by ‘n’ number of Read (SDA_R) commands where ‘n’ is the number of registers to read starting with the start register number. Register 0 is always set to 0. The other 15 registers can contain data. In this example, register 1 contains the upper 8-bits of the ADC value based on the position of the R1 wiper.

twiRegSlave uses a state machine architecture to service the `TWI0_SSTATUS` register following a `TWI0_TWIS` interrupt. The status register indicate the cause of the interrupt and follows the state transitions of various I2c messages allowing the system to extract data from them.

To Read a register, the I2C Master uses the following message format:

```
SDA_W  REGNUM  SDA_R  DATA [DATA] …
```

Example: Read Register 1

```
0x50_W  0x01  0x50_R  0xNN
```

Example: Read Registers 4-6

```
0x50_W  0x04  0x50_R  0xNN  0xQQ  0xRR
```

To Write to a register, the I2C Master uses the following message format:

```
SDA_W  REGNUM  DATA [DATA] …
```

Example: Write to Register 1

```
0x50_W  0x01 0xNN
```

Example: Write Registers 3-4

```
0x50_W  0x03  0xNN  0xQQ
```

The main() is a simple sample loop that updates the contents of register 1 based on TWI_DELAY. It also provides a 'heartbeat' using the LED at a rate set by LED_DELAY. After initialization, this is the loop code. The WDT is used to recover from I2C start-up synchronization mis-match.

```
// Read voltage. Set register 1 based on value.
while (1) {
    resetWDT();

    // Heartbeat
        if( millis() > ledTime ) {
        ledTime = millis() + LED_DELAY;
        toggle_LED();
    }

    if( millis() > ledTime ) {
        twiTime = millis() + TWI_DELAY;

        /* sample the input voltage. */
        trigger_adc();
        /* wait for conversion to complete and return voltage. */
        voltage = read_adc();

        // update data register 1.
        twiSetRegister(1, voltage);
    }
}
```

The Arduino is set up as an I2C Master. It will periodically send a read register 1 message to the Slave at 0x50 and print the return data in the Serial Monitor window. An example of this is in the Arduino_I2C_Master folder that can be downloaded from "https://www.gameactive.org/dist/".

6. Other Timer/Counters

a. TCB

ref: https://ww1.microchip.com/downloads/en/DeviceDoc/TB3214-Getting-Started-with-TCB-90003214A.pdf

The TCB counter is designed to measure frequency and pulse width of external waveforms. It also has PWM capabilities and can be used to monitor processing time of internal functions.

In this example, the TCB timer will be used to support the millis() function. This might be needed if the TCA timer was being used to generate multiple RC servo signals as it can easily generate six of them. Section 21[2] describe the many other modes of operation for this timer.

The set up is very similar to configuring TCA in Section II-4. TCB will now be used to increment the `totalMilliseconds` variable. The TCB configuration code can be added to the systimer.c and systime.h files.

```
/* *** TCB Configuration as NORMAL counter with interrupt for 1ms OVF *** */
void init_systime(void)
{
    // The CCMP register contains the TOP value of the counter.
    TCB0.CCMP = 0x0D04;       // 3,333 - 1 since the counter starts at 0.
    /* enable periodic interrupt interrupt */
    TCB0.INTCTRL = TCB_CAPT_bm;
    /* set mode */
    // TCB0.CTRLB = (default, mode is periodic interrupt.)
    // TCB0.CTRLA = (default, use CLK_PER 3.333 MHz)
}

/* TCB interrupt service routine. */
ISR(TCB0_INT_vect)
{
    ++totalMilliseconds;
    TCB0.INTFLAGS = TCB_CAPT_bm;   // clear the interrupt flag
}
```

An example project using the code discussed is in the ITMC_III_6a.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

b. TCD

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3212-Getting-Started-with-TCD-DS90003212.pdf

The TCD timer is designed for use in switching power supplies and motor controllers when paired dual MOSFETs or with half-bridge or full-bridge driver circuits.

This example will use the Two Ramp Mode described in section 22.3.3.2.2 of the data sheet[2]. The two outputs, WOA and WOB, will be used to independently control the brightness of two LEDs. This method could be used to control the power levels of two independent MOSFETs in a switch power supply or motor drive circuit.

The clock source will be CLK_PER and is selected by the CTRLA register CLKSEL[1:0] bits. CTRLA also contains the ENABLE bit, which is set to '1' to enable the counter. The following code sets up the CTRLA register.

```
TCD0.CTRLA = SYSCLK
```

The CTRLB register controls the Waveform Generation Mode. The following code configures the counter for Two Ramp mode.

```
TCD0.CTRLB = TWORAMP
```

In Two Ramp Mode, the period for WOA is set by the CMPACLR register and the period for WOB is set by the CMPBCLR register. The duty cycles are set by the CMPASET and CMPBSET registers respectively. (see Figure 22.-5[2]) The following code will set up the counter to generate two 50% duty cycle waves at 50 kHz.

Finally, turn on the timer by setting the ENABLE bit to '1'.

```
TCD0.CTRLA |= ENABLE
```

An example project using the code discussed is in the ITMC_III_6b.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

7. Watchdog Timer - WDT (and Sleep Controller - SLPCTRL)

The Watchdog Timer (WDT)(Section 19[2]) is very useful in recovering from a 'code lockup'. If it is not triggered within its timeout period, it will reset the system. So, it is typically triggered within the main loop.
The WDT can also be used for battery management. The WDT can reset a system that has been put in SLEEP mode where just about everything has been shut down to reduce power usage. The WDT is clocked by an independent ultra-low power oscillator. (section 19[2])

This example will use the Sleep Controller (SLPCTRL)(Section 11[2]) and WDT to blink an LED every minute. The number of blinks will be equal to the number of minutes the system has been running. It will number will reset back to 1 after 10 minutes.
The circuit in Figure 2 will be used. The switch will be used to start the timer.

a. SLPCTRL setup

"The content of the register file, SRAM and registers, is kept during sleep."(section11.2[2]). This is very important as it allow us to store data during Sleep and update that data after waking up.
This example will us the Power-Down(PDOWN) mode. SLPCTRL0.CTRLA will be configured for this and Sleep enable set to '1' to make Sleep available to be activated. The code below is one way to do this.

```
SLPCTRL.CTRLA = SLPCTRL_SMODE_1_bm | SLPCTRL_SEN_bm;
```

b. WDT setup

The WDT will be set to NORMAL mode with an 8s timeout. This is done by setting the PERIOD[3:0] bits to 0xB in CTRLA.

```
WDT.CTRLA = WDT_PERIOD_3_bm | WDT_PERIOD_1_bm | WDT_PERIOD_0_bm;
```

A read_Switch() function is added to io_ctlr to read the state of the push-button switch connected to PA1.

```
bool read_Switch()
{
    return(PORTA.IN & BUTTON_PIN_BM);
}
```

The main code has a test for a test word in memory to control the code after reset. If is not set, the system does the code initialization process else it continues to the regular code process. After the regular code process, the system is put to sleep with an assembly code directive '`asm("SLEEP");`'.
NOTE: At RESET, all register are cleared so they must be reconfigured each time. RAM is not affected UNLESS a variable initialization is used like '`int v = 0;`'. This causes 'v' to be set to '0' on reset. Using '`int v;`' does not initialize 'v' on reset. An example of all of this is shown below.

```
int main(void) {

    init_io();                  // set up IO pins.
    flash_LED();                // indicate a RESET

    // Initialize Sleep Control for Power-Down mode.       PDOWN | SEN
    SLPCTRL.CTRLA = SLPCTRL_SMODE_1_bm | SLPCTRL_SEN_bm;
    // Initialize WDT for 8 sec timeout Normal mode.
    WDT.CTRLA = WDT_PERIOD_3_bm | WDT_PERIOD_1_bm | WDT_PERIOD_0_bm;

    if (active != 0xAA55) {
        active = 0xAA55;
        cycleCounter = 0;
        while( PORTA.IN & read_Switch() );   // wait for Push Button press.
```

```
    } else {
        ++cycleCounter;
        for (int k = 0; k < cycleCounter; k++) {
            flash_LED();
            // Non-timer delay. Wait 65536 counts.
            for (uint16_t i = 0; i < 0xFFFF; i++) {
                asm("NOP");
            }
        }
    }

    asm("SLEEP");
}
```

An example project using the code discussed is in the ITMC07_WDT.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

8. Real Time Counter - RTC

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3213-Getting-Started-with-RTC-DS90003213.pdf

The RTC can be use to keep track real time since it runs even during Low-Power Sleep mode. Using the Internal Ultra Low-Power 32.768 kHz oscillator run though a divide by 32 counter with the 15-bit prescaler set for divide by 1024, the RTC can generate an interrupt each second.

9. Configurable Custom Logic - CCL

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3218-Getting-Started-with-CCL-DS90003218.pdf

The CCL is a Look-Up Table (LUT) based logic block with two LUTs. These can be treated as customizable three input logic gates. There is also a configurable latch on the output.

This example will use LUT0 as an AND gate and feed its output to LUT1 configured as an NOT gate. Two switches will provide the input and an LEDs will be use as the output indicators. Section 28 of the data sheet [2] describes the CLL feature.

10. Analog Comparator - AC

ref: https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ApplicationNotes/ApplicationNotes/TB3211-Getting-Started-with-AC-DS90003211.pdf

11. Brown Out Detector - BOD

Use a 1K pot to vary VCC.

In battery powered systems, a BOD can be useful to gracefully shut down the system if the VCC voltage drops too low for reliable operation.

12. Accessing EEPROM

The Nonvolatile Memory Controller (NVMCTRL) is used to access the EEPROM as described in section 9.[2] All AVR ATtiny series-1 devices have at least 64 bytes of EEPROM. This is nonvolatile memory that retains its data even if power is lost.

NOTE: When the EEPROM is ‘erased’, all memory bit will be ‘1’.

Section 6.2[2] shows the EEPROM memory to start at address 0x1400.

To write data to EEPROM, write the data into the buffer. Check the EEBUSY flag, then activate the Configuration Change Protection (CPU.CCP) key (SPM:0x9D) and set the WP command (0x01) in the NVMCTRL.CTRLA CMD[2:0] section.

The avr/eeprom.h library does all of this in its

```
eeprom_write_byte((* uint8_t) address, data)
```

and takes care of adjusting the memory address so the starting address is 0x0000. To read EEPROM, use

```
eeprom_read_byte((* uint8_t) address)
```

See the avr/eeprom.h file for other functions.

Using Figure III-1 for this example, the following code snippet will write thirteen characters into EEPROM and then read them all back at once and output them to the Serial port. The Data Visualizer can be used to monitor the action.

```
// Write 13 character into EEPROM, then read them all back and output to Serial.
while (1) {
    if( millis() > ledTime ) {
        ledTime = millis() + LED_DELAY;
        eeprom_write_byte((uint8_t *)count, val); // avr/eeprom takes care of setting
                                                  // CCP and Write address register.
        ++count;
        if( ++val > 'Z' ) {
            val = 'A';
        }
    }
    if(count > 12) {
        for (int i=0; i<13; i++) {
            epval = eeprom_read_byte((uint8_t *)i); // avr/eeprom lib ASSUMES 0x1400
                                                    // as a base address.
            /* Send an ASCII character out. */
            USART0_sendChar(epval);
        }
        USART0_sendChar(0x0D);
        USART0_sendChar(0x0A);
        count = 0;
    }
}
```

An example project using the code discussed is in the ITMC_III_12.X.zip project file that can be downloaded from "https://www.gameactive.org/dist/".

SECTION IV - Example Projects

1. 20 LEDs controlled with 5 I/O lines.

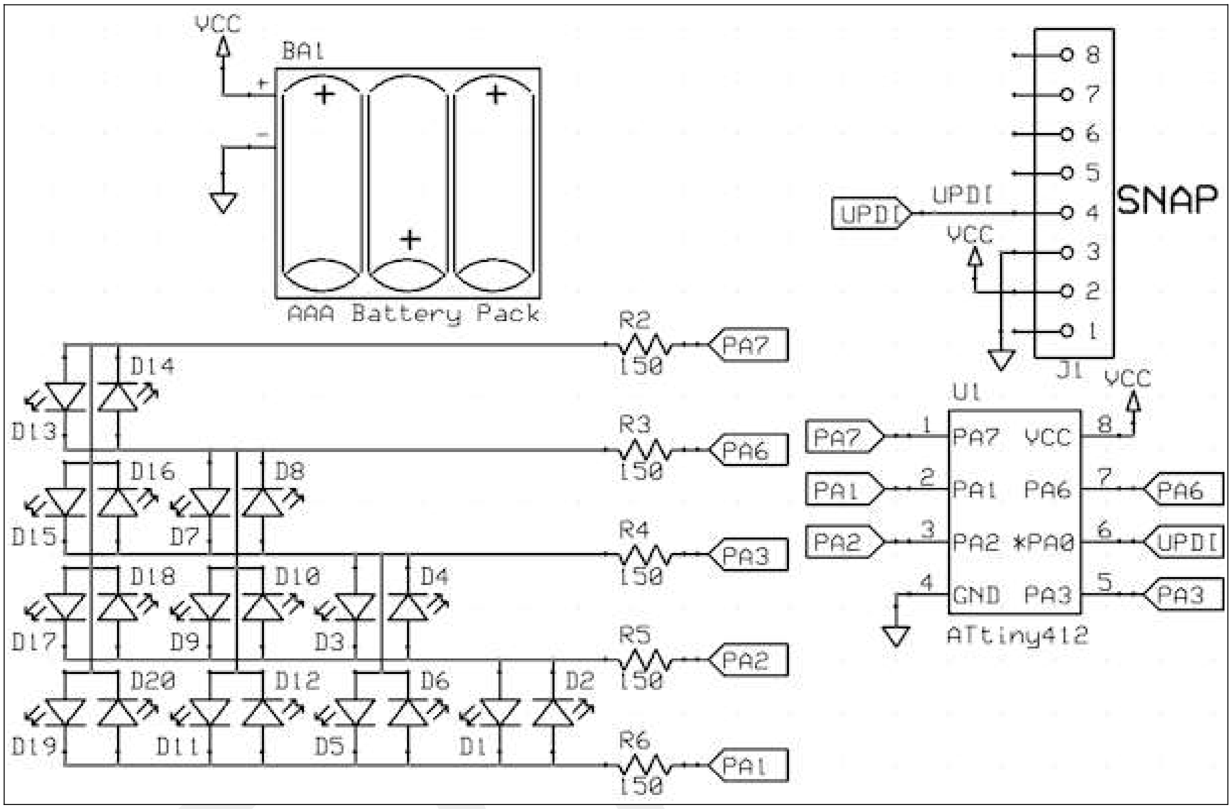


Figure IV-1

2. Remote Temperature Recorder

This example will use the Sleep Controller (SLPCTRL), ADC, USART, and WDT to build a device that measures temperature every 10 minutes and dumps the temperature data on command from the serial port.

3. Interfacing Push Button Switches with debounce

4. Interfacing Single and Multiple 7-Segment LED Displays

This example can also be used to control sixteen segment displays with little modification.

5. HC-SR04 Ultra-sonic Interface for Distance Sensing

6. Make an LED Flicker like a Candle

7. Interface to a Small DC Motor though a H-Bridge Controller

This example will also show how to control multiple motors.

8. Interface to Multiple RC Servos using TCA in Split Mode

SECTION IV- Power

Needs of the project. MCU 1.8vdc - 5.5vdc. External devices; LEDs, motors, servos, speakers

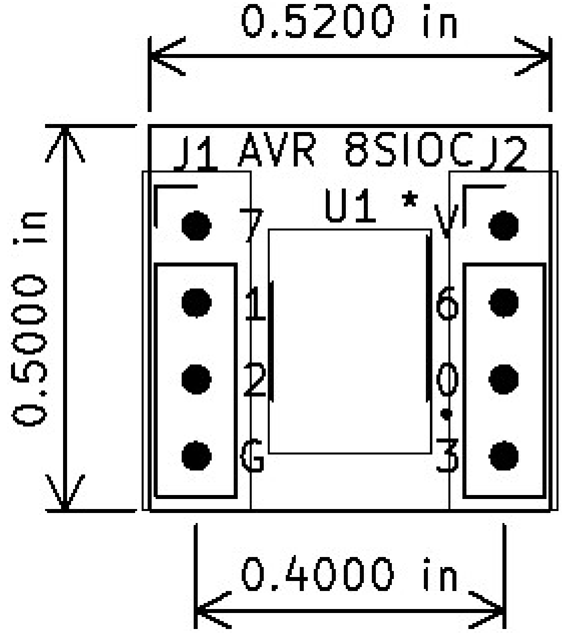
Voltage, peak current, capacity (mAh)

Batteries - 9v, AA, AAA, LiPo, coin cell.

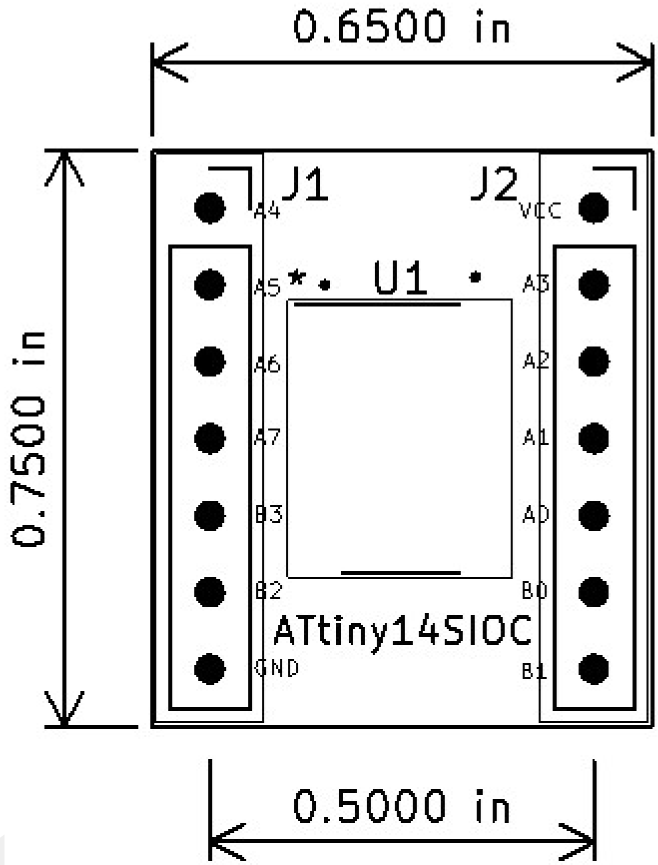
Wall Plug - 2.1mm female connector.

APPENDIX I - BOARD LAYOUTS

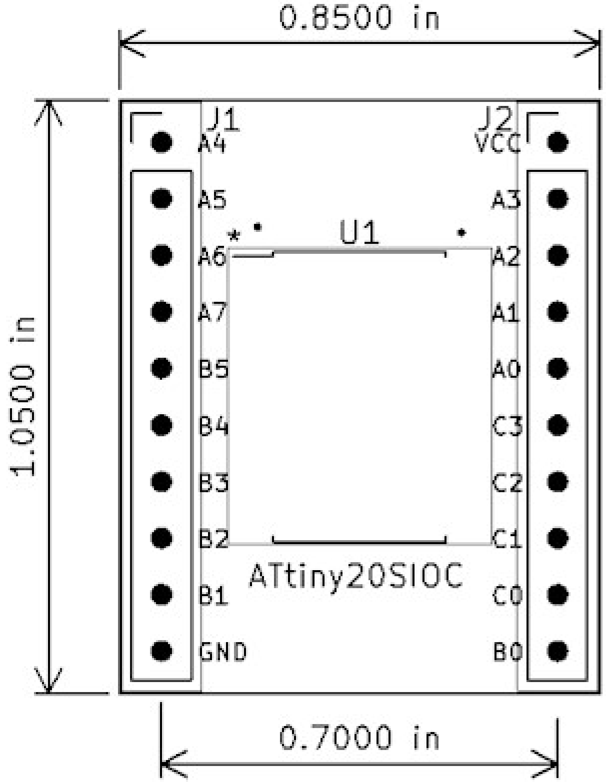
A. ATtiny8SIOC



B. ATtiny14SIOC



C. ATtiny20SIOC



APPENDIX III – Schematics

A. ATtiny8SIOC

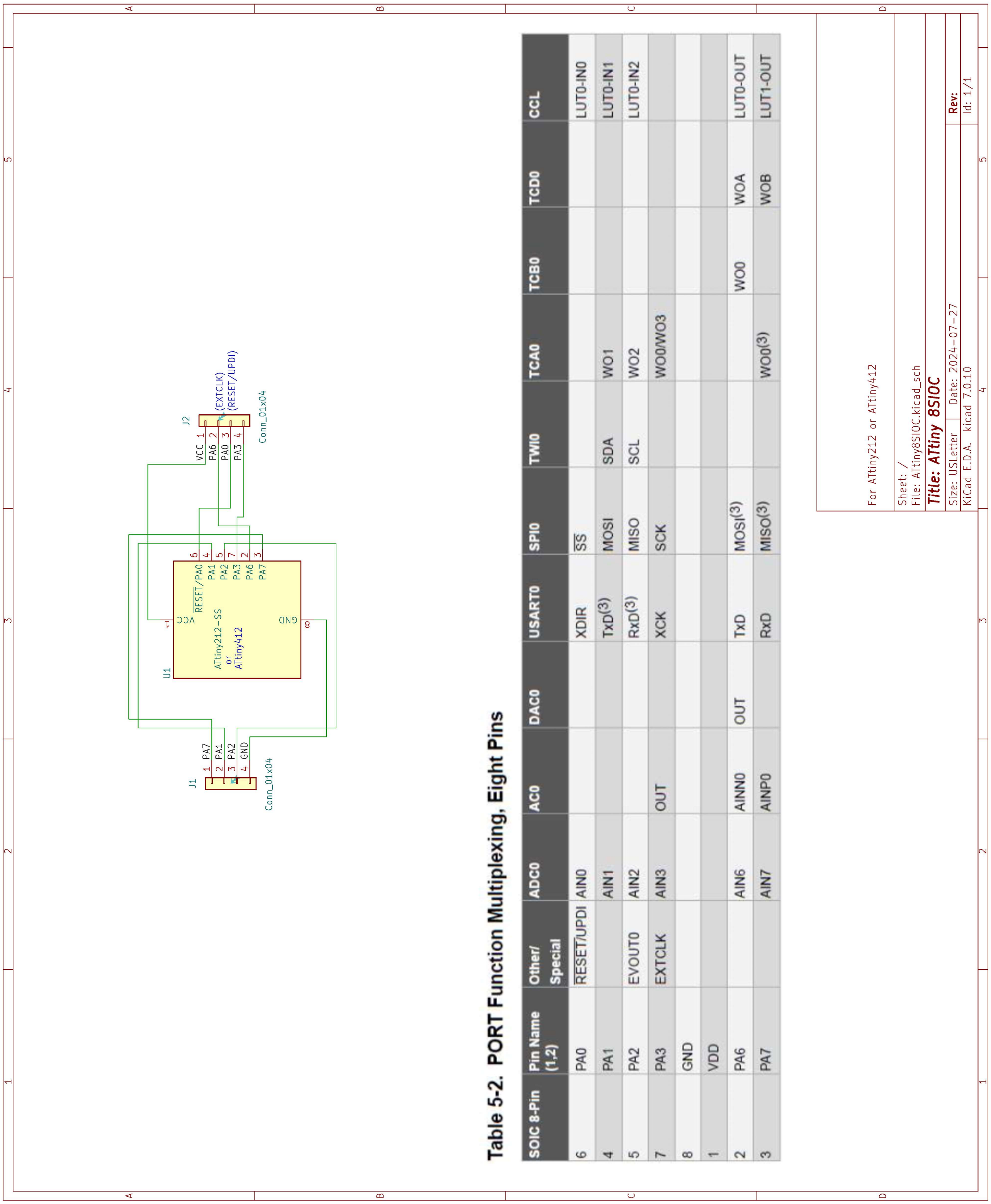
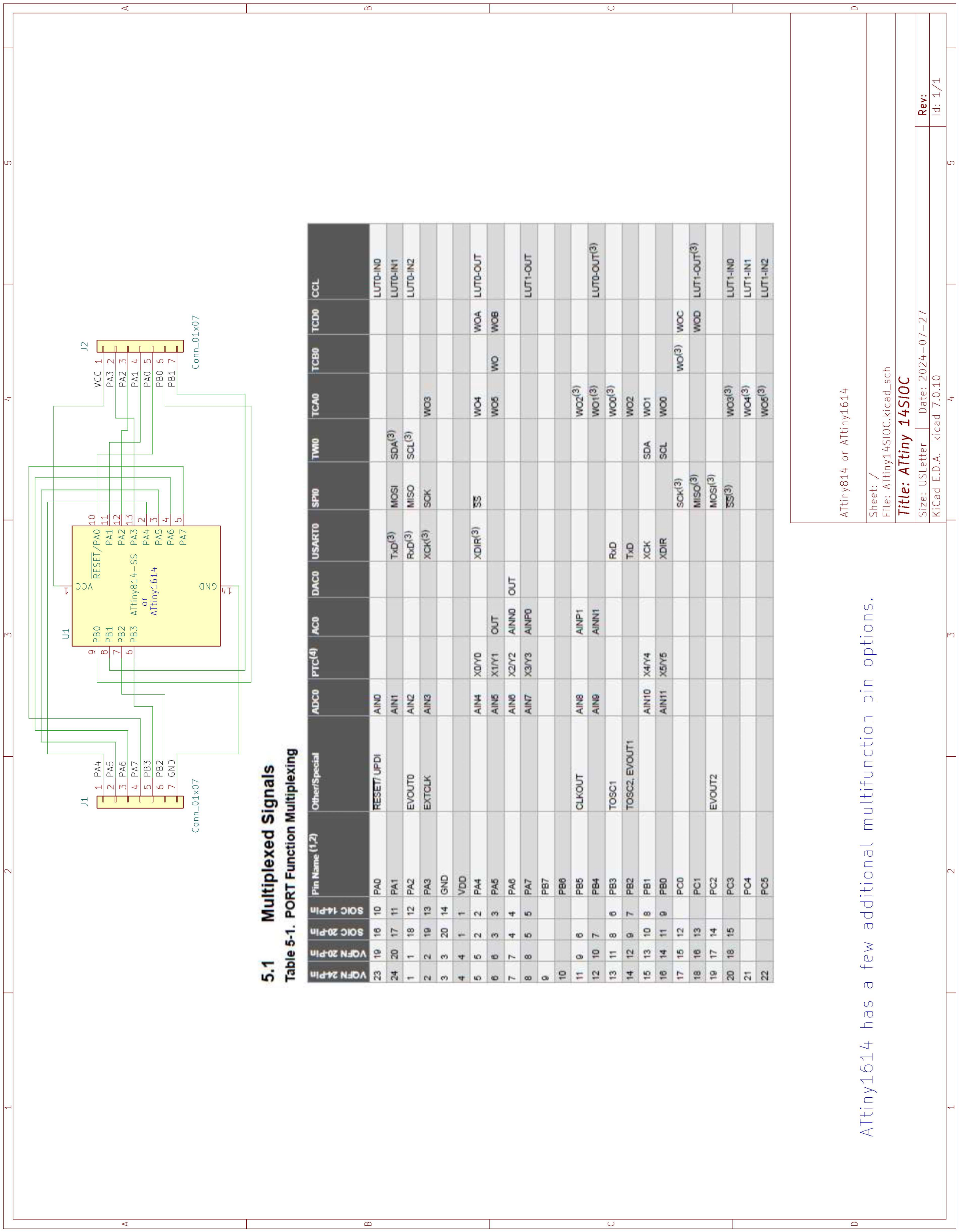


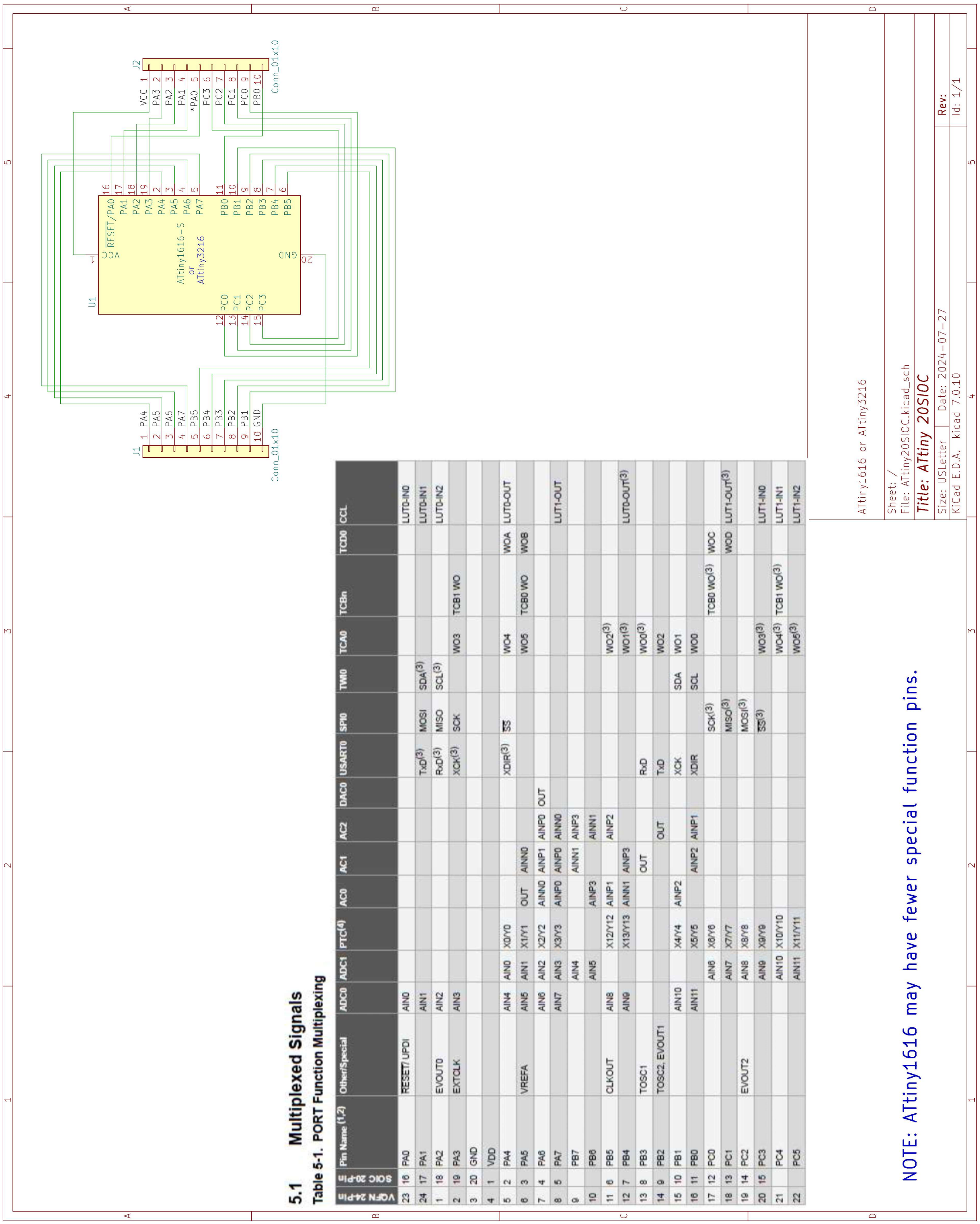
Table 5-2. PORT Function Multiplexing, Eight Pins

SOIC 8-Pin	Pin Name (1,2)	Other/ Special	ADC0	AC0	DAC0	USART0	SPI0	TWI0	TCA0	TCB0	TCD0	CCL
6	PA0	RESET/UPDI	AIN0			XDIR	SS					LUT0-IN0
4	PA1		AIN1			TxD(3)	MOSI	SDA	WO1			LUT0-IN1
5	PA2	EVOUT0	AIN2			RxD(3)	MISO	SCL	WO2			LUT0-IN2
7	PA3	EXTCLK	AIN3	OUT		XCK	SCK		WO0/WO3			
8	GND											
1	VDD											
2	PA6		AIN6	AINN0	OUT	TxD	MOSI(3)			WO0	WOA	LUT0-OUT
3	PA7		AIN7	AINP0		RxD	MISO(3)		WO0(3)		WOB	LUT1-OUT

B. ATtiny14SIOC



C. ATtiny20SIOC



5.1 Multiplexed Signals

Table 5-1. PORT Function Multiplexing

VQFN 24-Pin	SOIC 20-Pin	Pin Name [1,2]	Other/Special	ADC0	ADC1	PTC[4]	AC0	AC1	AC2	DAC0	USART0	SPI0	TWI0	TCA0	TCBn	TCD0	CCL
23	16	PA0	$\overline{\text{RESET}}$/ UPDI	AIN0													LUT0-IN0
24	17	PA1		AIN1							TxD[3]	MOSI	SDA[3]				LUT0-IN1
1	18	PA2	EVOUT0	AIN2							RxD[3]	MISO	SCL[3]				LUT0-IN2
2	19	PA3	EXTCLK	AIN3							XCK[3]	SCK		WO3	TCB1 WO		
3	20	GND															
4	1	VDD															
5	2	PA4		AIN4	AIN0	X0/Y0					XDIR[3]	$\overline{\text{SS}}$		WO4		WOA	LUT0-OUT
6	3	PA5	VREFA	AIN5	AIN1	X1/Y1	OUT	AINN0						WO5	TCB0 WO	WOB	
7	4	PA6		AIN6	AIN2	X2/Y2	AINN0	AINP1	AINP0	OUT							
8	5	PA7		AIN7	AIN3	X3/Y3	AINP0	AINP0	AINN0								LUT1-OUT
9		PB7			AIN4			AINN1	AINP3								
10		PB6			AIN5		AINP3		AINN1								
11	6	PB5	CLKOUT	AIN8		X12/Y12	AINP1		AINP2					WO2[3]			
12	7	PB4		AIN9		X13/Y13	AINN1	AINP3						WO1[3]			LUT0-OUT[3]
13	8	PB3	TOSC1					OUT			RxD			WO0[3]			
14	9	PB2	TOSC2, EVOUT1						OUT		TxD			WO2			
15	10	PB1		AIN10		X4/Y4	AINP2				XCK		SDA	WO1			
16	11	PB0		AIN11		X5/Y5		AINP2	AINP1		XDIR		SCL	WO0			
17	12	PC0			AIN6	X6/Y6						SCK[3]			TCB0 WO[3]	WOC	
18	13	PC1			AIN7	X7/Y7						MISO[3]				WOD	LUT1-OUT[3]
19	14	PC2	EVOUT2		AIN8	X8/Y8						MOSI[3]					
20	15	PC3			AIN9	X9/Y9						$\overline{\text{SS}}$[3]		WO3[3]			LUT1-IN0
21		PC4			AIN10	X10/Y10								WO4[3]	TCB1 WO[3]		LUT1-IN1
22		PC5			AIN11	X11/Y11								WO5[3]			LUT1-IN2

NOTE: ATtiny1616 may have fewer special function pins.

APPENDIX IV - UPDI Programmers

https://www.microchip.com/en-us/development-tool/ATATMEL-ICE

- Atmel-ICE $203.65

https://www.microchip.com/en-us/development-tool/PG164100

- MPLAB® Snap In-Circuit Debugger/Programmer $39.20 (Recommended)

https://ww1.microchip.com/downloads/en/DeviceDoc/ETN36_MPLAB%20Snap%20AVR%20Interface%20Modification.pdf

- ETN-36 MPLAB Snap AVR Interface Modification. Needed to program AVR devices.
- Add a 2.2k ohm resistor between VCC and UPDI pins as recommended in ETN-36.

https://www.microchip.com/en-us/development-tool/ev50j96a

- ATTINY3217 CURIOSITY NANO EVALUATION KIT includes a UPDI programmer and debugger. $14.99

See https://forums.adafruit.com/viewtopic.php?t=211365 on how to use these with MPLAB.

https://www.adafruit.com/product/5893

- Adafruit High Voltage UPDI Friend - USB Serial UPDI Programmer $9.95

https://www.adafruit.com/product/5879

- Adafruit UPDI Friend - USB Serial UPDI Programmer $6.95

References

ref1: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/avr-mcus
ref2: ATtiny212-214-412-414-416-DataSheet-DS40002287A.pdf
ref3: https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide
ref4: https://www.microchip.com/en-us/development-tool/PG164100 - SNAP Debugger/Programmer

APPENDIX V - Resources

ATtiny8SIOC.pcb ordered 01aug24, 3 for $1.30
ATtiny14SIOC.pcb ordered 01aug24, 3 for $2.40
ATtiny20SIOC.pcb ordered 03aug24, 3 for $4.45
(2)SNAP ordered 01aug24, 2 for $89.64 (see AVR mod sheet)