ROYAL ST. GEORGE'S COLLEGE

ADVANCED COMPUTER ENGINEERING SCHOOL

AVR OPTIMIZATION

HARDWARE ◆ SOFTWARE ◆ DESIGN ◆ MATHEMATICS

ACES IIIb takes our prospective engineers to the deepest accessible layer of the hardware architecture of the 8 bit AVR microcontroller family, specifically the ATmega328P and the ATtiny84. AVR's Register Level and Assembly Language Programming are explored.

ACE: ______________________________

Course: ICS4U (ACES IIIb)

Year: 2022-2023

Instructor: C. D'Arcy

Photo: X. Chin's, *Giant RGBW LED Matrix,* Spring 2022

Video: https://www.youtube.com/watch?v=KatHkq3PDNg

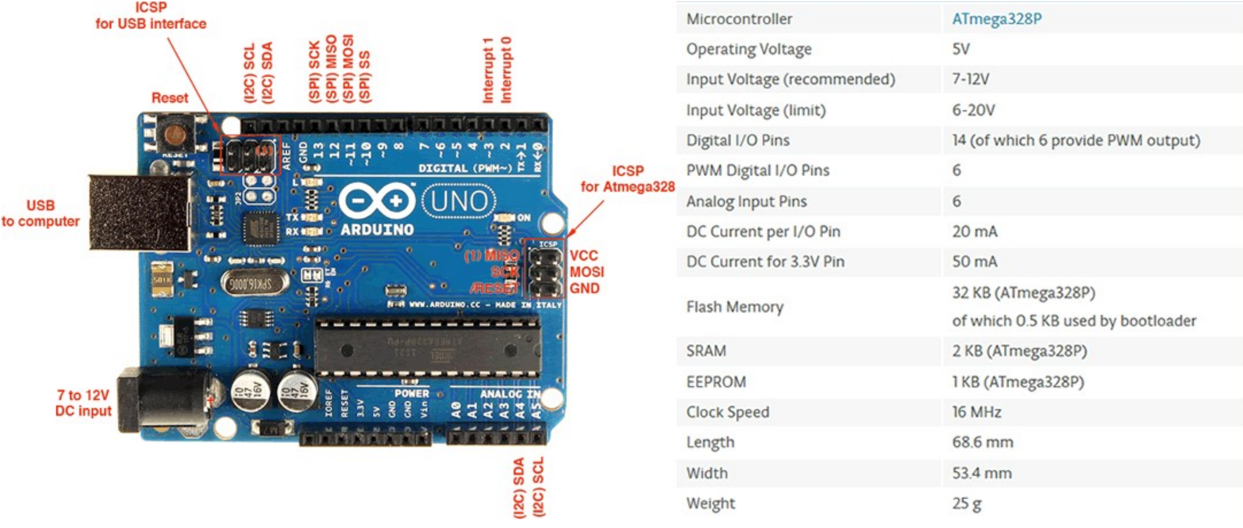
ATmega328P/Arduino Quick Reference

ATmega168/328 Pin Mapping

Arduino function	Pin name	Pin	Pin	Pin name	Arduino function
reset	(PCINT14/RESET) PC6	1	28	PC5 (ADC5/SCL/PCINT13)	analog input 5
digital pin 0 (RX)	(PCINT16/RXD) PD0	2	27	PC4 (ADC4/SDA/PCINT12)	analog input 4
digital pin 1 (TX)	(PCINT17/TXD) PD1	3	26	PC3 (ADC3/PCINT11)	analog input 3
digital pin 2	(PCINT18/INT0) PD2	4	25	PC2 (ADC2/PCINT10)	analog input 2
digital pin 3 (PWM)	(PCINT19/OC2B/INT1) PD3	5	24	PC1 (ADC1/PCINT9)	analog input 1
digital pin 4	(PCINT20/XCK/T0) PD4	6	23	PC0 (ADC0/PCINT8)	analog input 0
VCC	VCC	7	22	GND	GND
GND	GND	8	21	AREF	analog reference
crystal	(PCINT6/XTAL1/TOSC1) PB6	9	20	AVCC	VCC
crystal	(PCINT7/XTAL2/TOSC2) PB7	10	19	PB5 (SCK/PCINT5)	digital pin 13
digital pin 5 (PWM)	(PCINT21/OC0B/T1) PD5	11	18	PB4 (MISO/PCINT4)	digital pin 12
digital pin 6 (PWM)	(PCINT22/OC0A/AIN0) PD6	12	17	PB3 (MOSI/OC2A/PCINT3)	digital pin 11(PWM)
digital pin 7	(PCINT23/AIN1) PD7	13	16	PB2 (SS/OC1B/PCINT2)	digital pin 10 (PWM)
digital pin 8	(PCINT0/CLKO/ICP1) PB0	14	15	PB1 (OC1A/PCINT1)	digital pin 9 (PWM)

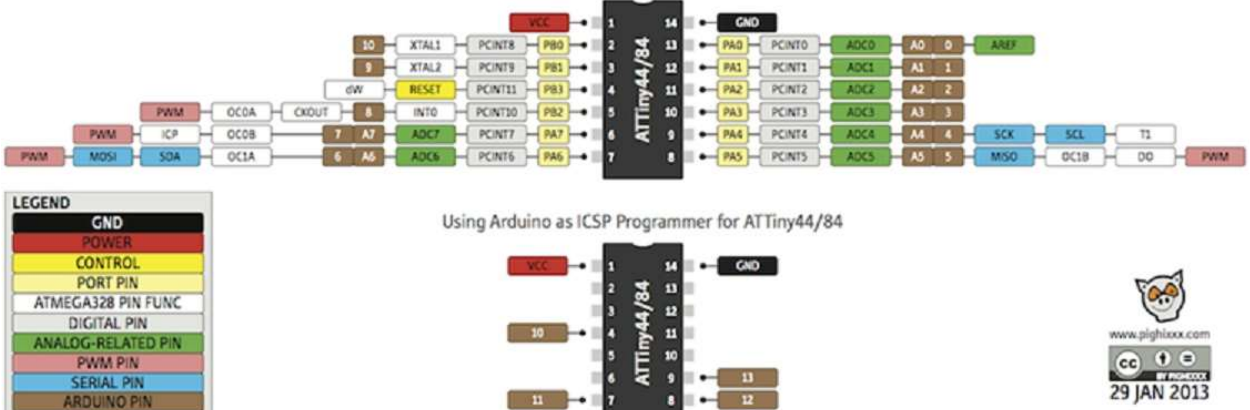
Digital Pins 11,12 & 13 are used by the ICSP header for MISO, MOSI, SCK connections (Atmega168 pins 17,18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

https://www.arduino.cc/en/Main/ArduinoBoardUno

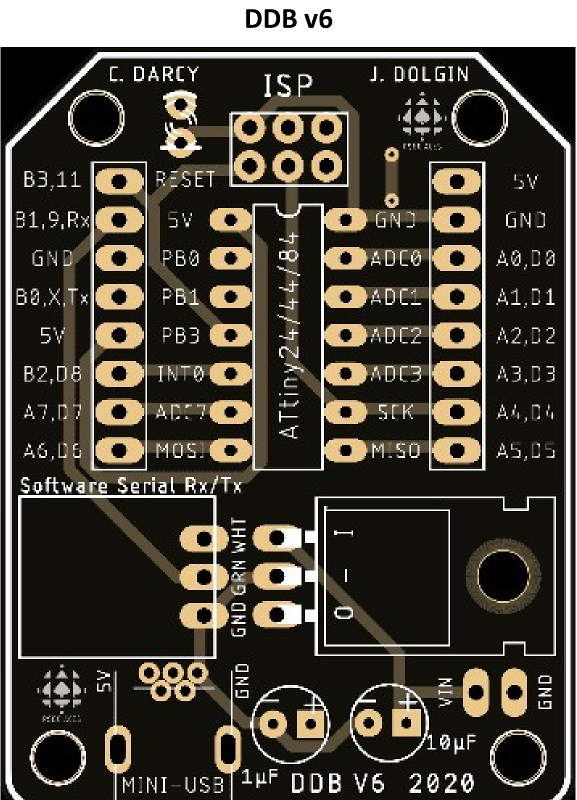


Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
Length	68.6 mm
Width	53.4 mm
Weight	25 g

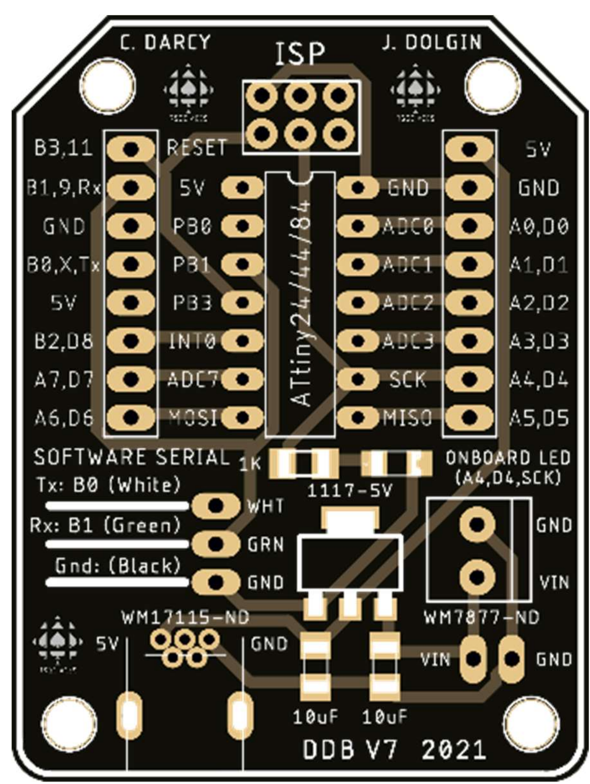
ATtiny84/DDB Quick Reference



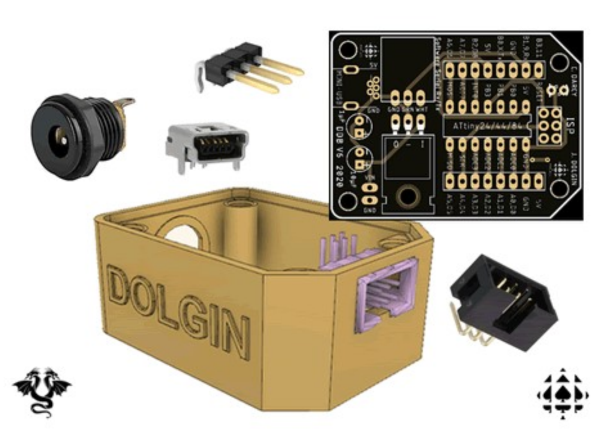
DDB v6



DDB v7



Dolgin Development Board: Parts List and Encasement



PARTS TABLE		
#	DESCRIPTION	SUPPLIED.
1	AVR ATtiny84 MCU (Microchip)	S
1	IC Socket 14 Pin (CNC Tech)	S
1	USB MINI B Connector (Molex)	S
1	3 POS RA (kinked) Conn. Hdr (Molex)	S
2	1×8 Female Header (Sullins)	S
1	10 μF Electrolytic Capacitor (short)	S
1	1 μF Electrolytic Capacitor (short)	S
1	2.1 mm Power Jack (Schurter)	S
1	2×3 Shrouded ISP Header (Wurth)	S
1	Dolgin Development Platform Board V6 PCB	S
1	Dolgin Development Platform Case with Insets	S
4	5mm M3 Nylon Screws (McMaster-Carr)	S
1	LM7805 Voltage Regulator	X
2	Power (Red) and GND (Black) hookup wire	X

Live Parts Links: http://darcy.rsgc.on.ca/ACES/TEI4M/2021/DDPv6.html

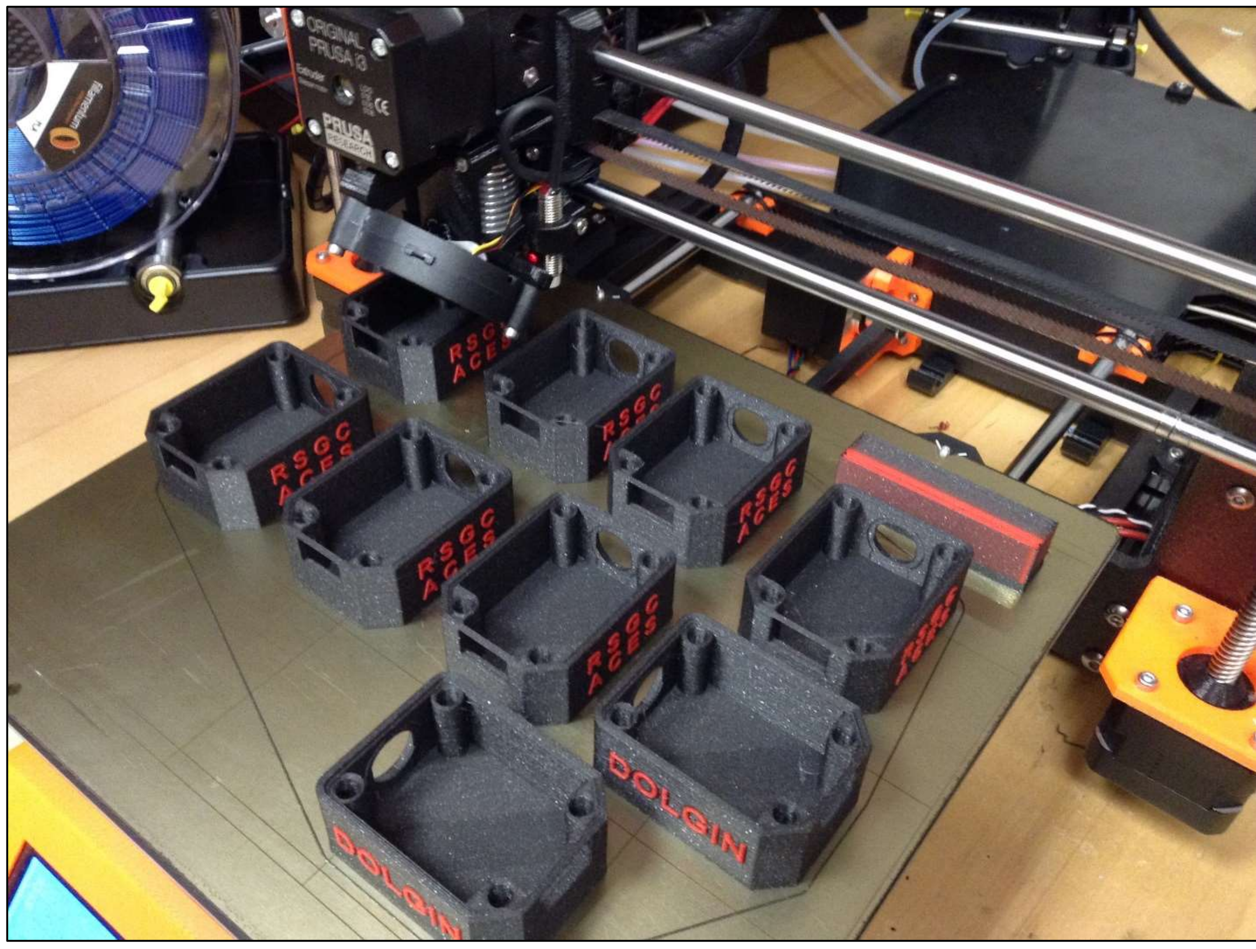


Table of Contents

ATMEGA328P/ARDUINO QUICK REFERENCE .... **I**

ATTINY84/DDB QUICK REFERENCE .... **III**

DOLGIN DEVELOPMENT BOARD: PARTS LIST AND ENCASEMENT .... **IV**

RSGC ACES ATTINY84 BREADBOARD DEVELOPMENT PLATFORM .... **IX**

SELECTION OF RECENT ICS4U ISPS .... **X**

0 INTRODUCTION **1**

0.0 REGISTER-LEVEL (RLP) AND ASSEMBLY LANGUAGE (ALP) PROGRAMMING? 1
0.1 EMBEDDED SYSTEMS: ELIMINATING THE MIDDLE MAN 2
0.2 DOLGIN DEVELOPMENT PLATFORM 3
0.3 BIT CODING GYMNASTICS 3
0.3.0 Setting a Bit 3
0.3.1 Clearing a Bit 3
0.3.2 Inverting a Bit 3

1 AVR MEMORIES **4**

1.0 FLASH PROGRAM FLASH (PROGMEM) 4
1.1 STATIC RAM (SRAM) 5
1.1.0 32 Private General Purpose (GP) Registers (0x00-0x1F) 5
1.1.1 64 I/O Registers (0x20-0x5F) 6
1.1.1.0 Digital I/O Registers (Ports) (PINx, DDRx, PORTx) 6
1.1.1.0.0 ATmega328P Digital I/O Registers (Ports) 6
1.1.1.0.1 ATtiny84 I/O Registers (Ports) 6
1.1.1.0.2 DDRx 7
1.1.1.0.3 PORTx 7
1.1.1.0.4 PINx 7
1.1.1.2 Stack Pointer (SPH and SPL) 8
1.1.1.3 Status Register (SREG) 8
1.1.2 160 Extended I/O Registers (0x60-0xFF) 10
1.1.3 SRAM (Heap and Stack) (0x??-RAMEND) 10
1.1.3.0 Heap 10
1.1.3.1 (System) Stack 11
1.2 EEPROM 11
1.3 PREDEFINES (.H AND .INC) 12

2 INTERRUPTS **13**

2.0 INTERRUPT VECTOR TABLE (IVT) 13
2.0.0 ATmega328P IVT 13
2.0.1 ATtiny84 IVT 14
2.1 AVOIDING CONFLICTS WITH THE IVT IN ASSEMBLY LANGUAGE 14
2.1.0 Interrupt Priorities 14
2.2 RESET INTERRUPT 15
2.2.0 MCUSR (Reset) Register 15

2.2.1 Rotary Encoder on RSGC ACES Breakout Board *15*
2.3 EXTERNAL INTERRUPTS 16
2.3.0 ATmega328P External Interrupt Registers *16*
2.3.1 ATtiny84 External Interrupt Registers *16*
2.4 PIN CHANGE INTERRUPTS 17
2.4.0 ATmega328P Pin Change Interrupt Control Register *17*
2.4.1 ATtiny84 General Interrupt Mask Register *17*
2.4.2 ATtiny84 Pin Change Mask Registers *17*

3 TIMER/COUNTERS 18

3.0 ATMEGA328P 18
3.0.0 ATmega328P Timer/Counter0 Modes *18*
3.0.1 ATmega328P Timer/Counter1 Modes *19*
3.0.2 ATmega328P Timer/Counter2 Modes *19*
3.0.3 ATmega328P Pulse Width Modulation (PWM) with AnalogWrite() *20*
3.0.3.0 Scope Trace of an AnalogWrite() PWM Waveform 20
3.0.4 Atmega328P Timer/Counter1 Registers *21*
3.0.5 Atmega328P Timer/Counter 1 Normal Mode 0 .... *22*
3.1 ATTINY84 23
3.1.0 ATtiny84 Timer/Counter0 Modes *23*
3.1.1 ATtiny84 Timer/Counter1 Modes *24*
3.1.2 ATtiny84 Pulse Width Modulation (PWM) with AnalogWrite() *24*
3.1.3 ATtiny84 Timer/Counter Registers *25*
3.2 ATTINY85 TIMER APPLICATION: FUNCTION GENERATOR 25
3.3 ACCESSING 16-BIT REGISTERS 26

4 ADC: ANALOG TO DIGITAL CONVERSION 27

4.0 ANALOG COMPARATOR 27
4.1 DAC: DIGITAL TO ANALOG CONVERSION (DAC) 28
4.2 SUCCESSIVE APPROXIMATION 28

5 PREPARATIONS FOR AVR ASSEMBLY LANGUAGE PROGRAMMING (AALP) 29

5.0 DEVELOPMENT PREPARATIONS 30
5.0.0 Hardware: Atmel/Microchip AVR Microcontrollers *30*
5.0.0.0 Peripheral Integration 30
5.0.1 Software Development Tools *31*
5.0.1.0 Integrated Development Environment: Atmel Studio 7 31
5.0.1.1 Operating System: Windows 10 32
5.0.1.2 Programmer: Atmel ICE 35
5.1 MICROCHIP'S ONLINE REFERENCE 35
5.2 NEW ATMEL STUDIO PROJECT 36
5.3 YOUR FIRST PROJECT: BLINK 37
5.3.0 Simulator *37*
5.3.1 Hardware *37*
5.3.2 Software *38*
5.3.2.0 Source Code Appearance 38
5.3.2.1 Assembly Source Code 38

5.3.3 Debugging Blink.asm 39
1.3.3.0 Stepping and Breakpoints 39
5.4 ATMEGA328P FEATURES 40
5.5 PERIPHERALS 41
5.6 AVR CENTRAL PROCESSING UNIT (CPU) 41
5.7 PACKAGE TYPES 43
5.7.0 Digikey: Ordering 43
5.7.1 Digikey: SchemeIt 44
5.8 INTERESTING EXERCISES 45
5.8.0 Delay Calculator 45
5.8.1 Traffic Light 45
5.8.2 RGB LED 45
5.8.3 Questions 45
5.9 JUST BEFORE WE START: C 46

6 AALP: AVR ASSEMBLY LANGUAGE PROGRAMMING 47

6.1 ASSEMBLY CODE ORGANIZATION 47
6.2 REUSABLE BUILDING BLOCKS 48
6.3 BASIC INSTRUCTIONS BY FUNCTION 49
6.3.0 Register Setting 49
6.3.1 Copying 49
6.3.2 Adding 49
6.3.3 Subtracting 50
6.3.4 Shift & Rotate 50
6.3.5 Binary 51
6.3.6 Bit Manipulation 51
6.3.7 Compare 51
6.3.8 Jump: Unconditional 51
6.3.9 Skip: Conditional 51
6.3.10 Branch Instructions 52
6.4 SIGNED REPRESENTATION OF NUMBERS 53
6.4.0 Two's Complement 53
6.5 EXPRESSIONS 54
6.5.0 Operators 54
6.5.1 Expression Separation Functions 55
6.6 VARIABLES 56
6.6.0 Variable Use in SRAM 56
6.6.1 Variable Use in Program Flash and EEPROM 56
6.6.1.0 .DB 57
6.6.1.1 .DW 57
6.6.1.2 .DD 57
6.6.1.3 .DQ 57
6.6.1.4 Example: Variable in Flash 58
6.6.1.4.0 Questions 58
6.7 ARRAYS 59
6.7.0 C Array Example 59
6.7.0.0 Comments and Observations from C Array Example 59

6.7.1 Data Indirect Addressing Modes *60*
6.7.2 Assembly Example *61*
2.7.2.0 Comments and Observations from Assembly Array Example 61
6.8 IF…THEN…ELSE 62
6.9 LOOP 63
6.9.0 for Loop *63*

7 AALP: ARITHMETIC AND MATHEMATICS 64

7.0 TERMINOLOGY: OVERFLOW AND UNDERFLOW 64
7.1 ADDING OR SUBTRACTING ONE FROM A REGISTER 65
7.2 MULTIPLYING AND DIVIDING A SINGLE BYTE BY A POWER OF 2 65
7.2.0 Multiplying a Single Byte by a Power of 2 .... *65*
7.2.1 Dividing Two-Byte (Word) Dividend by a Power of 2 .... *66*
7.3 BYTE ARITHMETIC 67
7.3.0 Byte Addition with Overflow (Carry Flag) *67*
7.3.1 Byte Subtraction with Underflow (Carry Flag) *67*
7.3.2 Unsigned Byte Multiplication with the MUL Instruction *68*
7.3.3 Signed Byte Multiplication with the MULS Instruction *68*
7.3.4 Byte Division *68*
7.4 ARITHMETIC WITH MULTI-BYTE OPERANDS 69
7.4.0 Two Dedicated Word Instructions: ADIW and SBIW *69*
3.4.1 Preparing Multi-Byte Operands *70*
3.4.1.0 Applicable Byte Functions 70
3.4.2 Adding Two Words *70*
3.4.3 Subtracting Two Double Words *71*
3.4.4 Multiplying two Words with the MUL Instruction *72*

8 AALP: AVR ASSEMBLY LANGUAGE PROGRAMMING WITHIN THE ARDUINO IDE 73

8.0 INLINE ASSEMBLY 73
8.0.0 Blink *73*
8.0.1 Blink Without Delay *74*
8.1 PURE ASSEMBLY 76
8.1.0 Blink *76*

APPENDICES 77

A DEVELOPMENT ENVIRONMENTS 77

A.0 INTEGRATED 77
A.0.0 Arduino IDE *77*
A.0.1 ATMEL Studio 7 (Windows) *78*
A.0.2 Crosspack (Mac) *78*
A.0.3 Atom and PlatformIO (Cross-Platform) *78*
A.0.4 WinAVR (Windows) *78*
A.0.5 AVR-Eclipse *78*
A.1 STANDALONE 79
A.1.0 TextMate (Mac) *79*
A.1.1 Notepad++(Windows) *79*

A.1.2 Programmers Notepad (Windows) *79*

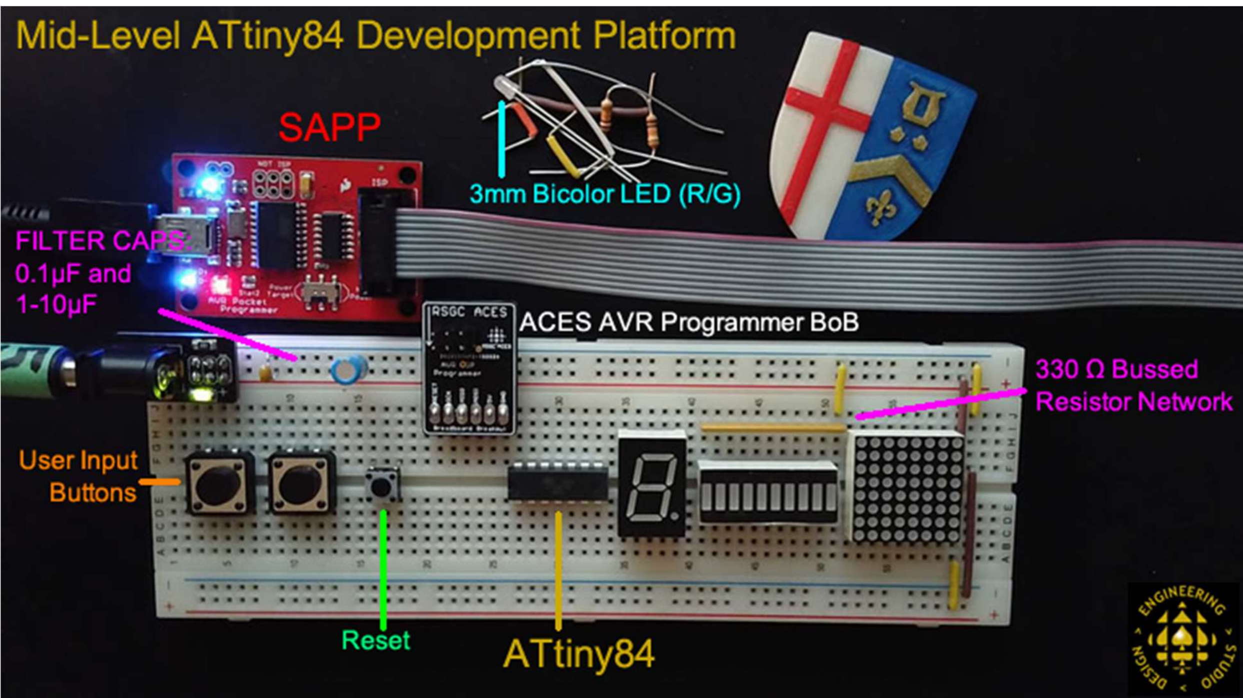
B SOFTWARE: GNU TOOLCHAIN 80

B.0 GCC 80
B.1 GNU BINUTILS 80
B.1.0 avr-as *81*
B.1.1 avr-ld *81*
B.2 AVR-LIBC 81
B.3 BUILDING SOFTWARE 81
B.4 AVRDUDE 81

C AVR ASSEMBLY REFERENCE 82

C.0 STATUS REGISTER (FLAGS), REGISTER AND INSTRUCTION OPERANDS 82
C.1 PROGRAM AND ADDRESSING MODES 82
C.2 REGISTER (GP, I/O & EXTENDED I/O) SUMMARY 82
C.3 FREQUENTLY USED AVR-AS DIRECTIVES 82
C.4 INTERRUPT VECTOR TABLE 82
C.5 INSTRUCTION SET 83
C.5.0 Summary of Instructions *83*
C.5.1 Detailed Instruction Set *83*

RSGC ACES ATtiny84 Breadboard Development Platform



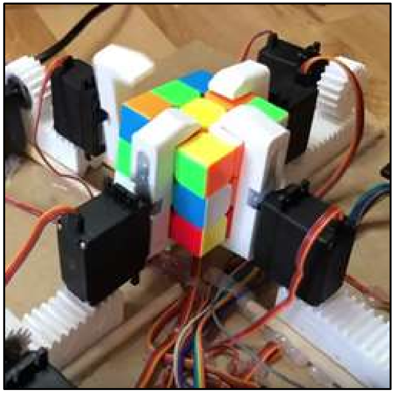
Selection of Recent ICS4U ISPs

Seb Atkinson: Dolgin/Atkinson Dev Board v8
https://www.youtube.com/watch?v=SYQoG84IRUQ

Jackon Shibley: Rocket Guidance System
https://www.youtube.com/watch?v=Jd08QXdUqzw

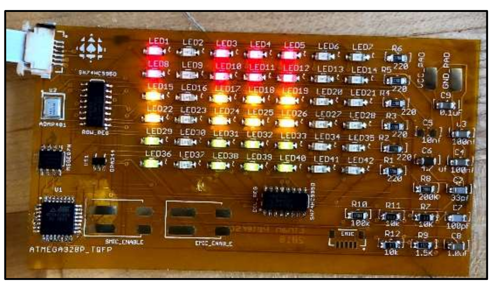
Adam Goldman: Smart Soap Dispenser
https://www.youtube.com/watch?v=3JjT-ef_25w

Jasper Schaffer: Rubik's Cube Solver
https://www.youtube.com/watch?v=hqTbJypHeqQ



Ethan McAuliffe: Photophone
https://www.youtube.com/watch?v=s8sXL5Ja8Gs

Ethan Peterson: Flex Equalizer
http://portfolio.petetech.net/flex-equalizer/



0 Introduction

Our second course within the ACES program, ICS3U, focuses largely on an introduction to various interfacing techniques and devices under monitoring and control of an AVR microcontroller. On the hardware side, the Arduino UNO offers beginners easy access to the ports and peripherals of the ATmega328P. On the software side, the Arduino IDE offered programmers an enhanced subset of ANSI C which can reasonably be referred to as Arduino C. Your own code, coupled with open source, off-the-shelf component libraries, formed the basis of your programs or *sketches* as they're referred to.

Our third course, ICS4U, takes engineering-minded Georgians behind the curtain, down to the lower levels of hardware and software concepts where the efficiencies and optimization of embedded systems are best achieved. To support your studies in this course you may wish to take a quick scan of the video tutorial offerings Vegard Wollan, co-inventor of AVR, provides on his YouTube Channel,

https://www.youtube.com/playlist?list=PLtQdQmNK_0DQgr3A3C6AEHp6DggewClmM

0.0 Register-Level (RLP) and Assembly Language (ALP) Programming?

My preferred reasons for introducing you to this curriculum are that it elevates you to an unparalleled level of embedded software competency. There's (almost) nothing between you and the CPU that is executing your code and, finally, this knowledge will give you a head start on your university courses and advantage in your internship interviews.

Here's the justification in a frame from an informative online slide show.

https://www.slideshare.net/rsamurti/l10-assemblylanguageprogrammingofatmega328-p

0.1 Embedded Systems: Eliminating the Middle Man

Like most things in life, coding involves tradeoffs. The high-level C, Java, or Python programmers get to express themselves in an English-like language with little to no regard for the underlying hardware that the code will be executed on. This mindset ranges anywhere from a missed opportunity to an outright problem for the *Embedded Systems* engineer.

Consider the ubiquitous `Blink` sketch in C that high-level coders are quite familiar with.

```
1  /*
2   * CBlink.c
3   *
4   * Created: 8/17/2018 9:07:44 AM
5   *  Author: Chris Darcy
6   */
7  #include <avr/io.h>
8  #define F_CPU 16000000UL  // 16 MHz
9  #include <util/delay.h>
10 int main(void)
11 {
12     DDRB |= 1<<PB5;
13     while(1)
14     {
15         PORTB ^= 1<<PB5;
16         _delay_ms(1000);
17     }
18 }
```

`rsgcaces > AVROptimization > CBlink.c`

Now, here's a low-level assembly language view of the same `Blink` sketch that is actually flashed into your MCU,

```
    DDRB |= 1<<PB5;
00000040  SBI 0x04,5        Set bit in I/O register
        PORTB ^= 1<<PB5;
00000041  LDI R25,0x20      Load immediate
--- No source file -------------------------------------------------------------
00000042  IN R24,0x05       In from I/O location
00000043  EOR R24,R25       Exclusive OR
00000044  OUT 0x05,R24      Out to I/O location
--- c:\program files (x86)\atmel\atmel toolchain\avr8 gcc\native\3.4.1061\avr8-gnu-toolchain\avr\include\util/delay.h
    __builtin_avr_delay_cycles(__ticks_dc);
00000045  SER R18         Set Register
00000046  LDI R19,0xD3      Load immediate
00000047  LDI R24,0x30      Load immediate
00000048  SUBI R18,0x01     Subtract immediate
00000049  SBCI R19,0x00     Subtract immediate with carry
0000004A  SBCI R24,0x00     Subtract immediate with carry
0000004B  BRNE PC-0x03      Branch if not equal
0000004C  RJMP PC+0x0001        Relative jump
0000004D  NOP         No operation
0000004E  RJMP PC-0x000C        Relative jump
```

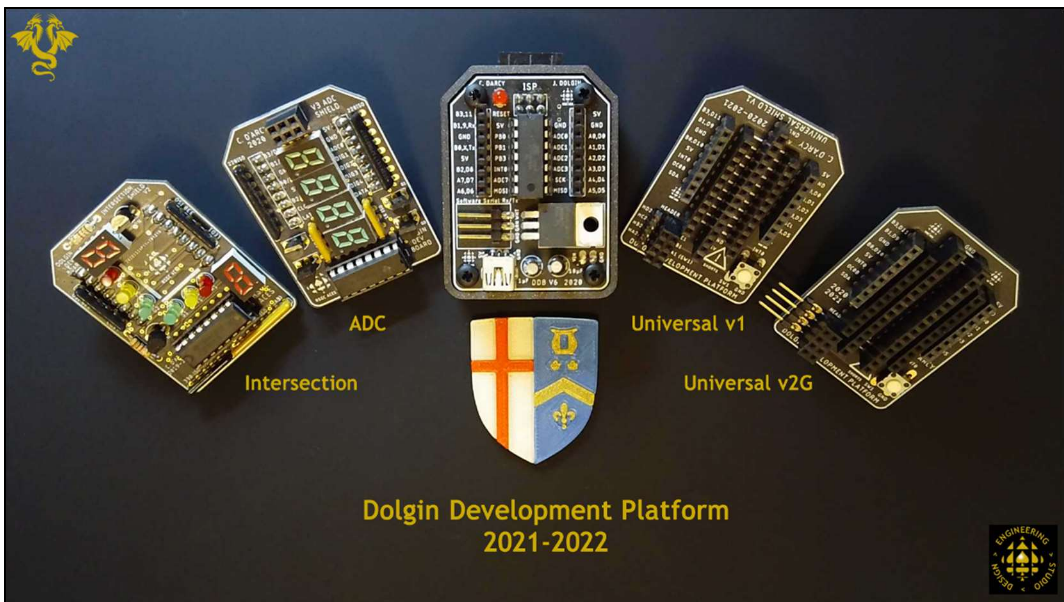
This **Disassembler** view is available within your new IDE, Atmel Studio, while engaging a debugging session (Debug > `Window > Disassembler`)

As can be seen, the compiler translates each high-level statement into **one or more** assembly language instructions. Generic comments are even added for your convenience.

The opportunity for the embedded systems programmer is to make this even more efficient.

Teaching you how to code in assembly language is one of the goals of this final ACES course. In doing so, we eliminate the compiler and all the assumptions it makes about your high-level intentions to ensure what is flashed into your MCU is the most efficient code achievable.

0.2 Dolgin Development Platform



0.3 Bit Coding Gymnastics

One difference between a painter and an artist may very well be the size of the brush. So, too, does the beginning coder use the broad coding strokes that may accomplish the intended task but often results in collateral damage (aka *side effects*) and performance inefficiency. Setting, clearing or inverting a single, or group of, bits is an example of the fine brush strokes the register-level or assembly-level programmer is frequently required to do. Use of the bitwise operators (`not-~`, `and-&`, `or-|`, and `xor-^`) are brought to bear. Register-level examples of these tasks appear below.

0.3.0 Setting a Bit

Setting a bit means making it 1. The example below is a register-level improvement on `pinMode(13, OUTPUT);` for the ATmega328P,

```
PORTB |= 1<<PB5;
```

0.3.1 Clearing a Bit

Clearing a bit means making it 0. The example below is a register-level improvement on `digitalWrite(7, LOW);` for the ATmega328P,

```
PORTD &= ~(1<<PD7);
```

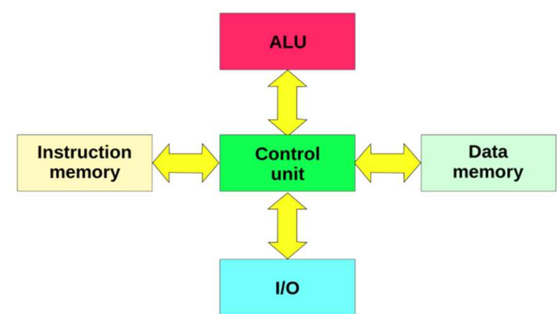
0.3.2 Inverting a Bit

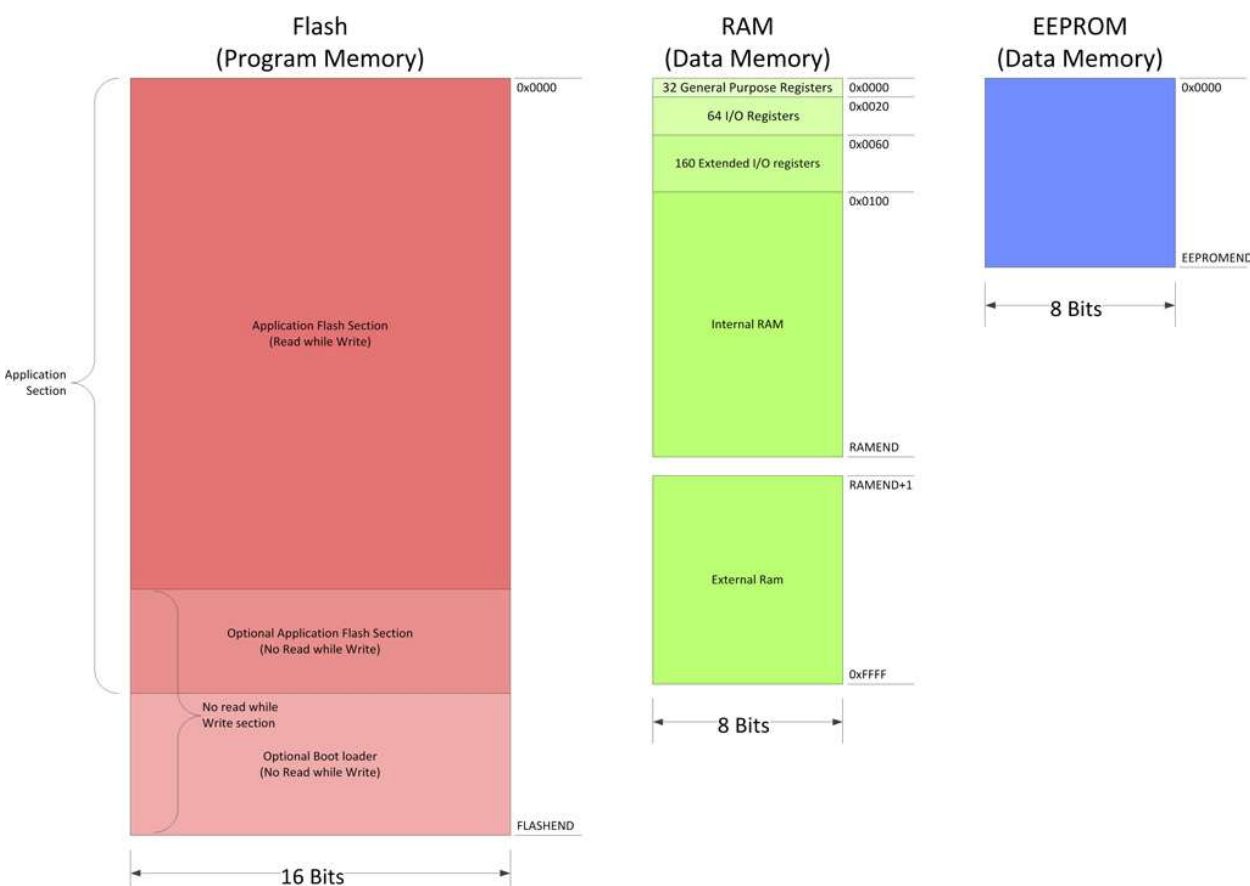
Inverting (aka *complementing*) a bit means switching it from 0 to 1 or vice versa. The example below is a register-level approach to inverting the I/O state of digital pin 13 for the ATmega328P,

```
PORTB ^= 1<<PB5;
```

1 AVR Memories

The AVR family of microcontrollers uses a modified Harvard Architecture (*instructions and data in separate areas*) which uses 3 types of memory: Flash, SRAM and onboard EEPROM.





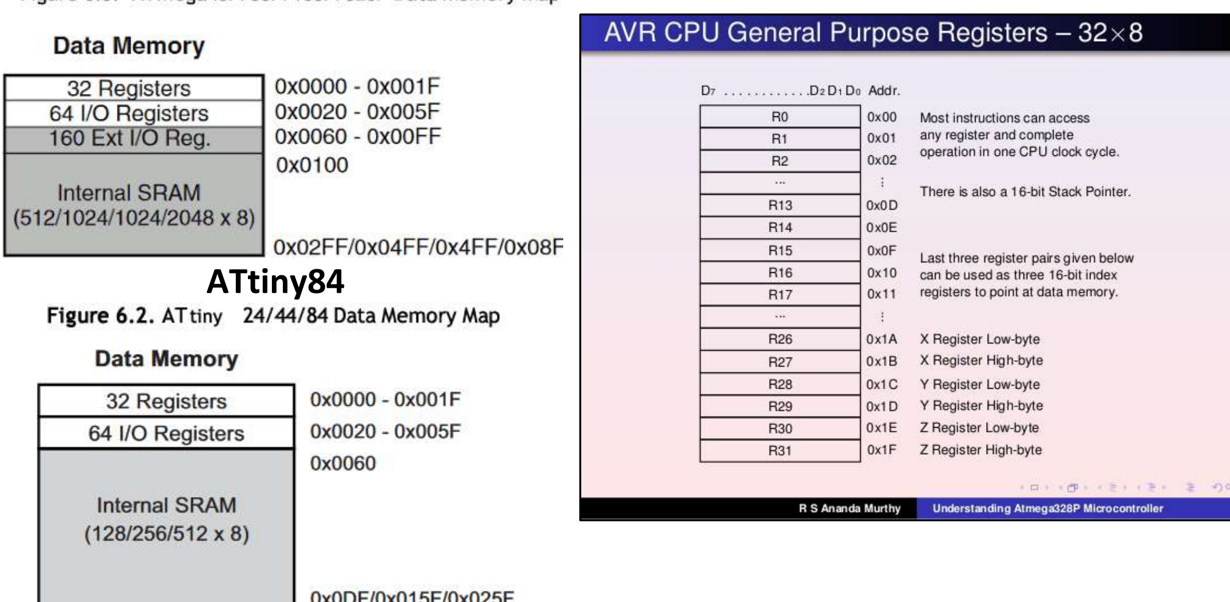
1.0 Flash Program Flash (ProgMem)

Flash is **non-volatile** memory, which means it persists when power is removed. Its purpose is to hold instructions that the microcontroller executes. The amount of flash can range from 512 bytes on an ATtiny to 384K on an ATxmega384A1.

1.1 Static RAM (SRAM)

SRAM (Data Memory) is **volatile** memory that stores the runtime state of the program being executed. The amount of RAM can range from 32 bytes on an ATtiny28L to 32KB on an ATxmega384A1. In many AVR microcontrollers RAM is split into 4 subsections: General Purpose Registers, General Purpose I/O Registers, Extended I/O Registers, and Internal RAM. AVR microcontrollers have RAM on-chip but some AVRs (e.g. ATmega128) can use external RAM modules.

Ahead. *Given there is considerably more space available in Flash Program Memory that either SRAM or EEPROM, C allows programmers to place data in the former when the latter is full. Certain steps must be undertaken to do so but it is easily doable. We'll discuss this technique later in the course.*



1.1.0 32 Private General Purpose (GP) Registers (0x00-0x1F)

The lowest 32 bytes of the AVR SRAM (`0x00–0x1F`) are mapped to the CPU for its efficient manipulation of data in support of assembly language instructions. These are referred to as the MCU's *private*, *general purpose* registers and are consistent throughout the *mega* and *tiny* families. As far as I know these locations are inaccessible to the register-level programmer.

1.1.1 64 I/O Registers (0x20-0x5F)

Located above the GP Registers, within the AVR's SRAM, lies a block of 64 bytes (0x20-0x5F) referred to as the I/O Register Space. The digital I/O Registers (aka Ports) are mapped to this area and, understandably, vary within the MCU families depending on their offerings. There are some consistencies maintained for compatibility.

1.1.1.0 Digital I/O Registers (Ports) (PINx, DDRx, PORTx)

IO Ports are the most common vehicle for your AVR to interface with real world. Each of the 328P and 84 digital pin numbers your code referenced in ICS3U are available for your review in the Quick Reference Guides inside the front cover of this workbook. Control over each digital pin number is accomplished through bit manipulation within three registers as shown below.

1.1.1.0.0 ATmega328P Digital I/O Registers (Ports)

A subset of the digital I/O addresses for the ATmega328P appears below.

http://mail.rsgc.on.ca/~cdarcy/Datasheets/RegisterSummary.pdf

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x15 (0x35)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0	
0x14 (0x34)	Reserved	–	–	–	–	–	–	–	–	
0x13 (0x33)	Reserved	–	–	–	–	–	–	–	–	
0x12 (0x32)	Reserved	–	–	–	–	–	–	–	–	
0x11 (0x31)	Reserved	–	–	–	–	–	–	–	–	
0x10 (0x30)	Reserved	–	–	–	–	–	–	–	–	
0x0F (0x2F)	Reserved	–	–	–	–	–	–	–	–	
0x0E (0x2E)	Reserved	–	–	–	–	–	–	–	–	
0x0D (0x2D)	Reserved	–	–	–	–	–	–	–	–	
0x0C (0x2C)	Reserved	–	–	–	–	–	–	–	–	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	92
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	92
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	92
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	91
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	91
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	92
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	91
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	91
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	91
0x02 (0x22)	Reserved	–	ATmega328P				–	–	–	
0x01 (0x21)	Reserved	–					–	–	–	
0x0 (0x20)	Reserved	–					–	–	–	

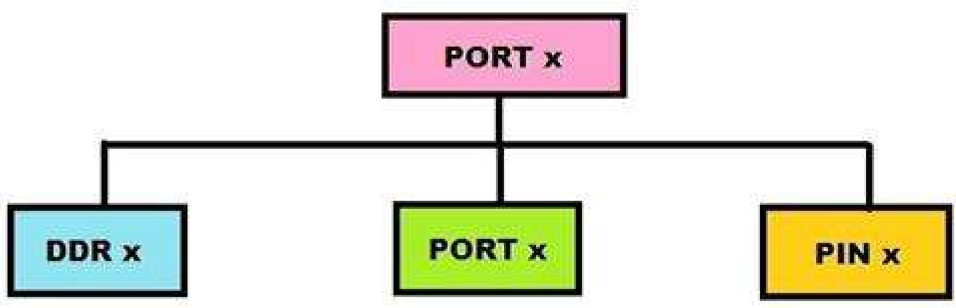
1.1.1.0.1 ATtiny84 I/O Registers (Ports)

A subset of the I/O addresses for the ATtiny84 appears below.

http://mail.rsgc.on.ca/~cdarcy/Datasheets/ATtiny84Registers.pdf

0x1B (0x3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
0x1A (0x3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
0x19 (0x39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
0x18 (0x38)	PORTB	ATtiny84				PORTB3	PORTB2	PORTB1	PORTB0
0x17 (0x37)	DDRB					DDB3	DDB2	DDB1	DDB0
0x16 (0x36)	PINB					PINB3	PINB2	PINB1	PINB0

Each Port has dedicated set of 3 registers mapped to it that are manipulated by your code to control the flow of data between the AVR and its external circuitry. The registers (aka ports) are the Data Direction Register (**DDRx**), and Output Register (**PORTx**) and Input Register (**PINx**), where **x** is replaced with an MCU-specific uppercase letter.



By virtue of its 8-bit width, each Port can govern to eight pins on the AVR. For example, the 8-bit register PORTD, on the ATmega328P is responsible for managing the behaviour of pins PD0 through PD7. One bit in each of the three PORTD registers is dedicated to each pin. A quick glance at the pinout diagram of the ATmega328P a few pages later reveals that PD0 is actually pin 2 on the chip. This pin maps to digital pin 0 on the Arduino UNO. A second look at the pinout diagram reveals other interesting details. There is no PORTA on this chip and PORTC only has seven active pins (PC0-PC6). All of these details can be reviewing the snapshot of the Register Summary on presented earlier on page 5.

Many of the high-level Arduino C instructions you used last year manipulate the bits in these ports in some way. For example, the `pinMode(pin, mode)` instruction, first determined the PORT pin that as mapped to the Arduino pin you were attempting to manipulate before clearing (`INPUT`) or setting (`OUTPUT`) the corresponding bit in the PORTs DDR register.

PORT	High-Level Arduino C	Register Level
DDRx	`pinMode(13,OUTPUT);`	`DDRB |= 1<<5;`
PORTx	`digitalWrite(13, LOW);`	`PORTB &= 1<<5;`
PINx	`uint8_t res = digitalRead(13);`	`uint8_t res = PINB &(1<<5)?1:0;`

1.1.1.0.2 DDRx

The value of the bits within a *Data Direction* Register defines the I/O direction of the corresponding digital pin: 0 for Input, 1 for Output. This helps explain why Input is the default.

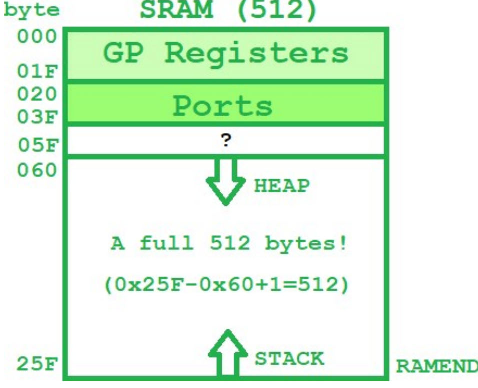
1.1.1.0.3 PORTx

The bits within a Port's *Output* Register define the voltage level for the corresponding digital pin: 0 for 0V, 1 for 5V.

1.1.1.0.4 PINx

The bits within a Port's *Input* Register define the voltage level read that appears on corresponding digital pin: 0 for 0V, 1 for 5V.

1.1.1.2 Stack Pointer (SPH and SPL)



The Stack is a (LIFO) data structure of significance that will be discussed thoroughly. Similar to the **SREG**, its use is essential for the correct execution of code. Not surprisingly then, the addresses of this *two*-byte register is tucked just under the **SREG** address. Again from the images above, the addresses are consistent between the 328P and 84. The Stack is an area of SRAM that expands and shrinks dynamically during execution. The *Stack Pointer* (**SP**) always holds the **address** of the top of the Stack. Initially, it is positioned at the highest address of available SRAM and grows 'backwards' in the sense that as data or addresses are added to the Stack (pushed), the contents of the Stack Pointer, decreases. As data or addresses are removed from the Stack (popped), the contents of the Stack Pointer, increases.

An implication of the Stack's characteristics is the number of bits that must be reserved for the Stack Pointer. From the images above it is 10 for the 328P and 9 for the 84. The Stack Pointer then consists of two sub-registers, Stack Pointer High (**SPH**) and Stack Pointer Low (**SPL**).

1.1.1.3 Status Register (SREG)

Each of the hundred-plus AVR assembly language instructions has the ability to reflect the result of the operation through the setting of a set of 8 bits, referred to as *flags*. These flags are bundled together in a register known as the *Status Register* or **SREG**. So critical to the correct execution of code is the **SREG** that it is given a prominent address in SRAM at the top (`0x5F`). This address is consistent between the ATmega328P and ATtiny84 MCUs.

ATmega328P

0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	10
0x3E (0x5E)	SPH	–	–	–	–	–	(SP10) 5.	SP9	SP8	13
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	13
0x3C (0x5C)	Reserved	–	–	–	–	–	–	–	–	
0x3B (0x5B)	Reserved	–	–	–	–	–	–	–	–	
0x3A (0x5A)	Reserved	–	–	–	–	–	–	–	–	
0x39 (0x59)	Reserved	–	–					–	–	
0x38 (0x58)	Reserved	–	–					–	–	
0x37 (0x57)	SPMCSR	SPMIE	(RWWSB) 5.	SIGRD	(RWWSRE) 5.	BLBSET	PGWRT	PGERS	SPMEN	278

ATtiny84

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	Page 8
0x3E (0x5E)	SPH	–	–	–	–	–	–	SP9	SP8	Page 11
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	Page 11
0x3C (0x5C)	OCR0B	Timer/Counter0 – Output Compare Register B								Page 85
0x3B (0x5B)	GIMSK	–	INT0	PCIE1	PCIE0	–	–	–	–	Page 51
0x3A (0x5A	GIFR	–	INTF0	PCIF1	PCIF0	–	–	–	–	Page 52
0x39 (0x59)	TIMSK0	–					OCIE0B	OCIE0A	TOIE0	Page 85
0x38 (0x58)	TIFR0						OCF0B	OCF0A	TOV0	Page 85

0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	10
0x3E (0x5E)	SPH	–	–	–	–	–	(SP10) 5.	SP9	SP8	13
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	13

When an assembly language instruction completes execution, the results are reflected in the Status Register (SREG). Conditions that can be examined and registered include whether the result of a calculation was negative (N), or whether the result of an arithmetic operation overflowed the 8 bit destination register (V) In total, there are 8 'flags' that can potentially be affected. The manner in which the flags are affected is detailed for each instruction in the AVR Instruction Manual (a link appears at the top of our course page). The second-to-last column in the Instruction Set Summary indicates which flags are affected by each instruction, but not how.

Flag	Description
I	Global Interrupt Enable/Disable Flag
T	Transfer bit used by BLD and BST instructions
H	Half Carry Flag
S	N ⊕V, For signed tests
V	Two's complement overflow indicator
N	Negative Flag
Z	Zero Flag
C	Carry Flag

The purpose of leaving flags in certain state is so that next instruction can take appropriate action based on the result of its previous instruction. This is particularly true of the branching instructions.

BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1/2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1/2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1/2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1/2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1/2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1/2

http://mail.rsgc.on.ca/~cdarcy/Datasheets/InstructionSetSummary.pdf

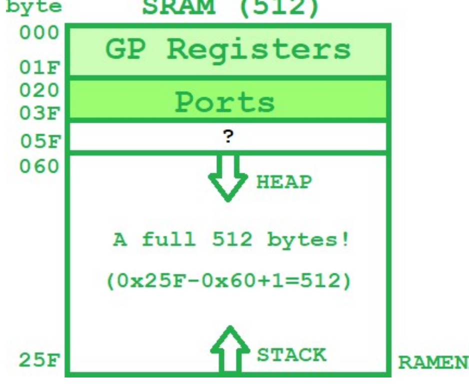
1.1.2 160 Extended I/O Registers (0x60-0xFF)

To accommodate the broader capabilities in the ATmega family over the ATtiny family, an additional 160 bytes of SRAM are set aside in the mega family for the *extended* I/O register set. This might seem like an unusual number, but when taken together with the previous address ranges, the total amounts to a familiar 256 bytes of reserved SRAM (32+64+160=256).

(0x8C)	Reserved	–	–	–	–	–	–	–	–	
(0x8B)	OCR1BH	Timer/Counter1 - Output Compare Register B High Byte								135
(0x8A)	OCR1BL	Timer/Counter1 - Output Compare Register B Low Byte								135
(0x89)	OCR1AH	Timer/Counter1 - Output Compare Register A High Byte								135
(0x88)	OCR1AL	Timer/Counter1 - Output Compare Register A Low Byte								135
(0x87)	ICR1H	Timer/Counter1 - Input Capture Register High Byte								135
(0x86)	ICR1L	Timer/Counter1 - Input Capture Register Low Byte								135
(0x85)	TCNT1H	Timer/Counter1 - Counter Register High Byte								134
(0x84)	TCNT1L	Timer/Counter1 - Counter Register Low Byte								134
(0x83)	Reserved	–	–	–	–	–	–	–	–	
(0x82)	TCCR1C	FOC1A	FOC1B	–	–	–	–	–	–	134
(0x81)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	133
(0x80)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	131
(0x7F)	DIDR1	–	–	–	–	–	–	AIN1D	AIN0D	236
(0x7E)	DIDR0	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	251
(0x7D)	Reserved	–	–	–	–	–	–	–	–	
(0x7C)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	248
(0x7B)	ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0	251
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	249

Consult Chapter 36, *Register Summary*, of the ATmega328P datasheet to see the specific details of address mapping.

1.1.3 SRAM (Heap and Stack) (0x??-RAMEND)



With the lowest SRAM addresses (varies between families) set aside for dedicated Register use as described above, the remaining space is free for use by your code to influence and exploit dynamically. The amount of free space remaining depends on your MCU. The highest address can be determined programmatically by accessing a predefined constant typically included in the toolchain as RAMEND.

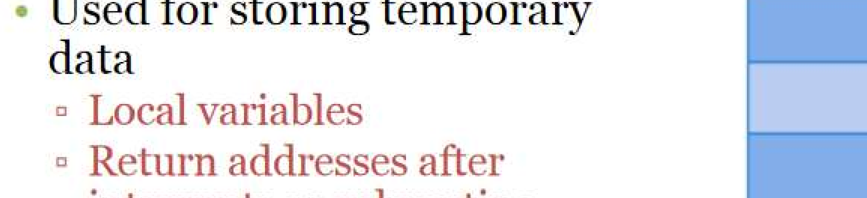
Although your code is free to use the entire range of SRAM between the end of the Extended Register section and RAMEND, there are additional transparent code behaviours you must be aware to ensure correct code performance. The concepts are generally referred to as the **heap** and the **system stack**.

1.1.3.0 Heap

The **heap** is the preferred area of SRAM that the assembler looks to, to satisfy the bytes of storage required by your global variable declarations (*dynamic* memory allocation is beyond the scope of this course). Generally, the byte range of the heap extends from just above the Extended Register set and continues as required.

Should your code attempt to declare an array of bytes required storage that exceeded the .variables that you declare are stored here, as are parameters passed to functions and local variables declared within them.

1.1.3.1 (System) Stack



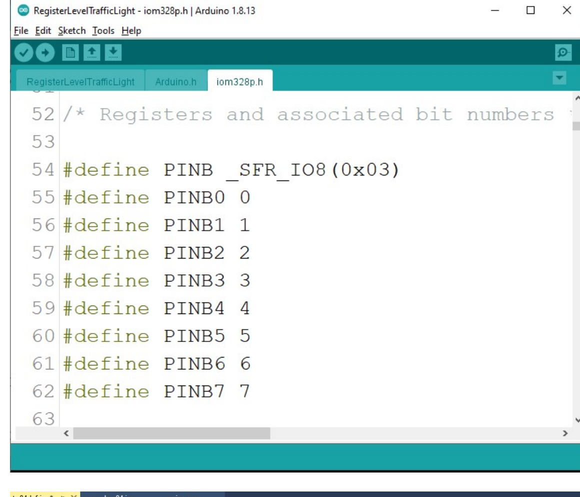
1.2 EEPROM

EEPROM (*Electronically Erasable Programmable Read Only Memory*) is **non-volatile** memory which is used to store data. The most common use is to store configurable parameters. The amount of EEPROM can range from 32 bytes on an ATtiny to 4KB on an XMega.

EEPROM is a good place to log data from sensors, store values as a Lookup Table (LuT) for faster performance by avoiding computationally-intense calculations (trig values), or data such as font maps, to name a couple of common uses.

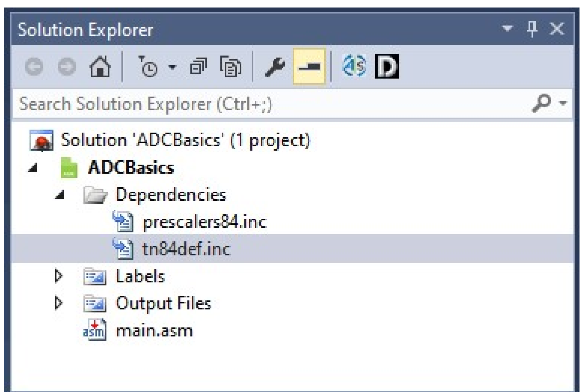
.Reference: http://www.protostack.com/blog/2010/12/avr-memory-architecture/

1.3 Predefines (.h and .inc)



The specific register *names* and corresponding *addresses* are available for use in your register-level Arduino C programs in the form of a header file (**.h**). Selecting the target board within the Arduino IDE results in the correct files of predefines being included in the toolchain, automatically. The files of predefines are `iom328p.h` and `iotnx4.h` for the UNO and DDB, respectively. You can explore the contents of these files by following the links at the top of our course page.

When programming in Assembly Language within Atmel Studio 7, the appropriate file of predefines (**.inc**) are made known to your project when you select the target board in the project creation dialog sequence.



Once you complete a successful build of your assembly language project the predefine include file (**.inc**) will appear in the *Dependencies* section. Click to open to examine its contents.

2 Interrupts

MCUs are designed with the ability to immediately stop executing some code and address a service alert from a secondary source (e.g. *sensor*, *timer*, *button*, etc.) they are responsible for. Software that is configured in this manner is called *interrupt-driven*. The list of alerts to which 8-bit AVR MCUs can respond are summarized within the respective datasheets in an Interrupt Vector Table.

2.0 Interrupt Vector Table (IVT)

An *Interrupt Vector Table* (aka, Interrupt *Jump* Table) is a dedicated set of bytes at the beginning of Program Flash Memory reserved for programmers to populate with code addresses of their functions to execute when specific events occur. These user functions are best referred to as *Interrupt Service Routines* (ISRs). When correctly configured, the system automatically saves the current contents of the Program Counter (*on the Stack*), goes to a location within the IVT and loads the address it finds there into the Program Counter, thereby transferring control (aka *jump*) to your ISR. When your ISR finishes execution, the previously saved address is retrieved from the top of the Stack and execution continues as it did prior to the event.

2.0.0 ATmega328P IVT

12.1 Interrupt Vectors in ATmega48A and ATmega48PA

Table 12-1. Reset and Interrupt Vectors in ATmega48A and ATmega48PA

Vector No.	Program Address	Source	Interrupt Definition
1	0x000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	PCINT0	Pin Change Interrupt Request 0
5	0x004	PCINT1	Pin Change Interrupt Request 1
6	0x005	PCINT2	Pin Change Interrupt Request 2
7	0x006	WDT	Watchdog Time-out Interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x008	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x009	TIMER2 OVF	Timer/Counter2 Overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x00B	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x00C	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x00D	TIMER1 OVF	Timer/Counter1 Overflow
15	0x00E	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x010	TIMER0 OVF	Timer/Counter0 Overflow
18	0x011	SPI, STC	SPI Serial Transfer Complete
19	0x012	USART, RX	USART Rx Complete
20	0x013	USART, UDRE	USART, Data Register Empty
21	0x014	USART, TX	USART, Tx Complete
22	0x015	ADC	ADC Conversion Complete
23	0x016	EE READY	EEPROM Ready

2.0.1 ATtiny84 IVT

Understandably, MCUs within the *tiny* family offer fewer resources, hence a smaller vector table.

Interrupt Vectors

Table 10-1. Reset and Interrupt Vectors

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External pin, power-on reset, brown-out reset, watchdog reset
2	0x0001	INT0	External interrupt request 0
3	0x0002	PCINT0	Pin change interrupt request 0
4	0x0003	PCINT1	Pin change interrupt request 1
5	0x0004	WDT	Watchdog time-out
6	0x0005	TIMER1 CAPT	Timer/Counter1 capture event
7	0x0006	TIMER1 COMPA	Timer/Counter1 compare match A
8	0x0007	TIMER1 COMPB	Timer/Counter1 compare match B
9	0x0008	TIMER1 OVF	Timer/Counter0 overflow
10	0x0009	TIMER0 COMPA	Timer/Counter0 compare match A
11	0x000A	TIMER0 COMPB	Timer/Counter0 compare match B
12	0x000B	TIMER0 OVF	Timer/Counter0 overflow
13	0x000C	ANA_COMP	Analog comparator
14	0x000D	ADC	ADC conversion complete
15	0x000E	EE_RDY	EEPROM ready
16	0x000F	USI_START	USI START
17	0x0010	USI_OVF	USI overflow

2.1 Avoiding Conflicts with the IVT in Assembly Language

Given its critical role in the successful execution of interrupt-driven applications, the IVT is expected to appear at the very start of Program Flash, addresses `0x0000-0x????`. To ensure your assembly language data and code avoids this range, use of the `.org` directive is encouraged. The predefine `INT_VECTORS_SIZE` supports a degree of MCU-compatibility, as in,

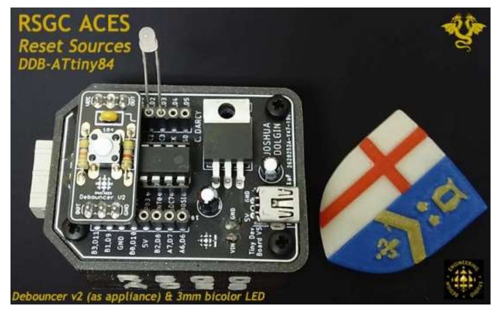
```
24  ; CODE Segment (default)
25  .cseg                           ;locate for Code Segment (PROGRAM FLASH)
26  ; ***** INTERRUPT VECTOR TABLE ************************************************
27  .org        0x0000              ;start of Interrupt Vector Table (IVT) aka. Jump Table
28      rjmp    reset               ;lowest interrupt address == highest priority!
29  .org        ADCCaddr            ;External Interrupt Request 0 (predefined in tn84def.inc)
30      rjmp    ADCComp             ;ISR for ADC ADC Conversion Complete interrupt
31  .org    INT_VECTORS_SIZE        ;position program data just beyond the IVT
```

2.1.0 Interrupt Priorities

The order of the interrupt sources within the vector table is significant. Should two or more interrupts occur simultaneously, the sources are queued, with the lower address given priority.

A natural consequence of this is that a request to reset the MCU is awarded the highest priority.

2.2 Reset Interrupt



A **Reset** event results in the clearing of (set to 0) the Program Counter. Your code then has the responsibility of placing a *jump* instruction at address 0x0000 of the first executable instruction.

A **Reset** event can be triggered in a number of ways depending on the MCU. The most common is the power on reset. Every time you reconnect power to the MCU a **Power On Reset** event is generated.

Alternatively, whenever a falling edge (5V→0V) is presented on pin 1, such as the momentary button your wired into your breadboard Arduino is Grade 11, you generate an **External Reset** event.

Another common Reset source is the **WatchDog Timer**. In this way your software can generate a **WatchDog** Reset when a specific event occurs or at periodic intervals.

2.2.0 MCUSR (Reset) Register

The source of a reset can be determined by examining the bits (flags) within the MCU Status Register (**MCUSR**). Unlike many other registers the address and bits with the register are the same for the mega328P and tiny84 MCUs.

9.10.1 MCUSR – MCU Status Register

The MCU status register provides information on which reset source caused an MCU reset.

Bit	7	6	5	4	3	2	1	0	
0x34 (0x54)	–	–	–	–	WDRF	BORF	EXTRF	PORF	MCUSR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	See Bit Description				

2.2.1 Rotary Encoder on RSGC ACES Breakout Board

2.3 External Interrupts

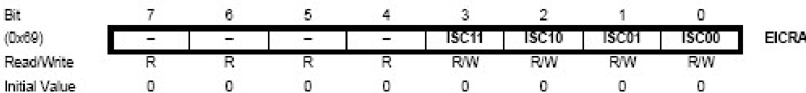
Next the Reset interrupt, an **external interrupt** event can be configured to trigger an immediate response. The mega328P has two pins (**INT0** and **INT1**) capable of responding to a changing edge, and the tiny84 (**INT0**) just one.

2.3.0 ATmega328P External Interrupt Registers

13.2 Register Description

13.2.1 EICRA – External Interrupt Control Register A

The External Interrupt Control Register A contains control bits for interrupt sense control.

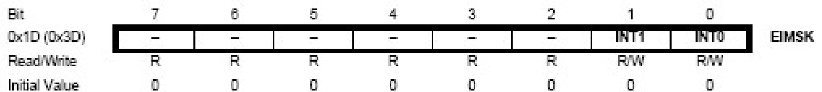


Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 13-2. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

13.2.2 EIMSK – External Interrupt Mask Register



Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – INT0: External Interrupt Request 0 Enable**

When the INT0 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control0 bits 1/0 (ISC01 and ISC00) in the External Interrupt Control Register A (EICRA) define whether the external interrupt is activated on rising and/or falling edge of the INT0 pin or level sensed. Activity on the pin will cause an interrupt request even if INT0 is configured as an output. The corresponding interrupt of External Interrupt Request 0 is executed from the INT0 Interrupt Vector.

2.3.1 ATtiny84 External Interrupt Registers

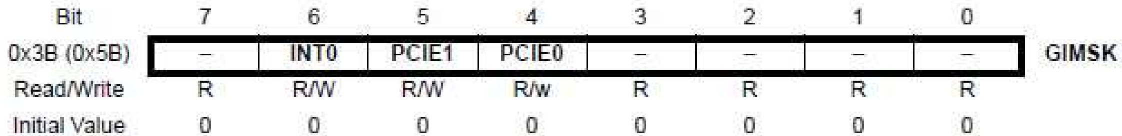
Interrupt Sense Control bits (**ISC01** and **ISC00**) for the ATtiny84 defines the same edges as the ATmega328P.

11.2.1 MCUCR – MCU Control Register

The external interrupt control register A contains control bits for interrupt sense control.

Bit	7	6	5	4	3	2	1	0	
0x35 (0x55)	BODS	PUD	SE	SM1	SM0	BODSE	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

11.2.2 GIMSK – General Interrupt Mask Register



Bit	7	6	5	4	3	2	1	0	
0x3B (0x5B)	–	INT0	PCIE1	PCIE0	–	–	–	–	GIMSK
Read/Write	R	R/W	R/W	R/w	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

2.4 Pin Change Interrupts

By this point you are aware that your MCUs can sense and respond to external events through the use of the External Interrupt System (**INTn**). The good news is that their corresponding **ISCnn** bits can configured to monitor low, logical, falling, or rising signals. The downside is that their application is limited to two specific pins on the ATmega328P and only one on the ATtiny84.

A useful alternative to External Interrupts for sensing and responding to external signal events are **Pin Change Interrupts** that are applicable to any digital pin! This means that your ATmega328P can perform a similar function on all 23 pins and the ATtiny84 on all 12. However, as is always the case, the downside is that, as its name implies, only a change (falling or rising) signal edge triggers the interrupt.

2.4.0 ATmega328P Pin Change Interrupt Control Register

13.2.4 PCICR – Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

2.4.1 ATtiny84 General Interrupt Mask Register

11.2.2 GIMSK – General Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x3B (0x5B)	–	INT0	PCIE1	PCIE0	–	–	–	–	GIMSK
Read/Write	R	R/W	R/W	R/w	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

2.4.2 ATtiny84 Pin Change Mask Registers

11.2.4 PCMSK1 – Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
0x20 (0x40)	–	–	–	–	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

11.2.5 PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
0x12 (0x32)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

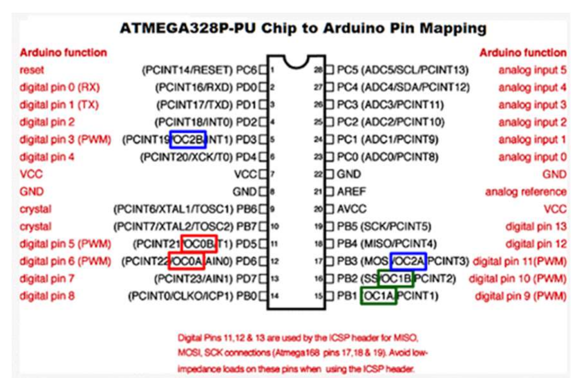
3 Timer/Counters

The heartbeat of a functioning MCU is either an internal or external **clock source,** aka *oscillator* (e.g. *crystal*, *RC*, etc.). The source of the oscillation can be an external crystal (attached to MCU pins 9 and 10) or an internal oscillator. Every tick of the clock source (up to 20 MHz) is registered as the MCU's *free-running clock* (clk_{IO}) that runs in the background. Each AVR MCU has multiple Timer/Counters. Each Timer/Counter has a suite of registers that can be programmed to produce into various waveform shapes from the clock. In addition to its role in coordination around a common beat, Timers these peripherals can also **count** pulses. Your first exposure to the value of counting may have been your Grade 10 Counting Circuit project that used a NAND-Gate Oscillator as a clock source that was fed into a 4017 decade counter to monitor the 'ticking'. Specific registers are set aside within each Timer/Counter for the accumulation of clock source 'ticks'.

Of the many uses a Timer/Counter can be put to, pulse width modulation (PWM) was likely your earliest and most common Grade 11 application. Arduino C's `analogWrite(pin,duty)` function exploits the uses of the respective Timer/Counter associated with the `pin` requested.

3.0 ATmega328P

The ATmega328P has **three** Timer/Counters available for use. For each of the timers two digital pins can be directly influenced by register behaviour, allowing for maximum efficiency, as shown to the right.



Care must be taken not to inadvertently overlap their use. Here is a brief table of common functions and libraries that rely on the availability of the timers for their correct execution.

3.0.0 ATmega328P Timer/Counter0 Modes

Table 15-8. Waveform Generation Mode Bit Description

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

3.0.1 ATmega328P Timer/Counter1 Modes

Table 16-4. Waveform Generation Mode Bit Description[1]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

3.0.2 ATmega328P Timer/Counter2 Modes

Table 18-8. Waveform Generation Mode Bit Description

Mode	WGM22	WGM21	WGM20	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[1][2]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Notes: 1. MAX= 0xFF
2. BOTTOM= 0x00

3.0.3 ATmega328P Pulse Width Modulation (PWM) with AnalogWrite()

Arduino C's `analogWrite(pin, uint8_t)` function generates a PWM signal on the back of an MCU's Timer/Counter. Through the use of the second parameter users can control the *duty cycle* of the signal. Since users can not control the *frequency* of the PWM waveform the function is not suitable in all cases where a digital approximation of a voltage level is required (e.g. servo motor horn positioning).

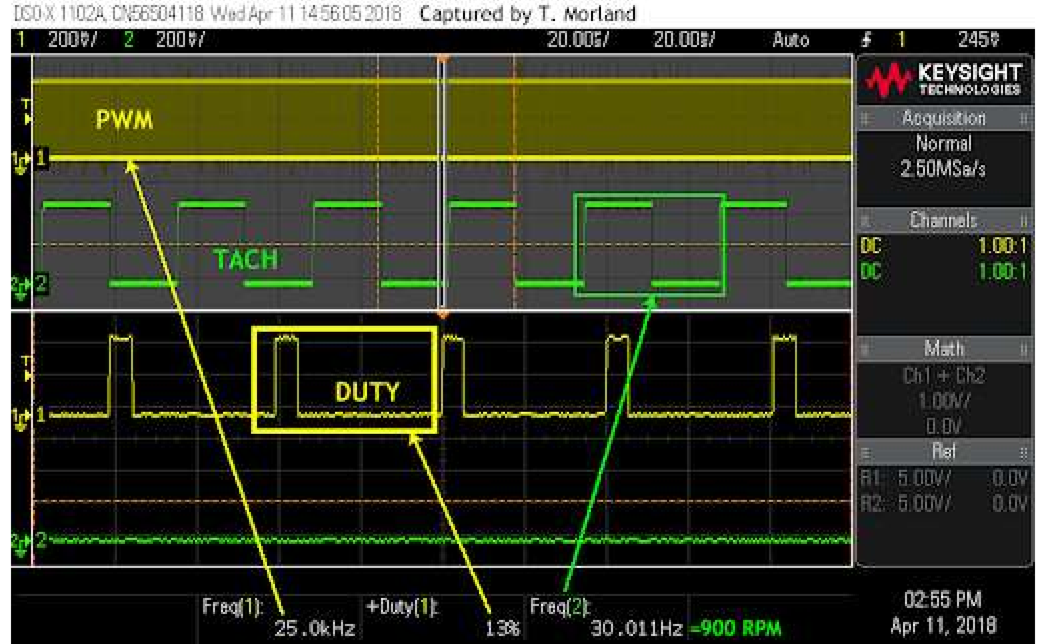
The table below lists the default frequencies associated with using the `analogWrite(pin,uint8_t)` on each of the respective pins of the ATmega328P, as well as the other library functions that rely on these respective Timer/Counters. Users should be aware of the potential conflicts that can arise.

Timer	Bits	Pins	`analogWrite` Frequency	Dependent Functions
0	8	5, 6	~980 Hz	`delay()`, `millis()`, `micros()`
1	16	9, 10	~490 Hz	`Servo` Library
2	8	3, 11	~490 Hz	`Tone` Library

Readers are encouraged to explore Ken Shirriff's remarkable blog, "*Secrets of Arduino PWM*" at, http://www.righto.com/2009/07/secrets-of-arduino-pwm.html

3.0.3.0 Scope Trace of an AnalogWrite() PWM Waveform

Below is one of my favourite images captured on our scope by Tim Morland (*ACES '18, Queen's '23*).



3.0.4 Atmega328P Timer/Counter1 Registers

16.11 Register Description

16.11.1 TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

16.11.2 TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

16.11.3 TCCR1C – Timer/Counter1 Control Register C

Bit	7	6	5	4	3	2	1	0	
(0x82)	FOC1A	FOC1B	–	–	–	–	–	–	TCCR1C
Read/Write	R/W	R/W	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

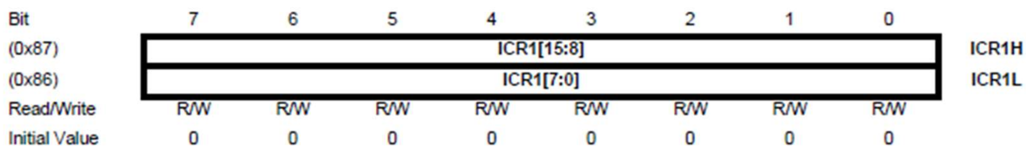
16.11.4 TCNT1H and TCNT1L – Timer/Counter1

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								TCNT1H
(0x84)	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

16.11.6 OCR1BH and OCR1BL – Output Compare Register 1 B

Bit	7	6	5	4	3	2	1	0	
(0x8B)	OCR1B[15:8]								OCR1BH
(0x8A)	OCR1B[7:0]								OCR1BL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

16.11.7 ICR1H and ICR1L – Input Capture Register 1



Bit	7	6	5	4	3	2	1	0	
(0x87)	ICR1[15:8]								ICR1H
(0x86)	ICR1[7:0]								ICR1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

16.11.8 TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

16.11.9 TIFR1 – Timer/Counter1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

3.0.5 Atmega328P Timer/Counter 1 Normal Mode 0

An example of the simplest programmable Timer/Counter1 mode would be **Mode 0: Normal Mode**. In this configuration,

1. Ticks of the clock sources are accumulated in its 16-bit (2^{16}) 2-byte register pair: **TCNT1H:TCNT1L**.
2. When the count reaches the top (`65535 = 0Xffff`), an overflow interrupt is generated
3. The interrupt can be dealt with in at least 3 ways: ignored completely, handled in software or responded to in hardware for example, with the **OC1A** or **OC1B** pins connected.
4. The counter simply rolls over and resumes counting from `0x0000`.
5. A prescaler may be applied to the clock to map the counting source to a reduced frequency.
6. As an example, consider Timer/Counter 1 in Normal Mode 0 under a 16 MHz crystal clock source with a prescaler of 256. The overflow frequency would be $2^{24}/2^{8}/2^{16}$ = 1 Hz.

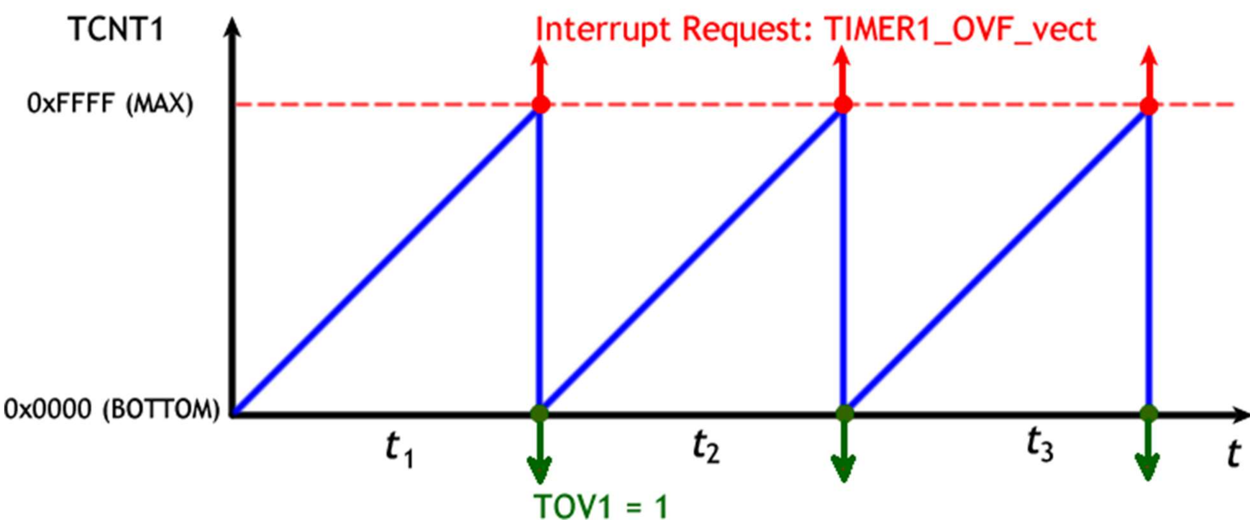


Table 13-9. Clock Select Bit Description

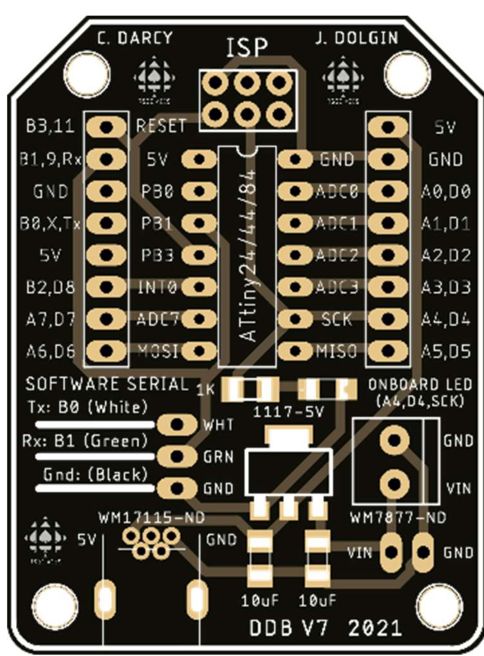
CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

If external pin modes are used for the Timer/Counter0, transitions on the T0 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

You'll be shown how different techniques in class for `#define`ing and `#include`ing these type of bit sequences depending on your preferred toolchain.

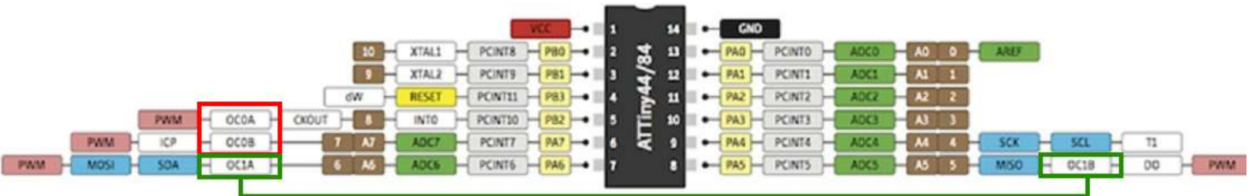
3.1 ATtiny84

From it inception, our RSGC ACES Dolgin Development Platform hosts the AVR ATtiny84 as its preferred MCU. The JLCPCB rendering of V7 appears to the right. This microcontroller was selected for a variety of reasons not the least of which was DAMellis/Konde ATtinyCore suite of Arduino IDE software supports, its compact footprint (14 pins) and having just enough peripheral features (*External Interrupt, two Timer/Counters, ADC, Watchdog etc.*) to support a wide range of applications.



You are encouraged to undertake a visual comparison of the features attached to each of the pins in the official diagram below and how the DDB breaks out the pins to the headers on our PCB to the right.

In the diagram below, the two pins of the 8-bit Timer/Counter0 are highlighted in red. As well, the two pins of the 16-bit Timer/Counter1 are highlighted in green.



3.1.0 ATtiny84 Timer/Counter0 Modes

Table 13-8. Waveform Generation Mode Bit Description

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[1]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, phase correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, phase correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Note: 1. MAX = 0xFF
BOTTOM = 0x00

3.1.1 ATtiny84 Timer/Counter1 Modes

Table 14-5. Waveform Generation Mode Bit Description[1]

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, phase correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, phase correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, phase correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, phase and frequency correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, phase and frequency correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, phase correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, phase correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

3.1.2 ATtiny84 Pulse Width Modulation (PWM) with AnalogWrite()

Through the use of the `analogWrite(pin, duty cycle)` function provide by the core Arduino libraries, a limited form of PWM signals have been available to you for such applications as dimming LEDs and DC motor speed control. Depending on which pin you invoke the behaviour on, you are implicitly selecting one of the available Timer/Counters on your MCU. This is summarized for the Ttiny84 in the table below.

Timer	Bits	Pins	`analogWrite` Frequency	Dependent Functions
0	8	8 (PB2), 7 (PA7)	? Hz	`delay()`, `millis()`, `micros()`
1	16	6 (PA6), 5 (PA5)	? Hz	`Tone` Library, `Servo` Library

Care must be taken when using `analogWrite` to avoid pins required by parallel use of the dependent functions indicated above that would result in strange behaviour.

Finally, As useful as the `analogWrite` function is, it does not offer control over the frequency of the square wave which is essential for a wider variety of MCU functionality and applictions. We need to dig deeper.

3.1.3 ATtiny84 Timer/Counter Registers

ATtiny84 Timer/Counter Ports

Timer/Counter 0

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3C (0x5C)	OCR0B	Timer/Counter0 – Output compare register B							
0x39 (0x59)	TIMSK0	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0
0x36 (0x56)	OCR0A	Timer/Counter0 – Output compare register A							
0x33 (0x53)	TCCR0B	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00
0x32 (0x52)	TCNT0	Timer/Counter0							
0x30 (0x50)	TCCR0A	COM01	COM0A0	COM0B1	COM0B0	–		WGM01	WGM00

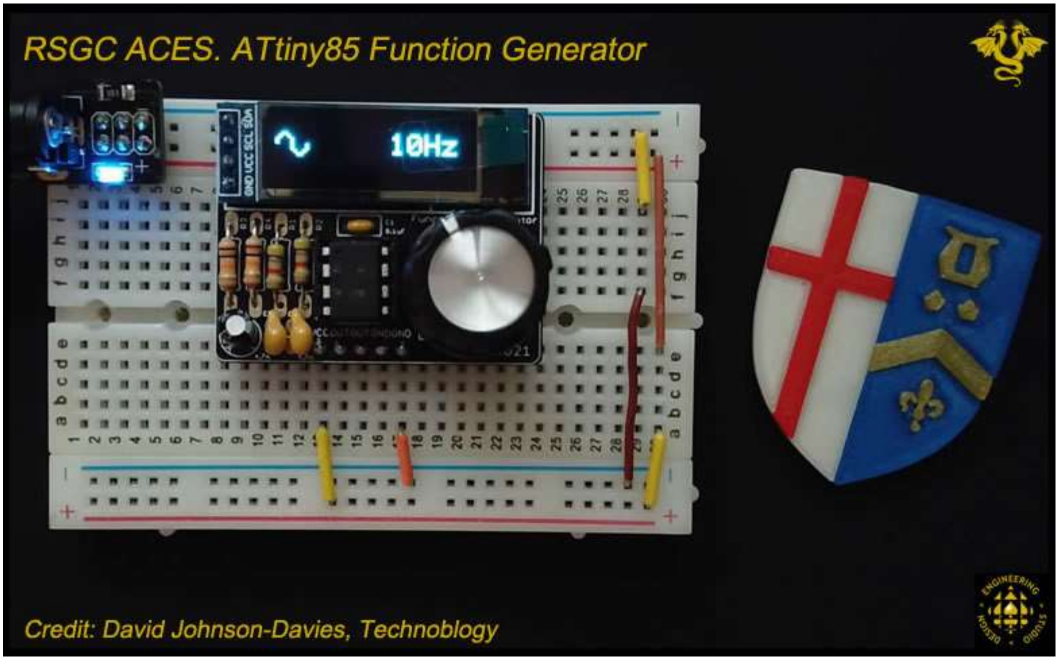
Timer/Counter 1

Address	Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x2F (0x4F)	TCCR1A	COM11	COM1A0	COM1B1	COM1B0	–		WGM11	WGM10
0x2E (0x4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
0x2D (0x4D)	TCNT1H	Timer/Counter1 – Counter register high byte							
0x2C (0x4C)	TCNT1L	Timer/Counter1 – Counter register low byte							
0x2B (0x4B)	OCR1AH	Timer/Counter1 – Compare register A high byte							
0x2A (0x4A)	OCR1AL	Timer/Counter1 – Compare register A low byte							
0x29 (0x49)	OCR1BH	Timer/Counter1 – Compare register B high byte							
0x28 (0x48)	OCR1BL	Timer/Counter1 – Compare register B low byte							
0x0C (0x2C)	TIMSK1	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1

ASM

3.2 ATtiny85 Timer Application: Function Generator

Inspiration for an RSGC ACES Function Generator developed in the Spring of 2021, based on the position of a rotary encoder, came from David Johnson-Davies terrific blog on the project, http://www.technoblogy.com/show?22HF.



3.3 Accessing 16-Bit Registers

(*Lifted directly from the ATmega328P datasheet…*)

The TCNT1, OCR1A/B, and ICR1 are 16-bit registers that can be accessed by the AVR CPU via the 8-bit data bus. The 16-bit register must be byte accessed using two read or write operations. Each 16-bit timer has a single 8-bit register for temporary storing of the high byte of the 16-bit access. The same temporary register is shared between all 16-bit registers within each 16-bit timer. Accessing the low byte triggers the 16-bit read or write operation. When the low byte of a 16-bit register is written by the CPU, the high byte stored in the temporary register, and the low byte written are both copied into the 16-bit register in the same clock cycle. When the low byte of a 16-bit register is read by the CPU, the high byte of the 16-bit register is copied into the temporary register in the same clock cycle as the low byte is read.

Not all 16-bit accesses uses the temporary register for the high byte. Reading the OCR1A/B 16-bit registers does not involve using the temporary register.

To do a 16-bit write, the high byte must be written before the low byte. For a 16-bit read, the low byte must be read before the high byte.

The following code examples show how to access the 16-bit Timer Registers assuming that no interrupts update the temporary register. The same principle can be used directly for accessing the OCR1A/B and ICR1Registers. Note that when using "C", the compiler handles the 16-bit access.

Assembly Code Examples[1]

```
...
; Set TCNT1 to 0x01FF
ldi     r17,0x01
ldi     r16,0xFF
out     TCNT1H,r17
out     TCNT1L,r16
; Read TCNT1 into r17:r16
in      r16,TCNT1L
in      r17,TCNT1H
...
```

C Code Examples[1]

```
unsigned int i;
...
/* Set TCNT1 to 0x01FF */
TCNT1 = 0x1FF;
/* Read TCNT1 into i */
i = TCNT1;
...
```

4 ADC: Analog to Digital Conversion

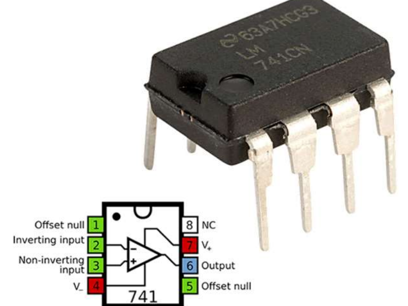
The real world is *continuous*; the behaviours that nature exhibits (heat, pressure, light, force, etc.) are said to be **analog**. Many forms of sensors exist that convert these analog behaviours to continuous voltage levels. A light-dependent resistor or LDR (aka. *photoresistor*) for example, in series with another known resistor level, can provide an MCU with access to a continuous voltage.

The **digital** world of MCUs interpret the analog voltage output of the sensors by mapping them to a range of *discrete* voltage levels represented internally, by binary numbers. The MCU's process of transforming a continuous analog voltage reading to a discrete digital approximation is the subject of this chapter.

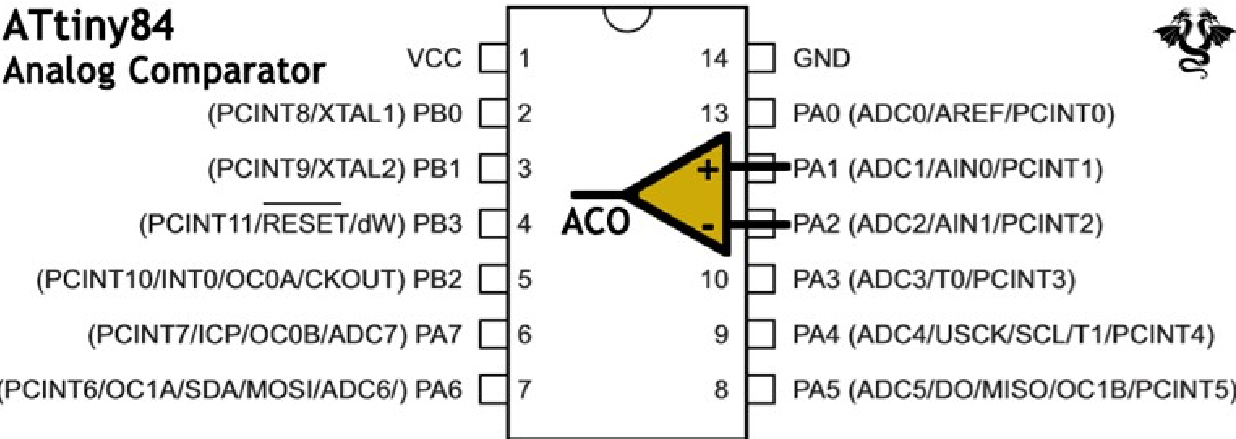
Of all the features that microcontrollers offer, a strong case could be made for **Analog to Digital Conversion** being its most important function. After all, the ability to capture real world data and digitize it for manipulation, transmission, and storage purposes is an undeniably critical feature within our modern world. Although the AVR line of 8-bit MCUs offers a **10-bit onboard ADC** unit that we've exploited for a number of purposes, what if our needs called for either a higher or lower sampling accuracy? A deeper understanding of how the ADC function works is called for should we wish to build our own ADC unit.

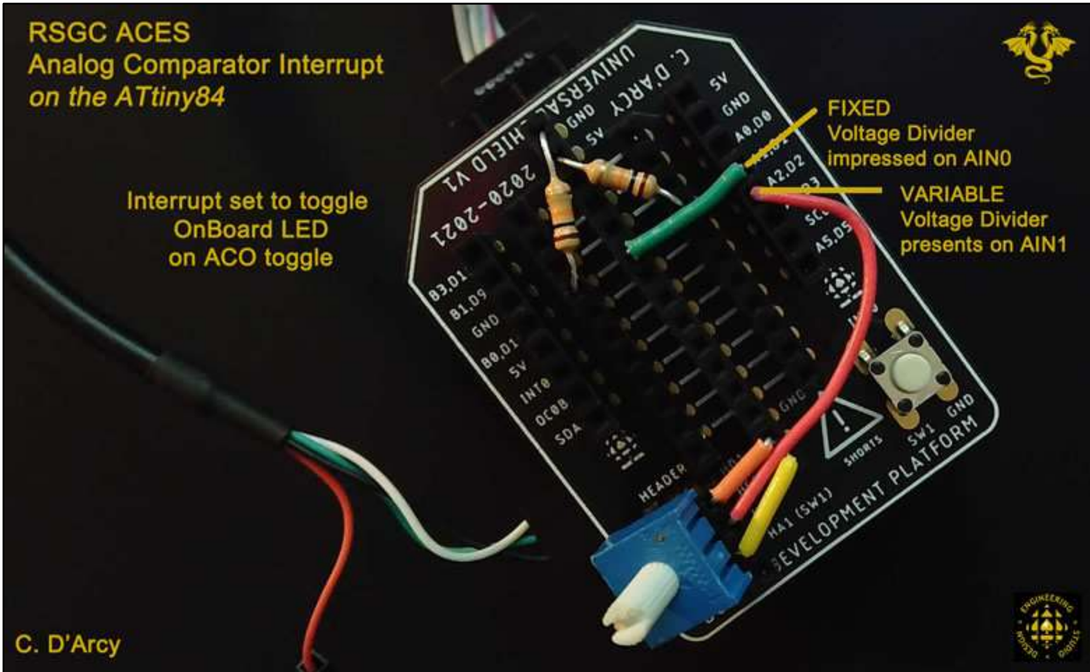
4.0 Analog Comparator

In previous years you likely explored the ability of operational amplifier to act as a *comparator*. The classic LM741 will output a **high** signal on Pin 6 if the voltage on the non-inverting input (Pin 3) is greater than the voltage appearing on the inverting input (Pin 2), otherwise Pin 6 will present a **low**.



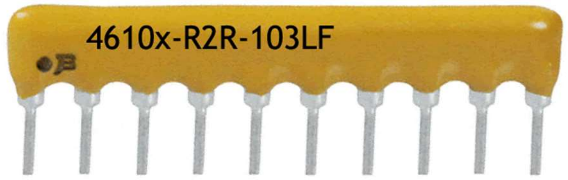
The AVR family of MCUs has a built-in comparator that can be accessed.





4.1 DAC: Digital to Analog Conversion (DAC)

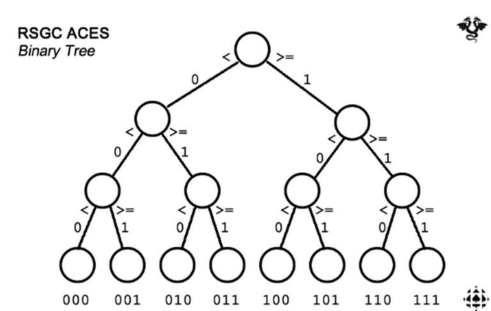
Of all the features that microcontrollers offer, a strong case could be made for Analog to Digital Conversion being its most important function. After all, the ability to capture real world data and digitize it for manipulation, transmission, and storage purposes is an undeniably critical feature within our modern world.



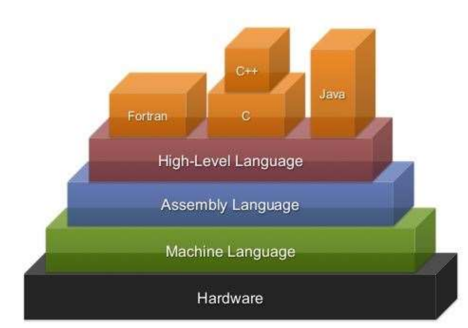
The fundamentals of how a DAC works can be vividly explored through the use of a passive resistor network known as an R2R Ladder, which we shall undertake. Although the AVR line of 8-bit MCUs offers a 10-bit onboard DC unit that we've exploited for a number of purposes, what if our needs called for either a higher or lower sampling accuracy? A deeper understanding of how the ADC function works is called for should we wish to build our own ADC unit.

4.2 Successive Approximation

An informative base from which to mount our investigation might start with a somewhat familiar **binary tree**. As a child you might have engaged in a guessing game in which a series of ranked guesses with a response of either lower or higher could lead you to your target. Indeed, this approach could lead to a conversion method from *decimal* to *binary* as suggested by the labeled paths.



5 Preparations for AVR Assembly Language Programming (AALP)



The past year-and-a-half has prepared you for a journey very few secondary school students are able to undertake. That is, descend to the deepest levels possible of a modern microcontroller. In fairness, the *deepest* level an embedded system programmer can go is to program in *Machine* Language (aka. *binary* or *hexadecimal*). Since this is barely readable by humans the numeric codes are assigned 2-4 letter mnemonic names to make them reasonably understandable while taking nothing away from their efficiency. This set of codes is the focus of this course and is known as *Assembly* Language. Here is a list of the top 10 most popular computer languages as of June 2021,

Jun 2021	Jun 2020	Change	Programming Language	Ratings	Change
1	1		C	12.54%	-4.65%
2	3	^	Python	11.84%	+3.48%
3	2	v	Java	11.54%	-4.56%
4	4		C++	7.36%	+1.41%
5	5		C#	4.33%	-0.40%
6	6		Visual Basic	4.01%	-0.68%
7	7		JavaScript	2.33%	+0.06%
8	8		PHP	2.21%	-0.05%
9	14	^^	Assembly language	2.05%	+1.09%
10	10		SQL	1.88%	+0.15%

Embedded systems is the computer engineers' term for modern smart devices. Microcontrollers lie at the heart of these systems and, in order to maximize their performance, you must speak their native language.

With the exception of C and Assembly the remaining eight languages are high-level tools designed to run on operating systems that hide keep the hardware efficiencies out of site for their practitioners.

Each microcontroller or microprocessor line (from AVR, PIC, NXP, Intel, etc.) has its own native machine and assembly language.

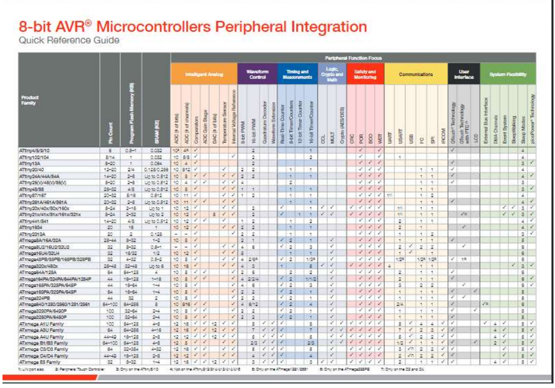
5.0 Development Preparations

5.0.0 Hardware: Atmel/Microchip AVR Microcontrollers

Up until recently, two microcontroller companies dominated the marketplace. ATMEL backed its AVR line of MCU products and Microchip championed its PIC family. The two companies merged in April 2016 under the Microchip name, continuing to offer both products. Until such time as the Arduino ceases to use the AVR line as its microcontroller of choice, RSGC ACES will stick with it.

http://www.microchip.com/design-centers/8-bit/avr-mcus

5.0.0.0 Peripheral Integration



All AVR microcontrollers share the same assembly language consisting of approximately 130 different instructions. A handful of instructions are MCU-specific.

Although our course focuses on the ATmega328P, ATtiny84, and ATtiny85, you should not feel limited to these alone for your particular application. You are encouraged to explore the 8-bit AVR MCU Peripheral Integration document to choose just the right combination of features for your embedded system. There are a wide variety of options for you to choose from that will suit almost any application.

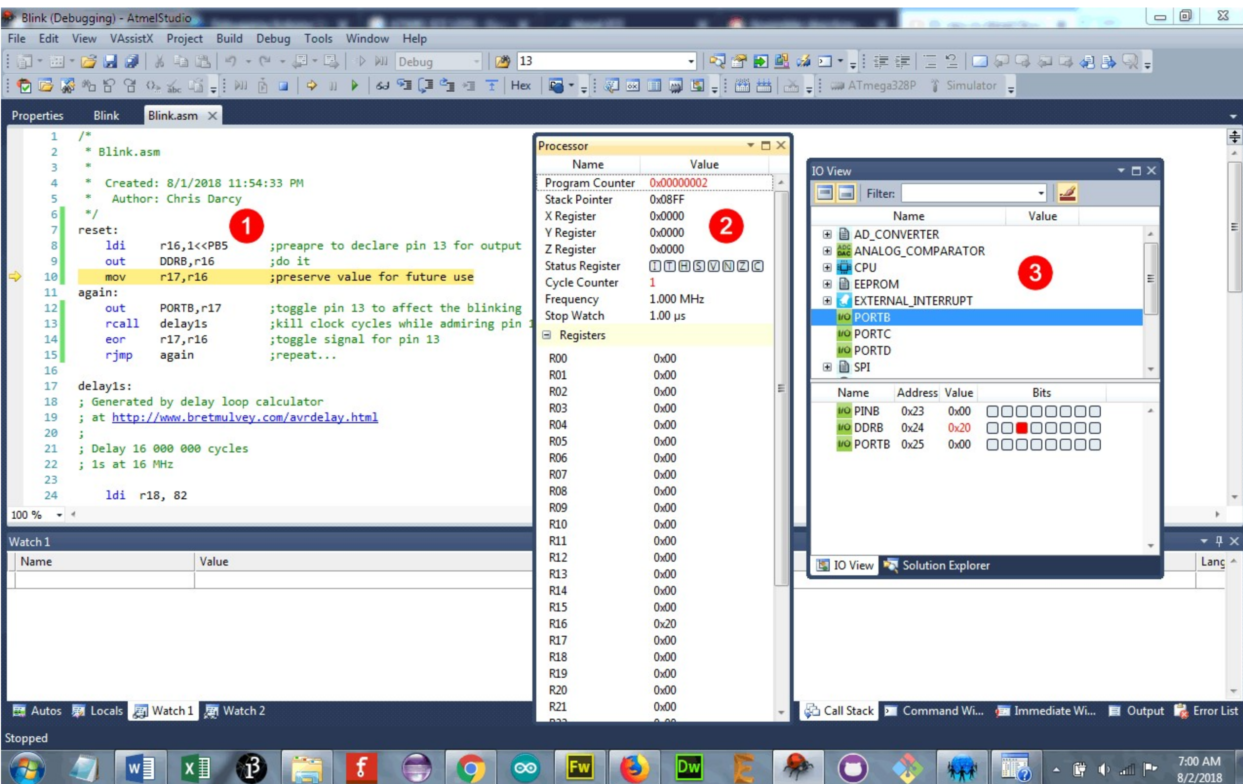
http://ww1.microchip.com/downloads/en/DeviceDoc/30010135D.pdf

5.0.1 Software Development Tools

5.0.1.0 Integrated Development Environment: Atmel Studio 7

ATMEL, the manufacturers of the AVR line of microcontrollers have developed the most comprehensive IDE for their MCUs. The latest version, ATMEL Studio 7, offers the richest, most professional, set of tools for AVR embedded systems development.

Note. Since I run Windows 7 on my laptop, I am limited to Atmel Studio 6 as the screenshots reflect.



The screenshot above is of Blink-like code running in the AS6 **Simulator**. Numbered panels are as follows,

1. **Source code**.
2. **Processor View**. Shows the contents of the General Purpose Registers and selected Extended Registers reflecting the flow of control after each statement execution.
3. **IO View**. Shows the state of the peripherals and IO Ports after each statement execution.

There is so much ahead of us, but what is unique to note at this early stage of the course is the intimate relationship between the assembly code statements and the hardware. It is only through deliberate precision that your code has on the hardware that the absolute efficiency required of your embedded systems can be achieved.

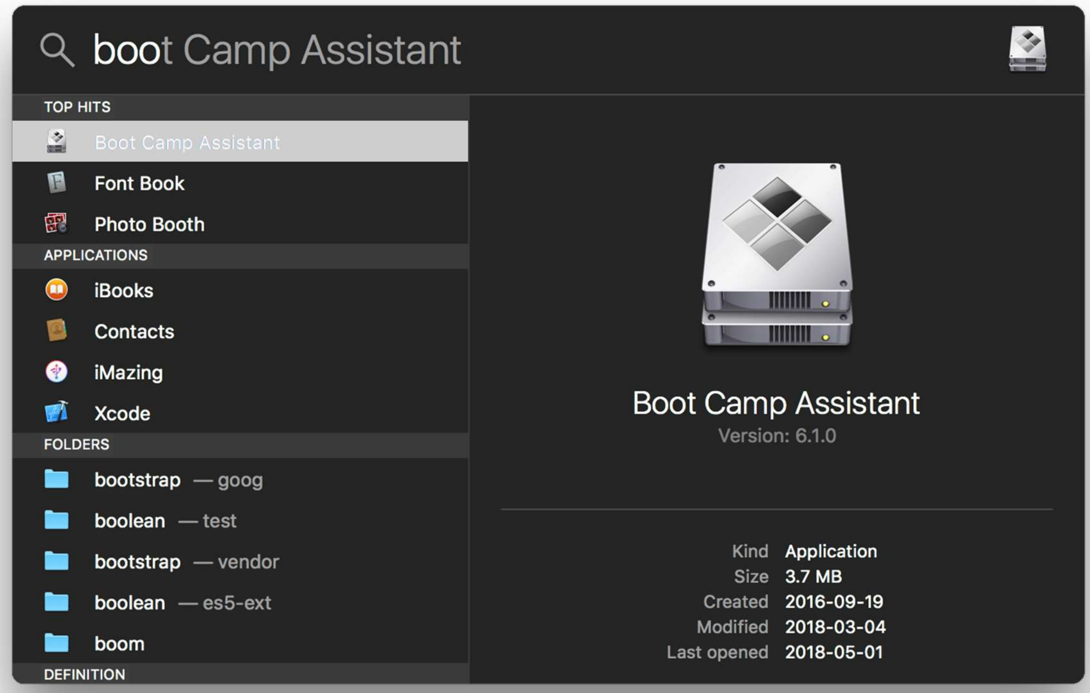
5.0.1.1 Operating System: Windows 10

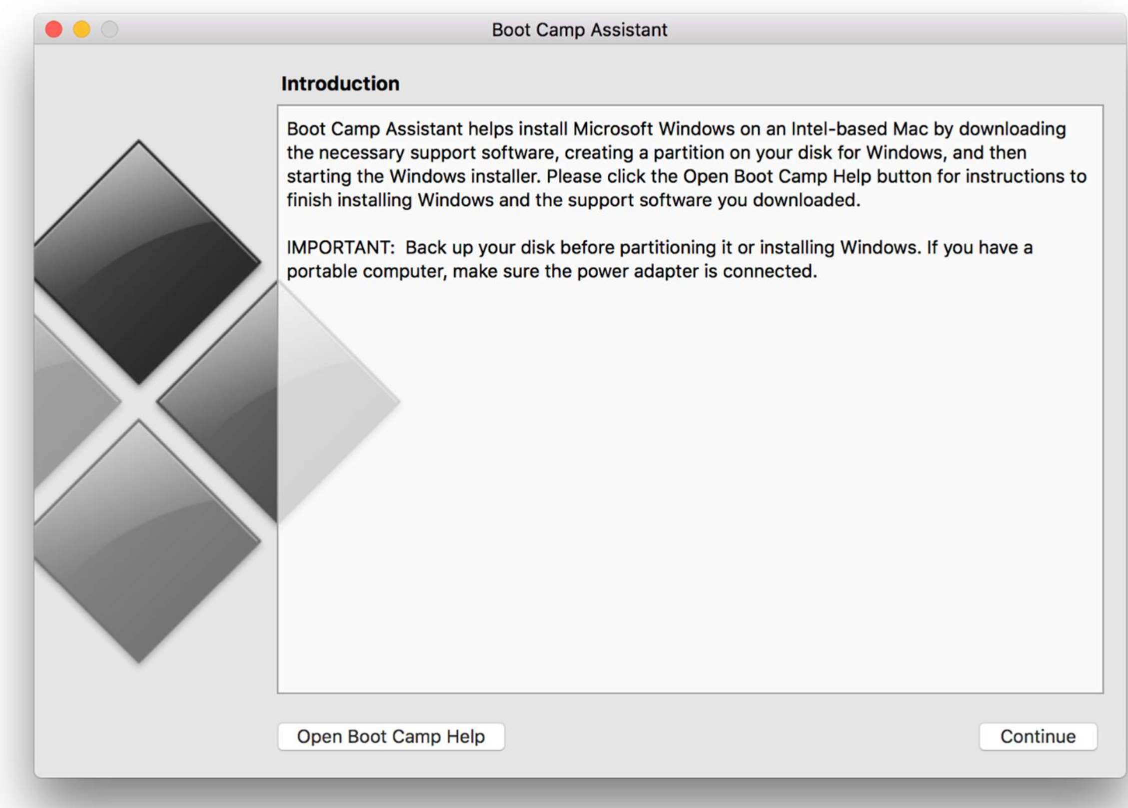
Unfortunately, only a Windows version of ATMEL Studio 7 exists. So, you need to install Windows 10 on your laptop. Ethan Peterson (ACES '18, Queen's '22) was kind enough to assemble the following installation guide for your convenience...

Before You Begin:

- Your Mac (Connected to its charger)
- A USB Flash Drive to act as your Windows installation media
  - 16GB or larger
  - Should be completely blank. If not, backup your files elsewhere and erase the flash drive. Depending on the file system, the flash drive may need to be reformatted using Disk Utility as a MS-FAT volume using Master Boot Record (MBR)
- Download a copy of the Windows 10 ISO Image. (https://www.microsoft.com/en-us/software-download/windows10)
- Make a backup of important files on the Mac side of your computer. This is needed in case the installation goes wrong.

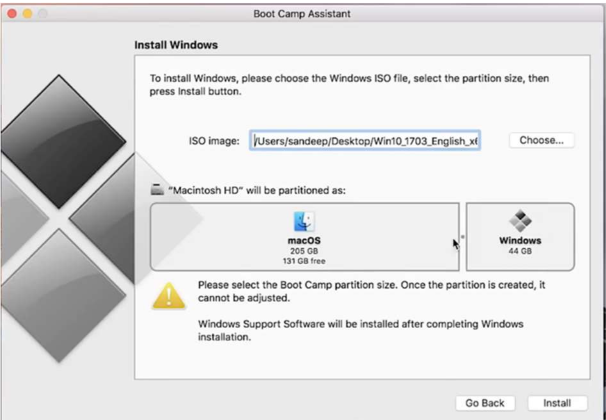
Step 1: Open BootCamp Assistant





- Click "Continue"

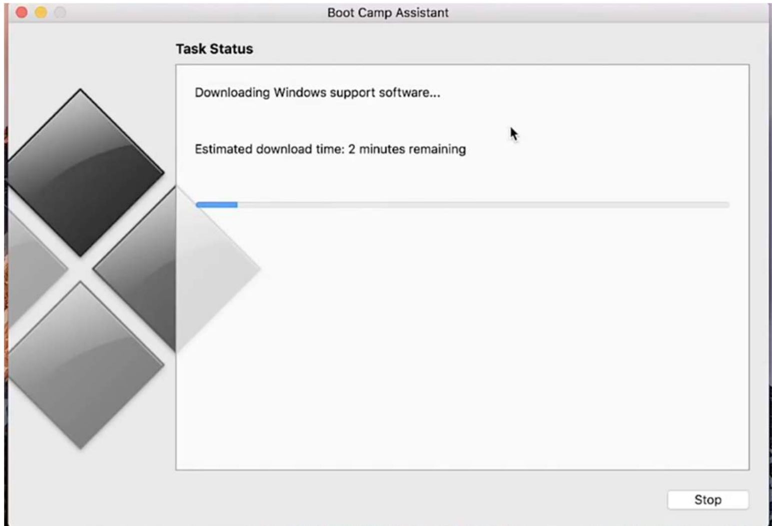
Step 2: Format your Windows Partition



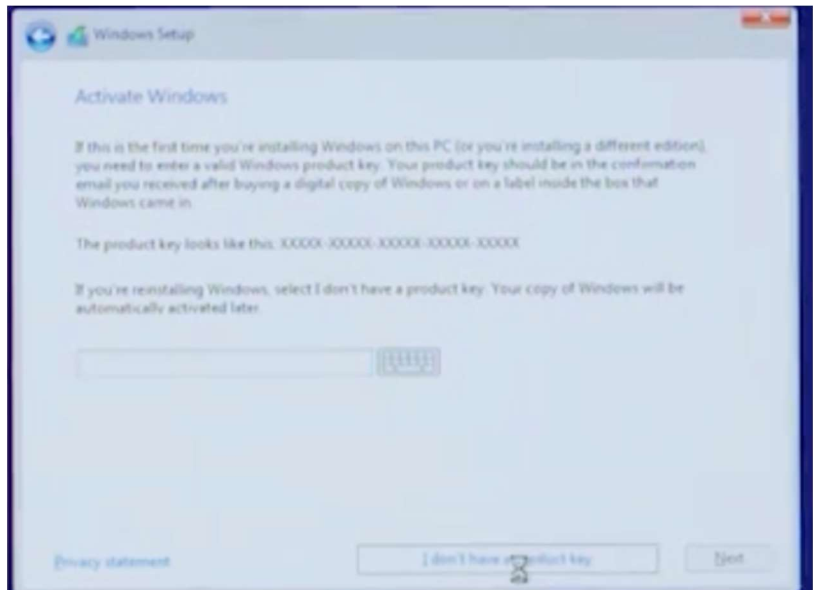
- Depending on the age of your MacBook the next screen in the BootCamp Wizard will be different.
- If you are prompted to connect a USB flash drive go ahead and do so. If not, your laptop will store the windows installation media internally.
- Select the location of the Windows 10 ISO file you downloaded as shown above.
- Select a size for the Windows partition on your computer. The minimum size is 40GB, which should be more than enough for your AVR Studio projects throughout the year. Depending on your MacBook, this minimum may be different.
- This partition cannot be adjusted later so it is better to reserve additional space if you think you may need it.

Step 3: Installing Windows

- Unplug any external devices from your laptop that are not critical to the installation, such as an external keyboard, mouse and hard drive, as they can interfere with the installation. Leave only the charger and USB connected if it is required and click "Install" on the BootCamp Wizard.



- BootCamp will start downloading the Windows Support Software drivers.
- These drivers are installed in Windows to ensure the keyboard, trackpad and other peripherals on your Mac run correctly.
- When the download is complete your computer will prompt you to restart.
- The computer should automatically boot into the Windows installer.
- Follow the onscreen instructions



- When the "Activate Windows" screen is reached, click "I don't have a product key" and select the Pro edition of Windows.
- Click "Next" and accept the terms and conditions
- If prompted, pick the partition called "BOOTCAMP" as your installation destination.
- Once Windows is done installing, the system will reboot into your fresh installation of Windows. If your computer boots into OSX, shut it down and hold the option key while starting up. When the computer prompts you for what OS you would like to use, select Windows.
- When Windows is started for the first time you will have to configure some settings. Follow the onscreen instructions for this.
- When you reach the Windows Desktop, the Windows Support Software installer should open and guide you through the driver installation. If not, you will have to install it manually from your installation media.
    - Follow the instructions here: https://support.apple.com/en-ca/HT208495
    - If you used a USB flash drive for installation, plug it back in and get the software from there as opposed to the OSXRESERVED Partition on Windows.

Step 4: Install Atmel Studio

- Download Atmel Studio 7: http://www.microchip.com/mplab/avr-support/atmel-studio-7
- Select the offline installer
- Open the file and follow the installation prompts.

5.0.1.2 Programmer: Atmel ICE

Currently, the best programmer for use with ATMEL Studio 7 is the ATMEL Basic ICE. These are expensive programmers so you will be lent one for use this year.

Return it in working order at the end of the year and its use is free of charge. Caution. For such an expensive device the ribbon cable is surprisingly cheesy.

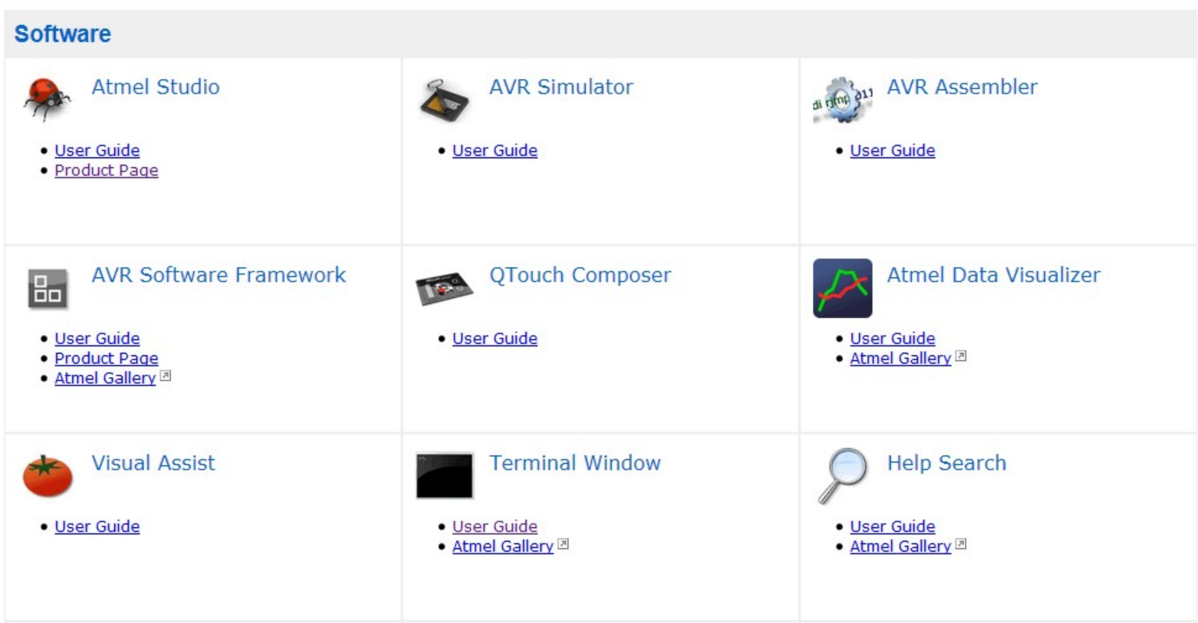
Strangely is does not provide its own power. You must supply power to your Arduino separately.

https://www.digikey.ca/product-detail/en/microchip-technology/ATATMEL-ICE-BASIC/ATATMEL-ICE-BASIC-ND/4753381

5.1 Microchip's Online Reference

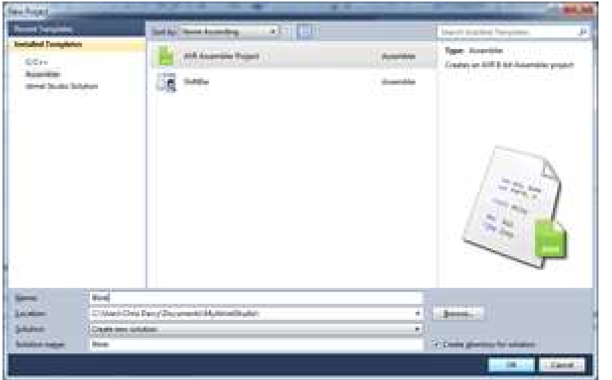
Learning Assembly Language and gaining familiarity with the tools takes a time and practice. Online support is available and I recommend starting any search for insight at Microchip's online home page. You may even wish to bookmark this URL,

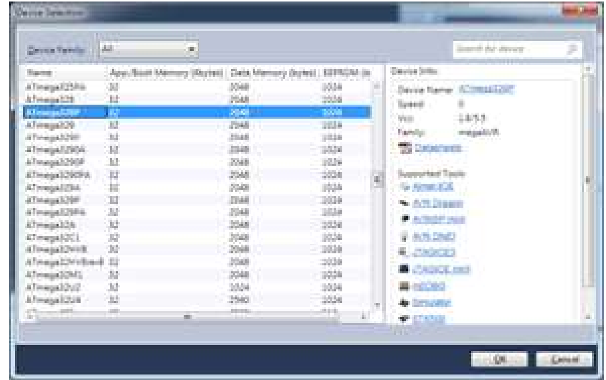
https://www.microchip.com/webdoc/index.html



5.2 New Atmel Studio Project

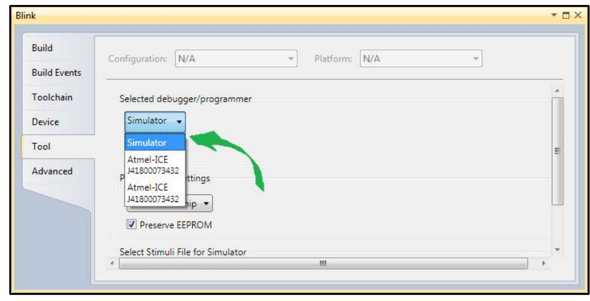
1. Before beginning your first project create a folder to house your ICS4U Assembly projects.
2. Launch `File > New > Project` and complete the dialog as shown below, left. Press, OK.
3. In the `Device Selection` dialog, select the target device (ATmega328P).





5.3 Your First Project: Blink

5.3.0 Simulator

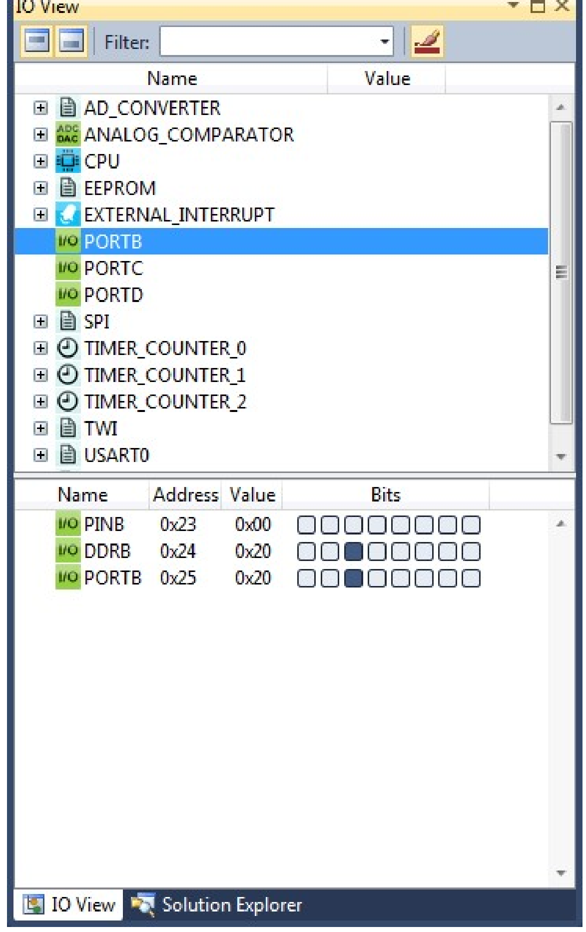


Without an Arduino plugged into your laptop, the **Arduino IDE** allows you to *compile* your code to stabilize your syntax but, understandably, prevents you from uploading to test if it works.

Atmel Studio *does* allow you to execute your code without a physical MCU being present. It does so through the services of a built-in Simulator utility.

Select `Project > (project) Properties > Tools > Simulator`

5.3.1 Hardware



Launching the Atmel Studio Simulator allows you to step through your assembly code, one statement at a time, and monitor the MCU hardware. The **IO View**, shown to right is where the detailed status of each peripheral can be followed.

Checking the pin mapping diagram inside the cover of this workbook reveals that pin 13 is mapped to bit 5 of PORTB. **Terminology,**

To **set** a bit, means to make it 1

To **clear** a bit means to make it 0

So, a blinking LED on Arduino pin 13 is the result of two actions,

a) The direct **setting** bit 5 of the DDRB (Data Direction Register of PORTB), and,

b) Applying a square wave signal (alternate **setting** and **clearing**) to bit 5 of PORTB.

5.3.2 Software

Create a new Atmel Studio Assembly project and name it `Blink`. Download the source code `Blink.asm` and replace the default `Blink.asm` that was create for you in `Debug` folder.

`rsgcaces > AVROptimization > 1_Getting_Started > Blink.asm`

5.3.2.0 Source Code Appearance

A quick glance that assembly source code of the Blink program reveals some standard features of the IDE that include,

1. Syntax highlighting (green for comments, blue for keywords)
2. Line numbers
3. Tab stops
4. Live hyperlinks

```
Blink.asm
 1  /*
 2   * Blink.asm
 3   *
 4   *  Created: 8/1/2018 11:54:33 PM
 5   *   Author: Chris Darcy
 6   */
 7  reset:
 8      ldi     r16,1<<PB5      ;preapre to declare pin 13 for output
 9      out     DDRB,r16        ;do it
10      mov     r17,r16         ;preserve value for future use
11  again:
12      out     PORTB,r17       ;toggle pin 13 to affect the blinking
13      rcall   delay1s         ;kill clock cycles while admiring pin 13
14      eor     r17,r16         ;toggle signal for pin 13
15      rjmp    again           ;repeat...
16
17  delay1s:
18  ; Generated by delay loop calculator
19  ; at http://www.bretmulvey.com/avrdelay.html
20  ;
21  ; Delay 16 000 000 cycles
22  ; 1s at 16 MHz
23
```

5.3.2.1 Assembly Source Code

Unlike high level source code, executable statements in AVR Assembly follow one of four a common syntactic structure (square brackets indicate an optional element),

```
[label:]  instruction    [operands]     [Comment]

[label:] directive [operands] [Comment]

Comment

Empty line
```

1. A label provides a symbol that is the target of branch or a variable.
2. An instruction is a 2-4 letter mnemonic opcode.
3. Zero, one, or two operands provide the domain of the instruction.
4. A preprocessor directive (starts with a #) and an assembler directive (starts with a dot, .) are requests for some preparatory action prior to the assembler converting your code to machine language.
5. A comment illuminates the purpose of the statement and either appears at the end of the statement, or starts in Column 1 and occupies a line by itself. The syntax below confirms that a semicolon alone qualifies as a comment.

   ```
   ; [Text]
   ```

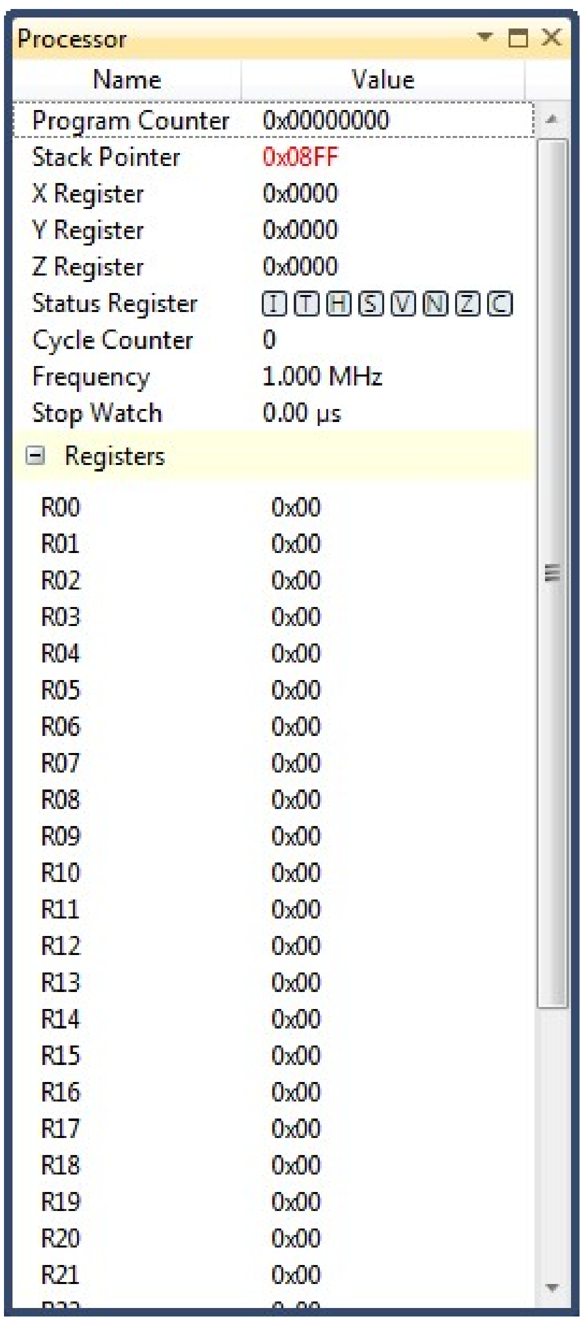
5.3.3 Debugging Blink.asm

Once the `Blink` Project has been established, and the Simulator declared as the default debugger/programmer Tool,

```
Project > Blink Properties
```

debugging is initiated by going to the Build menu and selecting,

```
Start Debugging and Break (Alt-F5)
```



At that moment you will see the first executable instruction line highlighted in yellow and Processor Window displayed (right). A few things to note about the Processor Window,

1. The **Program Counter** always hold the address of the instruction to be executed.
2. The **Stack Pointer** holds the address of the top of the Stack. The Stack starts at the end of SRAM (0x8FF) and grows upwards (more on this later).
3. The **X**, **Y**, and **Z** registers are aliases for the double 16-bit register combinations **R26:R27**, **R28:R29**, and **R30:R31**, respectively.
4. The **Status Register** (SREG) is a special byte register in SRAM (0x5F) consisting of a set of bits (aka flags) that are either set, cleared of left unchanged by the previously executed statement. This byte can be read by your code to determine whether a course of action needs to be taken.
5. The contents of the CPU's 32 General Purpose registers are updated dynamically to facilitate your debugging objectives.

1.3.3.0 Stepping and Breakpoints

Once the simulator is underway you have a options to step through your code, statement by statement monitoring the effect on the processors resources. Breakpoints can be toggled on or off by clicking in the left margin.

The IO View can also be displayed allowing to both read and write data on the fly.

5.4 ATmega328P Features

http://mail.rsgc.on.ca/~cdarcy/Datasheets/ATmega328PSummary.pdf

ATmega48A/PA/88A/PA/168A/PA/328/P

ATMEL 8-BIT MICROCONTROLLER WITH 4/8/16/32KBYTES IN-SYSTEM PROGRAMMABLE FLASH

Features

- High Performance, Low Power Atmel®AVR® 8-Bit Microcontroller Family
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
  - 4/8/16/32KBytes of In-System Self-Programmable Flash program memory
  - 256/512/512/1KBytes EEPROM
  - 512/1K/1K/2KBytes Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C[1]
  - Optional Boot Code Section with Independent Lock Bits
    - In-System Programming by On-chip Boot Program
    - True Read-While-Write Operation
  - Programming Lock for Software Security
- Atmel® QTouch® library support
  - Capacitive touch buttons, sliders and wheels
  - QTouch and QMatrix® acquisition
  - Up to 64 sense channels
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Six PWM Channels
  - 8-channel 10-bit ADC in TQFP and QFN/MLF package
    - Temperature Measurement
  - 6-channel 10-bit ADC in PDIP Package
    - Temperature Measurement
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Byte-oriented 2-wire Serial Interface (Philips I²C compatible)
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
  - Interrupt and Wake-up on Pin Change

- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby
- I/O and Packages
  - 23 Programmable I/O Lines
  - 28-pin PDIP, 32-lead TQFP, 28-pad QFN/MLF and 32-pad QFN/MLF
- Operating Voltage:
  - 1.8 - 5.5V
- Temperature Range:
  - -40°C to 85°C
- Speed Grade:
  - 0 - 4MHz@1.8 - 5.5V, 0 - 10MHz@2.7 - 5.5.V, 0 - 20MHz @ 4.5 - 5.5V
- Power Consumption at 1MHz, 1.8V, 25°C
  - Active Mode: 0.2mA
  - Power-down Mode: 0.1µA
  - Power-save Mode: 0.75µA (Including 32kHz RTC)

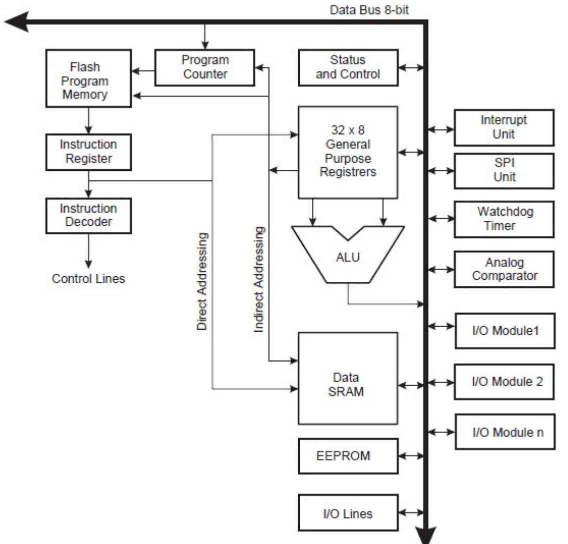
5.5 Peripherals

What separates a microcontroller from a microprocessor is that the format has a number of internal peripherals. Here is a partial list of those on the ATmeg328P,

- ADC: Analog-to-Digital Converter
- Timers/Counters: 8- and 16-bit
- PWM: 8- and 16-bit Pulse Width Modulation
- Temperature Sensor
- Internal Voltage Regulator
- Multiplication: Dedicated hardware for multiplying two 8-bit values with 16-bit result
- USART, I²C, SPI
- QTouch (Capacitive Touch Sensor) Support, Sleep Modes

5.6 AVR Central Processing Unit (CPU)

Just before we take a detailed look at the structure of AVR Assembly Source code in the next chapter, it is instructive to familiarize yourself with how the CPU works.



The AVR Central Processing Unit consists of a number of different modules interconnected through a number of buses. The 8-bit **data** bus is highlighted by the thicker line. The **address** bus and **control** bus are not shown. The execution cycle can be thought of as a repetition of three stages: **Fetch-Decode-Execute**.

Although it does not use the AVR as its hardware, sequence is identical in this terrific video,

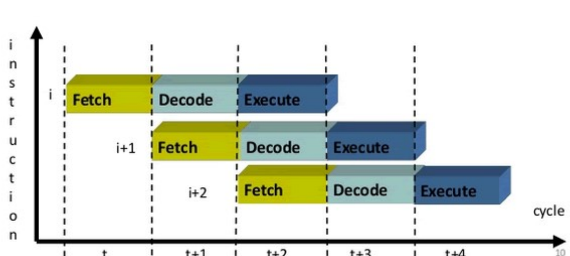
https://www.youtube.com/watch?v=XM4lGflQFvA

It's all quite fascinating but the **Decode** stage is worth a mention at this point. The first assembly instruction in the Blink code from Chapter 1 is,

Assembly Language	Machine Language Equivalent	Hexadecimal
`ldi r16,1<<PB5`	1110 0010 0000 0000	E200

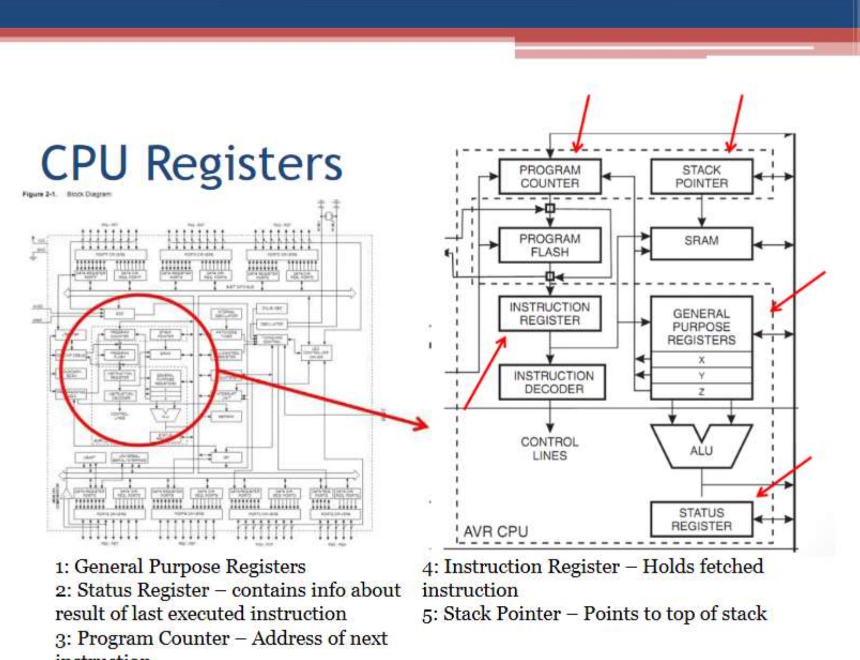
The decoder is a *combinational* hardware circuit that must accept the two byte (16-bit) input E020 and parse it in such a way that the rest of the CPU assets know to place the binary value of B0010 0000 in Register 16. To keep the Decoder to a manageable level of complexity, the number of possible instructions, and their complexity must be **reduced** to a minimum. These requirements give rise to the identification of the 8-bit AVR line of microcontroller as begin of a **RISC** (Reduced Instruction Set Computer) type.

Finally, a word about execution. Given a three-stage execution cycle, it might appear that instructions are only executed every third stage. However a strategy referred to as **pipelining** has the three stages functioning synchronously, resulting in a statement being executed every clock cycle.

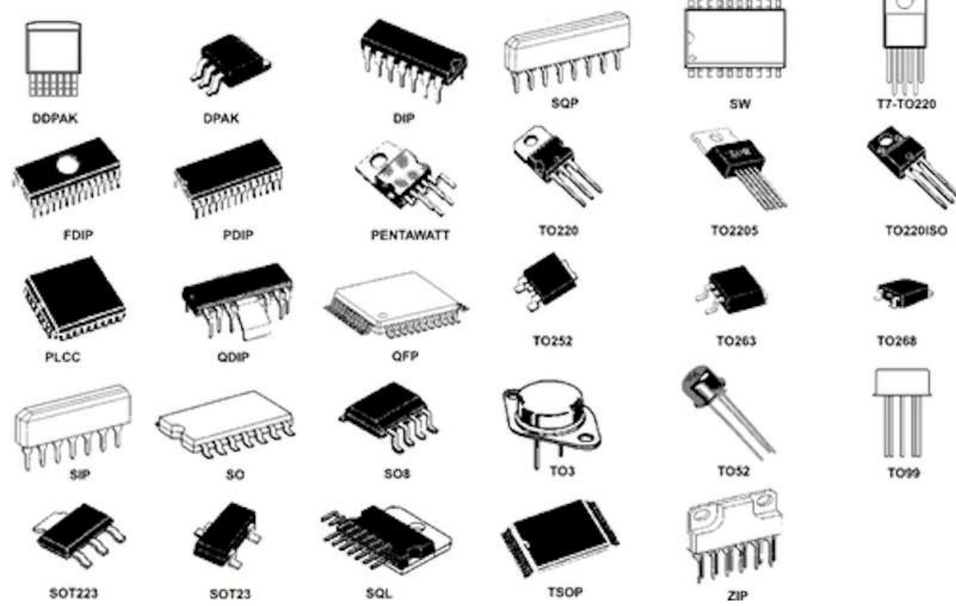


Reference: http://darcy.rsgc.on.ca/ACES/TEI4M/Assembly/AVRCPURegisters.pdf

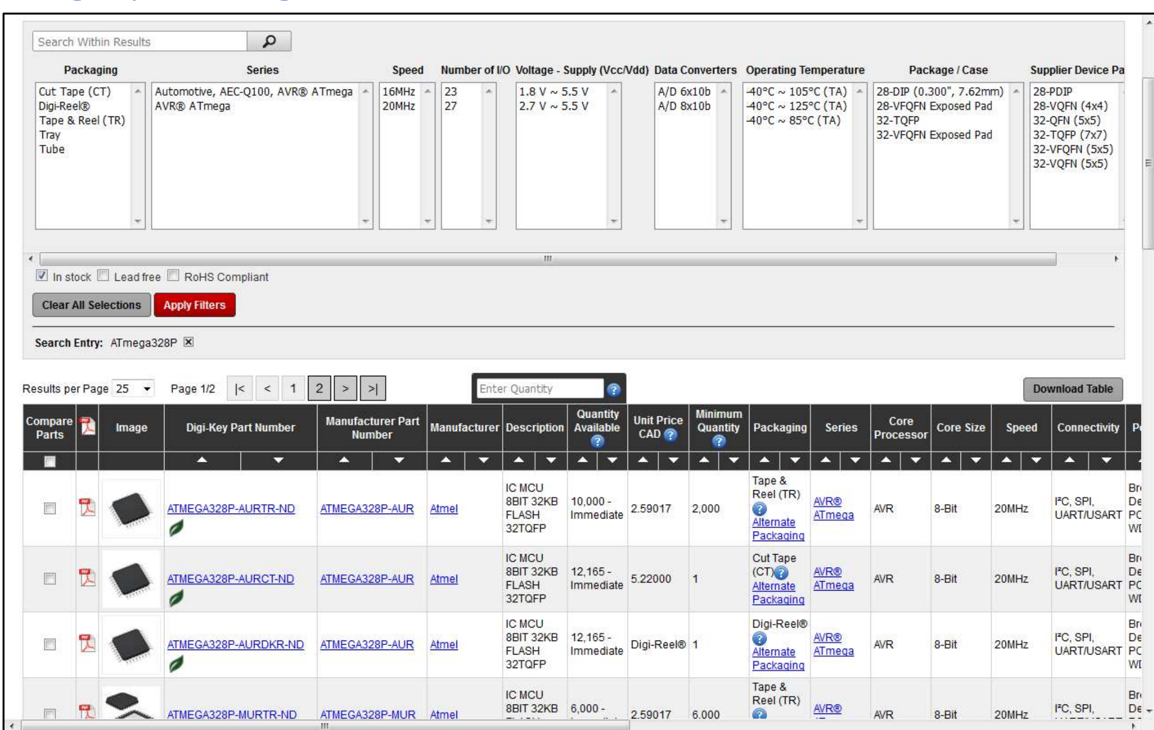
The image below offers a good summary of a number of features discussed to this point. It was taken from the short but informative pdf referenced above.



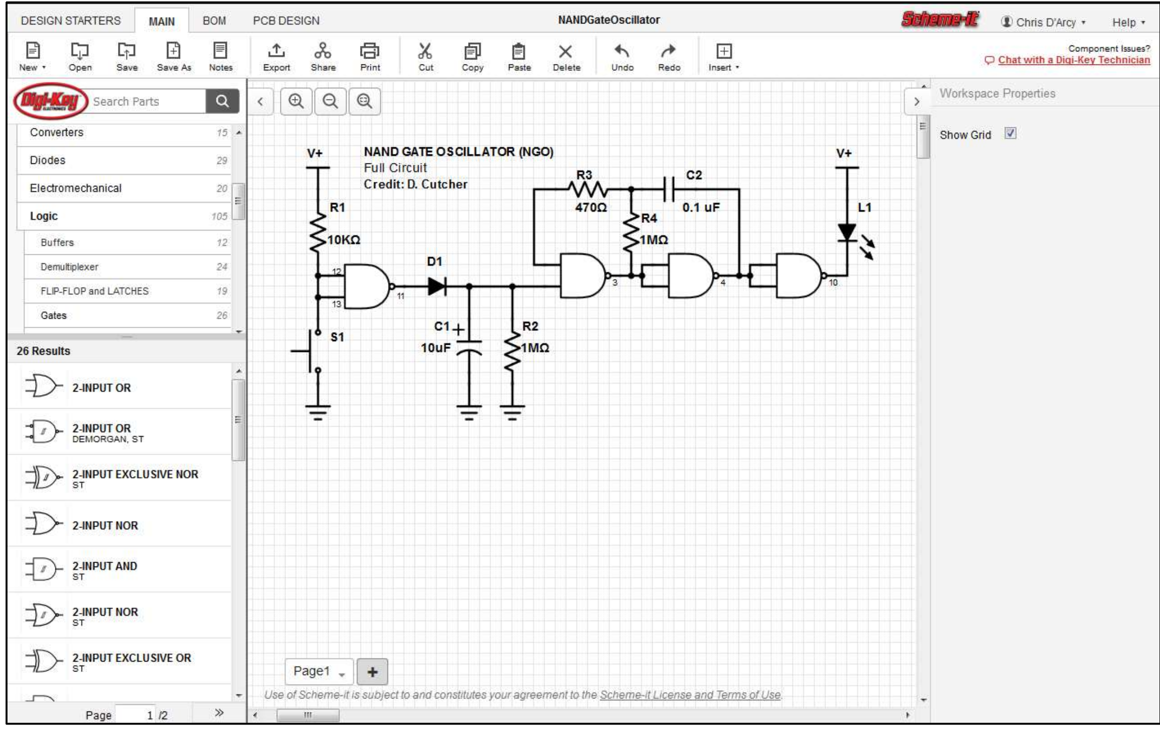
5.7 Package Types



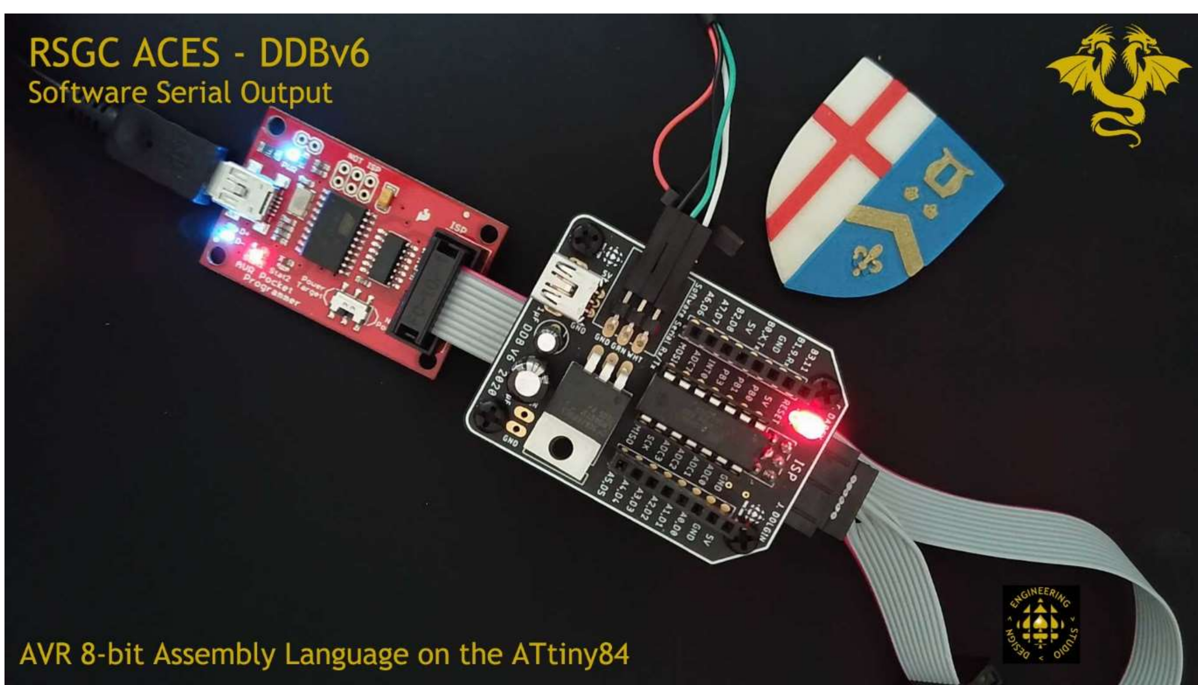
5.7.0 Digikey: Ordering



5.7.1 Digikey: SchemeIt



RSGC ACES: DDBv6 Software Serial Output



5.8 Interesting Exercises

5.8.0 Delay Calculator

Here is the link to the online version of Bret Mulvey's AVR Delay utility,

http://www.bretmulvey.com/avrdelay.html

A remarkable example of ACES insight and initiative was one afternoon in 2018 where I happened to mention that I thought Mulvey's calculator could be improved if the user was permitted to name the starting register for the sequence of delays. Nothing more was said in class. T. Morland (ACES '18) went home that afternoon and upgrade Mulvey's code that we prefer to link to, Amazing.

http://darcy.rsgc.on.ca/ACES/TEI4M/AVRdelay.html

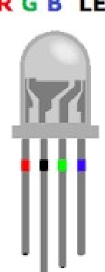
5.8.1 Traffic Light

Insert a Schaffer traffic light into your Arduino in such a way that all four pins land within a single PORT.

Create the project `TrafficLight` and model the solution to a continuous display after the `Blink` project code.

Comment your code thoroughly, but not gratuitously.

5.8.2 RGB LED



Insert an RGB LED into your Arduino in such a way that all four pins land within a single PORT.

Create the project `RGBLED` and develop a continuous display that runs through all eight combinations of LEDs (1 blank, 3 singles, 3 doubles, 1 triple).

Comment your code thoroughly, but not gratuitously.

5.8.3 Questions

1. ATmega328P has 32K worth of PROGRAM FLASH. What is the highest address in hexadecimal?

    ____

2. The ATmega328P has 2K worth of SRAM. What is the highest address in hexadecimal?

    ____

2. The ATmega328P has 1K worth of EEPROM. What is the highest address in hexadecimal?

    ____

5.9 Just Before We Start: C

```
/*
 * CBeast.c
 * Example of AVR-gcc C Code to present the
`* value of pi on 'The Beast'.'The Beast' is a
 * PCB designed to present a 12-digit
 * PoV display using a normal 595 shift registers
 * for the segment and TPIC6C595 Power logic
 * current sink for the LA-301 cathodes
 * Created: 8/10/2018 1:10:28 PM
 * Author: Chris Darcy */

#include <avr/io.h>
#define F_CPU 16000000UL  // 16 MHz
#include <util/delay.h>

uint8_t latch = 1<<PB2;          //digital pin 10
uint8_t clock = 1<<PB3;          //digital pin 11
uint8_t data = 1<<PB4;           //digital pin 12

// Seven Segment Order:dGFEDCBA
uint8_t dp = 1<<7;
//Assemble the hexadecimal segment maps into single array...
uint8_t  segMaps[] = {0b00111111,0b00000110, 0b01011011,
0b01001111,0b01100110,0b01101101,0b01111100,0b00000111,0b01111111,0b01100111,

0b01110111,0b01111100,0b00111001,0b01011110,0b01111001,0b01110001};

char display[] = "314159265539";  //Sample...

//my 'super' shiftout (23 bits are shifted in one go)
void shiftOut(uint8_t d, uint8_t c, uint8_t l, uint32_t segMap, uint32_t digit ){
	uint32_t bits = (segMap<<16) | digit;
	// assemble the 23-bit shift out value from the segMap and the respective digit
	//pull latch low...
	PORTB &= ~latch;
	//synchronously clock in the data bits
	for (uint32_t mask=1L<<23; mask>0; mask>>=1)   {
		PORTB &= ~clock;
		// should the data bit be set or clear?
		if (bits & mask)
			PORTB |= data;
		else
			PORTB &= ~data;
		PORTB |= clock;
	}
	//pull latch high to present present flipflops on the output pins...
	PORTB |= latch;
}

int main(void) {
  // Let's use three pins of portB for the shifting...
  DDRB = 0xFF;
  uint8_t i = 0;
  uint32_t segments;
  while(1) {
	segments = segMaps[display[i]-48];
	if (!i) segments |= dp;  //add decimal point on the 3 for pi
	shiftOut(data,clock,latch,segments,1L<<(11-i));
	i = (i+1) % 12;
  }}
```

6 AALP: AVR Assembly Language Programming

A clarification about the terminology, *assembly* and *assembler*. Whereas some sources prefer to use the terms interchangeably, I do not. I use the term *assembly* to refer to the mnemonic-based language of the CPU. I use to the term assembler, when I am referring to the **program** that converts code written in assembly (`.asm`) into machine code (`.hex`)

6.1 Assembly Code Organization

Earlier in Chapter 1 the four possible varieties of assembly source code statements were reviewed.

In this section we'll tackle the *organization* of these statements within ATMEL Studio and the conventional layout of your `.asm` files.

1. **Comment.** At the top of your code a comment describing the purpose, author and date is expected

2. **Preprocessor**. Starting after the opening comment and continuing throughout the source file, a set of directives assist the assembler in building your final machine loadable (hex) file. These commands start with # as the first non-space character of which `#include "m328Pdef.inc"` would be one such directive. This directive is done automatically so it is optional.

3. **Assembler Directives**. Numerous directives that start with a dot, are recognized by the assembler to facilitate code organization, memory requests, aliases, and conditional execution to name a few. Some examples are,

```
.EQU io_offset = 0x23

.DSEG
var1:  .BYTE 1          ; reserve 1 byte to var1
table: .BYTE tab_size   ; reserve tab_size bytes

.DEF temp = r16
.DEF ior = r0
.CSEG
ldi     temp,0xf0       ; Load 0xf0 into temp register
in      ior,0x3f        ; Read SREG into ior register
eor     temp,ior        ; Exclusive or temp and ior
```

4. **Interrupt Jump Table**. As required, the first 30 addresses of **Program Flash** or so are reserved for the Interrupt Jump Table. Care should be taken when using this space. To avoid code

```
.CSEG
.ORG    0                      ;ensure PC starts at the beginning
        rjmp    reset
```

```
reset:  ldi     r16,1<<PB5      ;first instruction in your code
```

5. **One-Time Initializations (setup)**. Very much like the `setup()` function in Arduino C last year, include assembly code that needs to run only **once** at this point.

6. **Main Loop Body**. Include assembly instructions that run continuously.

7. **Additional Functions and Interrupt Service Routines**.

6.2 Reusable Building Blocks

The code fragments below offer common building blocks you will use repeatedly.

1. **Numeric constants** can be defined as binary, octal, decimal (default) or hexadecimal. Here is how you would represent the base 10 number 100 in each of the other three bases,

   Binary (Base 2, leading zero): `0B01100100, 0b01100100`
   Octal (Base 8, leading zero): `0144`
   Hexadecimal (Base 16): `0x64, $64`

2. **The shift left** (<<) operator offers an efficient expression resulting in the setting a specific bit initialization of a byte.

```
ldi   r16,1<<3          ;B00001000
ldi   r17,1<<PB5        ;B00100000
ldi   r20,7<<4          ;B01110000
```

3. **Non-consecutive bits** within a byte are set with the `or` operator (|).

```
ldi   r16, (1<<ICIE1) | (1<<TOIE1) ;falling edges in ICR1H:L
```

4. **Labels as operands**. Use of the .ORG directive assists in laying out your code and data in memory. Labels are aliases for their location in memory and can be used as such as operands.

5. **Initializing the pointer registers** (`X`, `Y`, and `Z`) with the starting address of an array is as follows,

```
ldi   XL,low(RPMStart<<1)   ;position X and Y pointers to the
ldi   XH,high(RPMStart<<1)  ;start and end addresses of RPM array
```

6. **Toggling a specific bit** is best accomplished with the **exclusive-or**, `eor`. Consider how the **mask**, 0b0010000 can be applied repeatedly to toggle bit 5 of the **data**,

   data: `0b11`**`1`**`11111`

   mask: `0b00`**`1`**`00000`

data^mask = data: `0b11`**`0`**`11111`

mask: `0b00`**`1`**`00000`

data ^mask=data: `0b11`**`1`**`11111`

mask: `0b00`**`1`**`00000`

data ^mask=data: `0b11`**`0`**`11111`

7. Understand the difference between logical NOT and bitwise NOT.

Logical: `ldi r16,!0xf0 ; Load r16 with 0x00`

Bitwise: `ldi r16,~0xf0 ; Load r16 with 0x0f`

6.3 Basic Instructions by Function

The complete list of functions either in summary or full descriptions is available off links at the top of our home page.

6.3.0 Register Setting

```
clr   Rd    ;clears a register (0 ≤ d ≤ 31). Same as eor Rd,Rd
ser   Rd    ;sets a register to 255 (16 ≤ d ≤ 31). Same as ldi Rd,$FF
ldi   Rd,K  ;loads the constant K into Rd (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)
```

6.3.1 Copying

```
mov   Rd,Rs      ;copy contents of Rs to Rd
in    Rd,port    ;read the port contents into Rd
out   port,Rs    ;write the contents of Rs to the port
lds   Rd,K       ;load the contents of address K in SRAM into Rd
sts   K,Rs       ;store contents of Rs in address K of SRAM
lpm              ;load the contents of address pointed to by Z into R0
pop   Rd         ;copy the contents of top of the Stack to Rd
push  Rs         ;copy the contents of Rs onto the top of the Stack
```

6.3.2 Adding

```
inc   Rd         ;add 1 to Rd (rollover)
add   Rd,Rs      ;add Rs to Rd (no carry:C flag)
adc   Rd,Rs      ;add Rs to Rd (with carry:C flag)
adiw  Rd,K       ;add K (0 ≤ K ≤ 63) to Rd+1:Rd (d ∈ {24,26,28.30})
```

6.3.3 Subtracting

```
dec   Rd          ;subtract 1 from Rd (rollover 0-1=255)
sub   Rd,Rs       ;store the difference Rd-Rs in Rd
subi  Rd,K        ;store the difference Rd-K in Rd (16 ≤ d ≤ 31)
sbc   Rd,Rs       ;store the difference Rd-Rs in Rd with carry
sbci  Rd,K        ;store the difference Rd-K in Rd with carry
```

6.3.4 Shift & Rotate

```
lsl   Rd          ;logical shift left: C←Bit7,Bitn+1←Bitn,Bit0←0
lsr   Rd          ;logical shift right: 0→Bit7,Bitn+1→Bitn,Bit0→C
rol   Rd          ;rotate left: C←Bit7,Bitn+1←Bitn,Bit0←C
ror   Rd          ;rotate right: C→Bit7,Bitn+1→Bitn,Bit0→C
asr   Rd          ;arithmetic shift right: Bit7→Bit7,Bitn+1→Bitn,Bit0→C
```

RSGC ACES: Charlieplex Audio-Responsive Equalizer

6.3.5 Binary

```
and   Rd,Rs      ;logical AND: Rd←Rd&Rs
andi  Rd,K       ;logical AND: Rd←Rd&K,  (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)
or    Rd,Rs      ;logical OR: Rd←Rd|Rs
ori   Rd,K       ;logical OR: Rd←Rd|K,  (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)
eor   Rd,Rs      ;Exclusive OR: Rd←Rd^Rs
com   Rd         ;One's Complement: Rd←~Rd or Rd←$FF-Rd
neg   Rd         ;Two's Complement: Rd←~Rd+1 or Rd←0-Rd
```

6.3.6 Bit Manipulation

```
sbr   Rd,K  ;sets various bits: Rd←Rd|K,  (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)
cbr   Rd,K  ;clears various bits: Rd←Rd&~K,  (16 ≤ d ≤ 31, 0 ≤ K ≤ 255)
sbi   A,b   ;sets bit in I/O port: A←A|(1<<K),  (0 ≤ A ≤ 31, 0 ≤ b ≤ 7)
cbi   A,b   ;clears bit in I/O port: A←A&~(1<<K),  (0 ≤ A ≤ 31, 0 ≤ b ≤ 7)
```

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x15 (0x35)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0	
0x14 (0x34)	Reserved	–	–	–	–	–	–	–	–	
0x13 (0x33)	Reserved	–	–	–	–	–	–	–	–	
0x12 (0x32)	Reserved	–	–	–	–	–	–	–	–	
0x11 (0x31)	Reserved	–	–	–	–	–	–	–	–	
0x10 (0x30)	Reserved	–	–	–	–	–	–	–	–	
0x0F (0x2F)	Reserved	–	–	–	–	–	–	–	–	
0x0E (0x2E)	Reserved	–	–	–	–	–	–	–	–	
0x0D (0x2D)	Reserved	–	–	–	–	–	–	–	–	
0x0C (0x2C)	Reserved	–	–	–	–	–	–	–	–	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	92
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	92
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	92
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	91
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	91
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	92
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	91
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	91
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	91
0x02 (0x22)	Reserved	–	–	–	–	–	–	–	–	
0x01 (0x21)	Reserved	–	–	–	–	–	–	–	–	
0x0 (0x20)	Reserved	–	–	–	–	–	–	–	–	

6.3.7 Compare

```
cp    Rd,Rs      ;Form Rd-Rs to influence SREG Flags (0 ≤ d ≤ 31)
cpi   Rd,K       ;Form Rd-K to influence SREG Flags (16 ≤ d ≤ 31)
tst   Rs         ;influence N and Z SREG flags based on Rs (0 ≤ d ≤ 31)
```

6.3.8 Jump: Unconditional

```
rjmp  K          ;unconditional branch ±2K words from current address
rcall K          ;call to subroutine ±2K words from current address
ret              ;return from subroutine (STACK operation)
reti             ;return from interrupt (STACK operation)
```

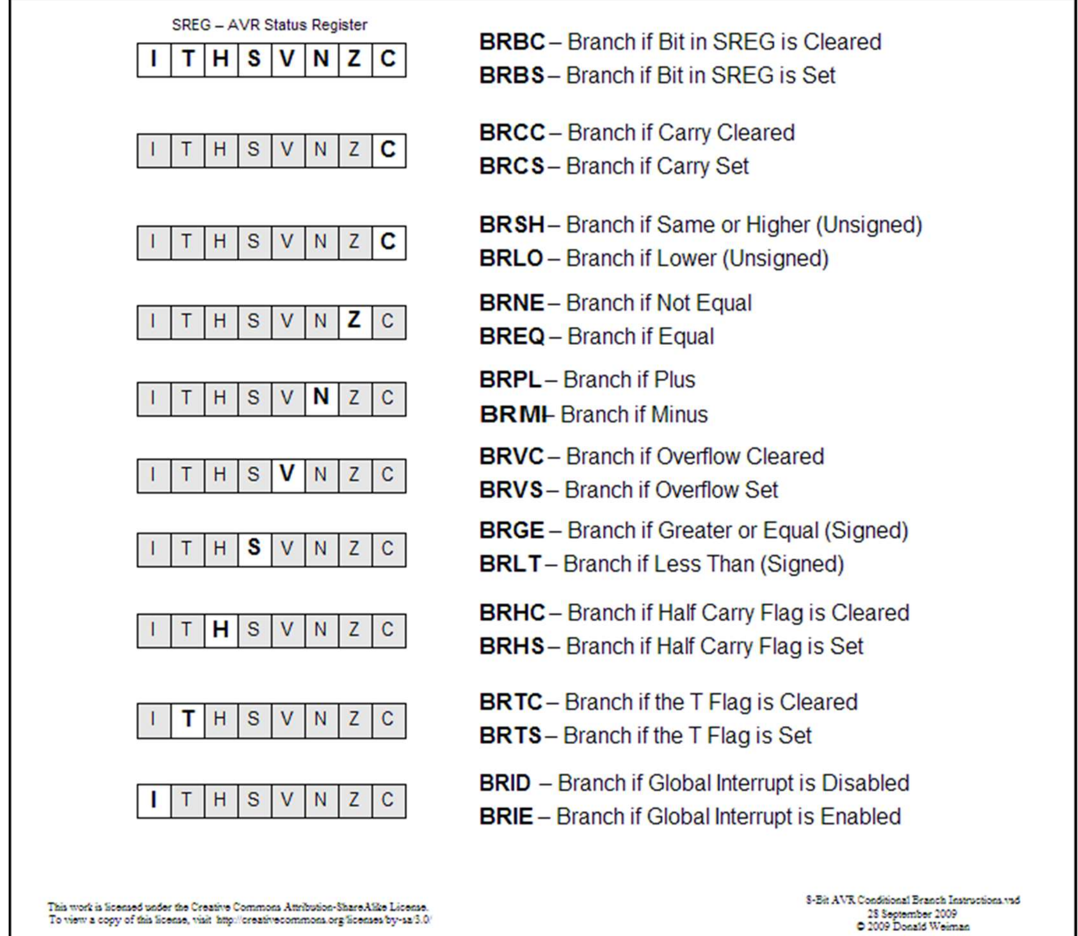
6.3.9 Skip: Conditional

```
sbic  A,b        ;skip next statement if bit in IO register is clear
sbis  A,b        ;skip next statement if bit in IO register is set
sbrc  Rd,b       ;skip next statement if bit in Rd is clear
sbrs  Rd,b       ;skip next statement if bit in Rd is set
```

6.3.10 Branch Instructions

Branch instructions are employed with the intent of immediately altering the contents of Program Counter (*that is, the address of the next instruction to be executed*) based on a flag, or combination of flags in the Status Register as a result of the previous executable statement. Unlike the `rcall` instruction that returns control to the *'what would have been the next instruction'* after the function is complete, branch instructions alter the **Program Counter** permanently.

BRANCH INSTRUCTIONS					
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1/2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1/2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1/2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1/2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1/2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1/2



6.4 Signed Representation of Numbers

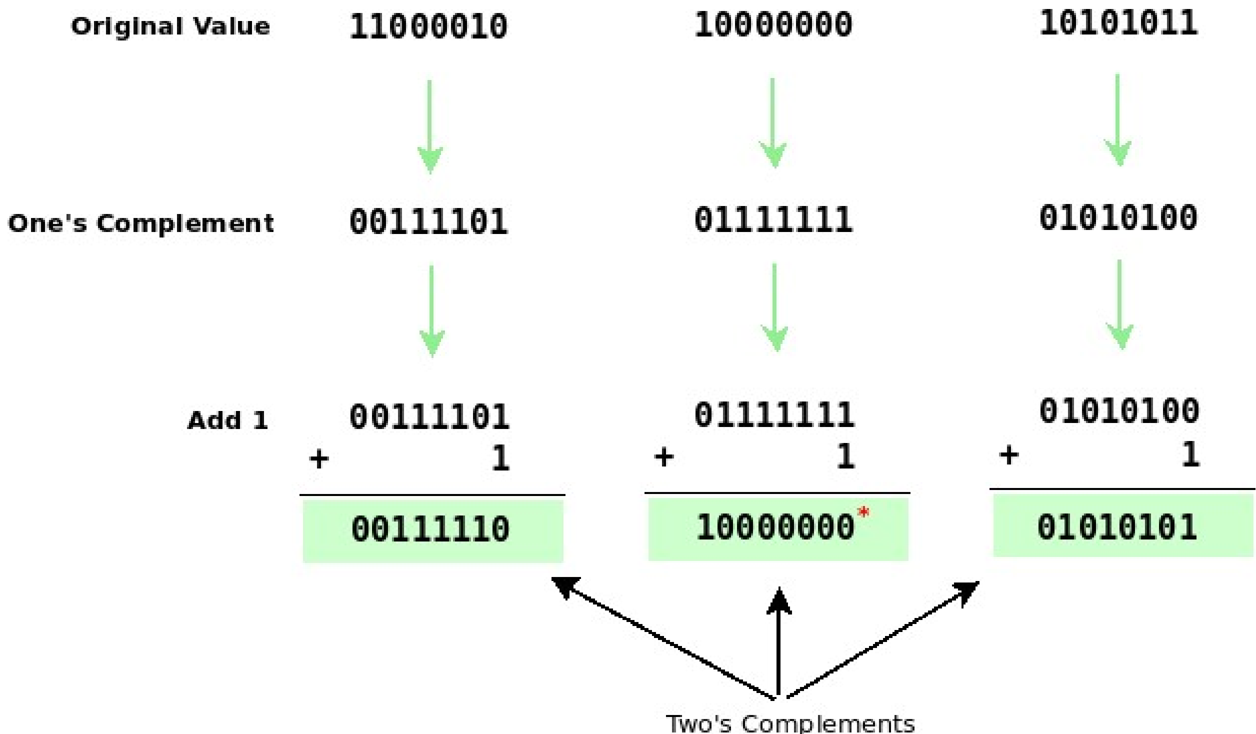
Due to the limitations of hardware it is simply not practical to maintain the normal arithmetic conventions of + and − signs as indicators of positive and negative numbers.

Even if that were possible, in designing a technique for the signed representation of numbers, computer engineers realized that whatever they came up with should place no additional burden on arithmetic operations for positive and negative operands.

Although a number of strategies have been employed for the signed representation of, binary numbers is *two's complement algorithm*.

6.4.0 Two's Complement

The accepted process of negation for binary numbers is referred to as the *two's complement* algorithm. This is a two-step process by which you form the *one's complement* first and then add 1. Three examples appear below. Explain each one in terms of it's decimal equivalent.



Note that the value 10000000 is its own *two's complement* value! What to do? Since the msb is 1 computer engineers decided that, to be consistent, it should be interpreted as a negative number, and the largest number in the range at that! This is why signed integer ranges are always asymmetric as in, $-\mathbf{128} \leq n < 127$, $-\mathbf{32768} \leq n < 32767$, etc.

6.5 Expressions

AVR expressions are constructed from numeric constants, operators, labels (addresses) and functions.

6.5.0 Operators

https://www.microchip.com/webdoc/index.html

<table>
<tr><th>Operator</th><th>Description</th><th>Precedence</th><th>Assoc</th><th>Example</th></tr>
<tr><td>!</td><td>Logical NOT</td><td>14</td><td>None</td><td><code>ldi r16,!0xf0 ;Load r16 with 0x00</code></td></tr>
<tr><td>~</td><td>Bitwise NOT</td><td>14</td><td>None</td><td><code>ldi r16,~0xf0 ;Load r16 with 0x0f</code></td></tr>
<tr><td>-</td><td>Unary Minus</td><td>14</td><td>None</td><td><code>ldi r16,-2 ;Load -2(0xfe) in r16</code></td></tr>
<tr><td>*</td><td>Multiplication</td><td>13</td><td>Left</td><td><code>ldi r30,label*2;Load r30 with label*2</code></td></tr>
<tr><td>/</td><td>Division</td><td>13</td><td>Left</td><td><code>ldi r30,label/2;Load r30 with label/2</code></td></tr>
<tr><td>%</td><td>Modulo</td><td>13</td><td>Left</td><td><code>ldi r30,label%2;Load r30 with label%2</code></td></tr>
<tr><td>+, -</td><td>Add, Sub</td><td>12</td><td>Left</td><td><code>ldi r17,c1-c2 ;Load r17 with c1-c2</code></td></tr>
<tr><td><<, >></td><td>Shift left, right</td><td>11</td><td>Left</td><td><code>ldi r17,c1>>c2 ;Load r17 with c1 shifted right c2 times</code></td></tr>
<tr><td><,<=,>,>=</td><td>Sign: 0 or 1</td><td>10</td><td>None</td><td><code>ori r18,bitmask*(c1<c2)+1 ;Or r18 with an expression</code></td></tr>
<tr><td>==, !=</td><td>Sign: 0 or 1</td><td>9</td><td>None</td><td><code>andi r19,bitmask*(c1==c2)+1 ;And r19 with an expression</code></td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>8</td><td>Left</td><td><code>ldi r18,high(c1&c2) ;Load r18 with an expression</code></td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>7</td><td>Left</td><td><code>ldi r18,low(c1^c2) ;Load r18 with an expression</code></td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>6</td><td>Left</td><td><code>ldi r18,low(c1|c2) ;Load r18 with an expression</code></td></tr>
<tr><td>&&</td><td>Logical AND</td><td>5</td><td>Left</td><td><code>ldi r18,low(c1&&c2) ;Load r18 with an expression</code></td></tr>
<tr><td>||</td><td>Logical OR</td><td>4</td><td>Left</td><td><code>ldi r18,low(c1||c2) ;Load r18 with an expression</code></td></tr>
<tr><td>? :</td><td>Ternary</td><td>3</td><td>None</td><td><code>ldi r18, a > b? a : b ;Load r18 with the maximum numeric value of a and b.</code></td></tr>
</table>

6.5.1 Expression Separation Functions

The AVR Assembler offers a number of built-in functions to facilitate your coding. A useful collection of functions allows you to separate bytes and words from larger expressions. Here is a list taken from the,

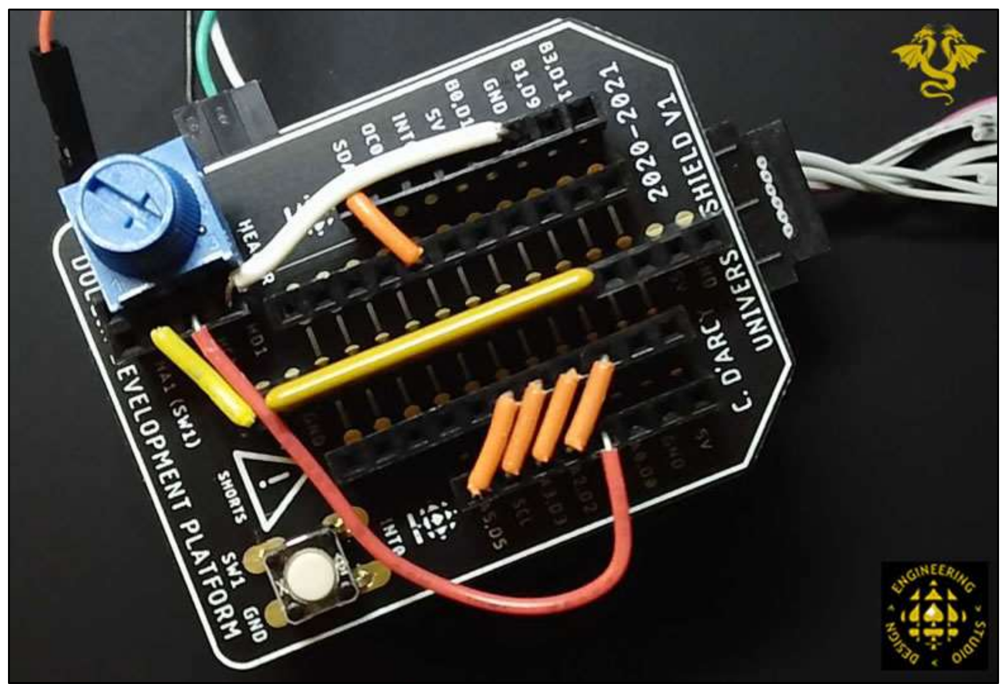
AVR Assembler's User's Guide (https://www.microchip.com/webdoc/)

Functions defined for the assembler.

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2 to the power of expression
- LOG2(expression) returns the integer part of log2(expression)
- INT(expression) Truncates a floating point expression to integer (i.e. discards fractional part)
- FRAC(expression) Extracts fractional part of a floating point expression (i.e. discards integer part).

The last four functions support your mathematics algorithms.

RSGC ACES Universal Shield V1: R2R Ladder as a DAC



6.6 Variables

Variables required an **identifier** (address) and **storage space** (bytes).

A **label** serves as the variable's identifier and bytes of storage can be set aside in any of the three memories (Program Flash, SRAM or EEPROM) through the use of **assembler directives**.

6.6.0 Variable Use in SRAM

This example demonstrates the use of byte-size variables in SRAM. Use of the `.DSEG` and `.BYTE` assembler directives ensure the storage for the `count` variable is located in SRAM. The assembly instructions `lds` and `sts` are the load and store instructions for SRAM addresses.

`rsgcaces > AVROptimization > 2_Small_Steps > VariablesSRAM.asm`

The screen capture below was taken at the end of a debug session with the Memory and Processor Windows (`Debug>Windows>etc.`) revealing the contents of their respective locations.

Notes.

- The SRAM free memory map begins at `0x0100` as expected and without the use of the `.org` directive this is default location of the where the value is loaded.
- An inline expression is used to calculate the target address of an `sts` instruction.
- The swap instruction interchanges the high and low nibbles of a byte.

```
1  /*
2   * VariablesSRAM.asm
3   * Created: 8/12/2018 3:32:09 PM
4   * Author: Chris D'Arcy
5   */
6  .DSEG
7  count:
8  .BYTE   2               ;reserve two bytes in SRAM
9  .def    util=r16        ;provide an alias for r16
10 .CSEG
11 .org    0x0000
12     rjmp    reset
13 .org    0x0100
14 reset:
15     ldi     util,0xAB   ;prepare a sample value for st
16     sts     count,util  ;an 'assignemnt' statement
17     lds     r17,count   ;retrieve the value from SRAM
18     swap    r17         ;perform some verifiable operation
19     sts     count+1,r17 ;store the modified byte value
20     rjmp    reset
21
```

```
Memory 1
Memory: data IRAM
data 0x0100  ab ba 00 00 00 00 00 00 00  «º.......
data 0x0109  00
data 0x0112  00
data 0x011B  00
data 0x0124  00
data 0x012D  00
data 0x0136  00
data 0x013F  00
data 0x0148  00
data 0x0151  00
data 0x015A  00
data 0x0163  00
```

Name	Value
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xAB
R17	0xBA
R18	0x00
R19	0x00
R20	0x00
R21	0x00

6.6.1 Variable Use in Program Flash and EEPROM

The ATmega328P offers just under 2 KB of SRAM. This space has to accommodate variables as well as the Stack which grows from the end of SRAM (0x8FF), upwards.

If SRAM space for your variables gets tight, you can consider using Program Flash or even EEPROM as a source of additional storage. The `.DB`, `.DW`, `.DD`, and `.DQ` assembler directives are used to reserve space in Program Flash or EEPROM.

6.6.1.0 .DB

(*From online help*)Define constant byte(s) in program memory and EEPROM. The DB directive reserves memory resources in the program memory or the EEPROM memory. In order to be able to refer to the reserved locations, the DB directive should be preceded by a label. The DB directive takes a list of expressions, and must contain at least one expression. The DB directive must be placed in a Code Segment or an EEPROM Segment.

The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -128 and 255. If the expression evaluates to a negative number, the 8-bits twos complement of the number will be placed in the program memory or EEPROM memory location.

If the DB directive is given in a Code Segment and the expression-list contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. *If the expression-list contains an odd number of expressions, the last expression will be placed in a program memory word of its own, even if the next line in the assembly code contains a DB directive.* The unused half of the program word is set to zero. A warning is given, in order to notify the user that an extra zero byte is added to the .DB statement.

6.6.1.1 .DW

Define constant word(s) in program memory and EEPROM.

The DW directive reserves memory resources in the program memory or the EEPROM. In order to be able to refer to the reserved locations, the DW directive should be preceded by a label. The DW directive takes a list of expressions, and must contain at least one expression. The DB directive must be placed in a Code Segment or an EEPROM Segment.

The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -32768 and 65535. If the expression evaluates to a negative number, the 16-bits two's complement of the number will be placed in the program memory or EEPROM location.

6.6.1.2 .DD

Define constant double-word(s) in program memory and EEPROM.

This directive is very similar to the DW directive, except it is used to define 32-bit (double-word). The data layout in memory is strictly little-endian.

6.6.1.3 .DQ

Define constant quad-word(s) in program memory and EEPROM.

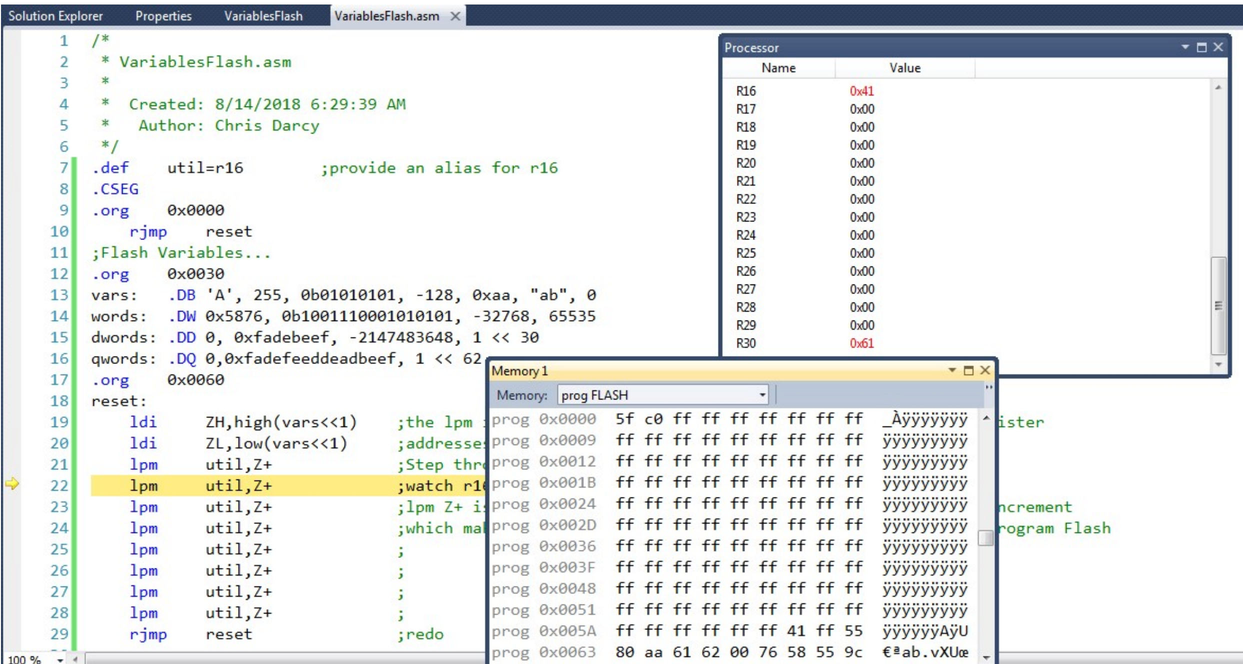
This directive is very similar to the DW directive, except it is used to define 64-bit (quad-word). The data layout in memory is strictly little-endian.

6.6.1.4 Example: Variable in Flash

The example below makes use of the four assembler directives that both reserve storage space in Program Flash and initialize the byte contents at the same time.

`rsgcaces > AVROptimization > 2_Small_Steps > VariablesFlash.asm`

Once the `VariablesFlash` project is created, select the Simulator Tool and use the debugger to step through the code. Be sure to have the Memory: Program Flash and Processor windows open (**Debug>Window>...**)



6.6.1.4.0 Questions

1. What is meant by *little* and *big* endian?

2. What is the AVR byte order (little or big endian)?

2. The first executable instruction, `rjmp reset`, appears as `5f c0`. Interpret these contents.

3. Identify the byte address of `vars` ________

4. What is the most efficient way express the largest possible positive value for,

   a) a double word (`.DD`)

   b) a quad word (`.DQ`)

5. Explain Lines 19 and 20.

6. How is the decimal value `−32768` stored? Explain this.

6.7 Arrays

An array declaration reserves a contiguous block of storage under a single identifier that is used to identify the base address of the storage block. Initialization of the elements is optional.

The requirement that the data in an array to be homogenous enables the index of each cell to be used to determine the address of the element as an offset from the base address.

6.7.0 C Array Example

Let's start with familiar high-level C code that declares and initializes an integer array, before proceeding to total the contents. A complete debugging session leaves SRAM in the state shown.

`rsgcaces > AVROptimization > 2_Small_Steps > CArrayExample1.c`

Disassembly Solution Explorer Properties ArrayExample1 ArrayExample1.c

ArrayExample1.c C:\Users\Chris Darcy\Documents\MyAtmelStudio\ArrayExample1\ArrayExample1\ArrayExample1.c

```
1  /*
2   * ArrayExample1.c
3   *
4   * Created: 8/17/2018 8:16:22 AM
5   *  Author: Chris Darcy
6   */
7
8  #include <avr/io.h>
9  uint8_t A [] = {16,2,77,40,107};
10 uint8_t B [] = {0x20,0x30};
11 uint8_t i, sum = 0;
12 int main(void)
13 {
14     for(i=0; i<5; i++)
15         sum+=A[i];
16 }
```

Memory 2 — Memory: data IRAM

```
data 0x0100  10 02 4d 28 6b 00 f2 05 00  ..M(k.ò..
data 0x0109  00 00 00 00 00 00 00 00 00  .........
data 0x0112  00 00 00 00 00 00 00 00 00  .........
data 0x011B  00 00 00 00 00 00 00 00 00  .........
data 0x0124  00 00 00 00 00 00 00 00 00  .........
data 0x012D  00 00 00 00 00 00 00 00 00  .........
data 0x0136  00 00 00 00 00 00 00 00 00  .........
data 0x013F  00 00 00 00 00 00 00 00 00  .........
data 0x0148  00 00 00 00 00 00 00 00 00  .........
data 0x0151  00 00 00 00 00 00 00 00 00  .........
data 0x015A  00 00 00 00 00 00 00 00 00  .........
```

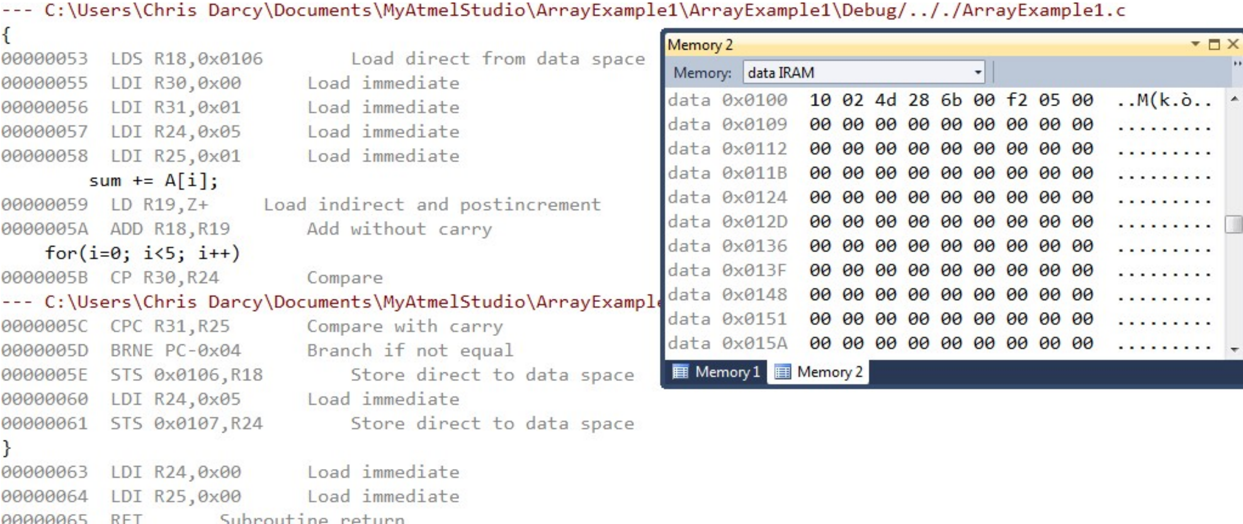
Memory 1 Memory 2

6.7.0.0 Comments and Observations from C Array Example

Some notable aspects from the graphic above include the following,

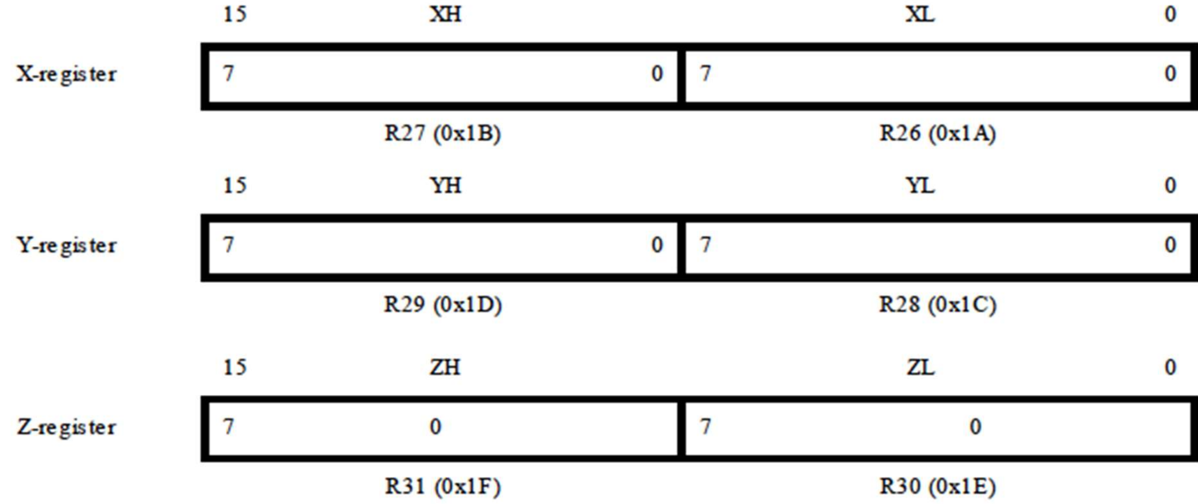
1. **Line 9:** The storage for the byte array **A** is located within SRAM and starts at **@0x0100**, immediately following the bank of 256 registers (32 GP, 64 IO and 160 Extended)
2. **Line 9**: The byte order of the array matches the initialization order
3. **Line 9**: The assembler maintains storage allocation to **even** byte boundaries. Since an odd number of elements were defined, it pads the storage with an extra byte (**@0x0105**)
4. **Line 10:** Although the byte array B is declared and initialized, since the assembler recognizes that it is never referenced, no storage is set aside for its use.
5. **Line 11**: The assembler appears to use a JIT (Just in Time) storage allocation strategy. Although the variable `i` is declared *before* the variable sum, only the latter's value is known at this time, so it is given the next available free address, @0x0106. Stepping through the code, you'll notice the updating of SRAM is suspended until the loop is finished.

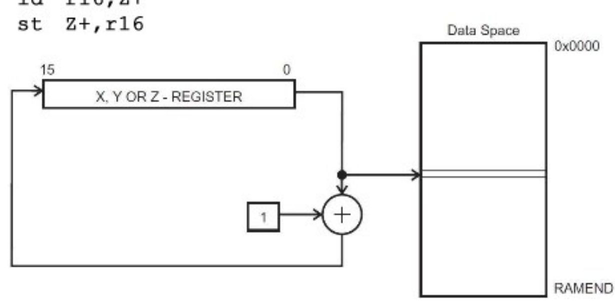
6. **Line 16**: Immediately following the end of the `main` method, the disassembled version appears.



6.7.1 Data Indirect Addressing Modes

Beyond their use as general purpose registers, the two-byte combinations R27:R26, R29:R28, and R31:R30 are given aliases (X, Y, and Z, respectively), for the purpose of facilitating the addressing of memory. These registers serve as 16-bit address pointers for indirect addressing of SRAM.





Look back at the disassembled version of the C array example at the top of this page, in particular Lines 55 and 56. Explain what this is doing.

Explain lines 57 and 58.

Explain the body of the loop: lines 59 to 5D.

Explain the exit of the loop: lines 5E to 61.

6.7.2 Assembly Example

With a working knowledge of the high-level C array example above, we take on the assembly equivalent. In doing so, we optimize where we can.

rsgcaces > AVROptimization > 2_Small_Steps > ArrayExample1.asm

```
/*
 * ArrayExample1.asm
 *  Created: 8/18/2018 8:21:17 AM
 *   Author: Chris Darcy
 */
 .DSEG                          ;
 .BYTE      6                   ;reserve an even number of bytes in SRAM
 .CSEG
.org  0x0000
      rjmp  copy
.org  0x0030
A:    .DB   16,2,77,40,107,0   ;define and initialize the array, A
Aend:
.org  0x0100
copy:                           ;.BYTE does not permit initialization
      ldi   ZL,low(A<<1)       ;we frst copy from program flash to SRAM
      ldi   ZH,high(A<<1)      ;lpm instruction requires Z register
      ldi   YL,low(Aend<<1)    ;point Y to the end of the array
      ldi   YH,high(Aend<<1)   ;
      clr   XL                  ;point X to the start of SRAM
      ldi   XH,0x01             ;
      lpm   r0,Z+               ;Load from program memory and postincrement
      st    X+,r0               ;Store indirect and postincrement
      cp    ZL,YL               ;end of the array? Compare low bytes
      brne  PC-0x03             ;branch if not equal
                                ;leave X at first address after array
      clr   r18                 ;zero a register for sum prior to accumulation
      clr   ZL                  ;point Z to the beginning of the array
      ldi   ZH,0x01             ;SRAM address: 0x0100

      ld    r19,Z+              ;get the (next) element of the array
      add   r18,r19             ;add it to the running sum: sum += A[i];
      cp    ZL,XL               ;end of the array? Compare low bytes
      brne  PC-0x03             ;branch if not at end
      st    Z,r18               ;store sum in SRAM
wait: rjmp  wait                ;done...
```

2.7.2.0 Comments and Observations from Assembly Array Example

1. Comparing this assembly version with the disassembled version of the C code, identify as many improvements, efficiencies, or optimizations that you can.

2.

6.8 If…then…else

For practice using branch instructions, consider coding an `if...then…else` structure in assembly. Specifically, place an RGB LED in pins 10 through 13 of you Arduino. Obtain a value for `temp` and display the red LED if it's greater than 25°, and the blue LED if it's less than 15°.

rsgcaces > AVROptimization > 2_Small_Steps > IfThenElse.asm

```
/*
 * IfThenElse.asm
 *
 *  Created: 8/8/2018 3:19:27 PM
 *   Author: Chris Darcy
 */
.def  util  = r16               ;
.def  led   = r17               ;
.equ  COOL  = 15                ;
.equ  WARM  = 25                ;
.equ  temp  = 10                ;
.equ  red   = 1<<PB2            ;
.equ  gnd   = 1<<PB3            ;
.equ  blue  = 1<<PB5            ;
.org  0x0000
      rjmp  reset               ;
.org  0x0100
reset:
      rcall initPORT            ;
again:
      rcall getTemp             ;
      cpi   r16,COOL            ;
      brmi  sayCool             ;
      cpi   r16,WARM            ;
      brpl  sayWARM             ;
      rjmp  again               ;
sayCool:
      ldi   led,blue            ;
      out   PORTB,led           ;
      rjmp  again               ;
sayWarm:
      ldi   led,red             ;
      out   PORTB,led           ;
      rjmp  again               ;

initPORT:
      ldi   util,red|gnd|blue   ;
      out   DDRB,util           ;
      ret
getTemp:
      ldi   r16,temp            ;
      ret
```

6.9 Loop

A **loop** is a structure in which a block of statements is repeated until an **event** occurs. In high-level languages the **event** is coded as a boolean expression (aka. condition).

If the number of repetitions (aka iterations) is not known in advance, the convention is to code the structure using the **while** keyword.

If the number iterations **is known in advance**, the convention is to code the structure using the **for** keyword.

Assembly languages do not have data types, per se, so a boolean expression is reduced to an interpretation of the state of one or more flags of the Status Register.

6.9.0 for Loop

Here's an example of how you might code a **for** loop that iterates from 9 to 0 inclusive, mimicking the C statement, `for(uint8_t i=0; i<10, i++).`

rsgcaces > AVROptimization > 2_Small_Steps > folLoop.asm

```
/*
 * forLoop.asm
 * Performs 10 iterations (5 cycles) of Blinking LED on pin 13
 * Author: Chris Darcy
 */
.equ   START  = 0               ;lower bound of for loop
.equ   END    = 10              ;upper (exclusive) bound of for loop
.def   index  = r18             ;index of the for loop (lcv)
.equ   PIN13  = 1<<PB5          ;visual confirmation of iteration
.def   util   = r16             ;generic utility register
.def   led    = r17             ;led register for toggling purposes

.org   0x0000
       rjmp   setup             ;let's use the Arduino C terminology
.org   0x0100                   ;well past the interrupt jump vector table
setup:
       ldi    led,PIN13         ;one-time code
       out    DDRB,led          ;set PORTB bit 5 for output (pin 13)
loop:
       clr    util
       out    PORTB,util        ;start with LED on pin 13 OFF
       ldi    index,START       ;initialize loop control variable
forLoop:
       cpi    index,END         ;are we finished?
       breq   exit              ;if so, exit the for loop
       eor    util,led          ;body of the for loop: toggle state of pin 13
       out    PORTB,util        ;display it
       rcall  delay1s           ;admire
       inc    index             ;advance the loop control variable
       rjmp   forLoop           ;back to the top of the for loop
exit:
       rcall  delay1s           ;admire
       rcall  delay1s           ;admire
       rcall  delay1s           ;admire
       rjmp   loop
delay1s:                        ;included in download
```

7 AALP: Arithmetic and Mathematics

The table of Arithmetic and Logic Instructions below is taken from Atmel's AVR 8-bit Instruction Set Manual.

http://mail.rsgc.on.ca/~cdarcy/Datasheets/InstructionSetSummary.pdf

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← 0xFF – Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← 0x00 – Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← 0xFF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C	2
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2

7.0 Terminology: Overflow and Underflow

Like the odometer on your car, when you go past the maximum value your display can represent the count simply rolls over and the counting starts again at 0. In computing, when an arithmetic operation on an MCU results in a value too large for the target register to contain we refer to this as an **overflow** condition.

The interpretation of the term **underflow** depends on the context. For **fixed point** numbers (integers), such as the 8-bit registers we are using, an underflow condition is said to occur when the value would be less than the minimum value an integer (register) can hold (0). For **floating point** numbers, an underflow condition occurs when the result of an arithmetic operation results in a value too close to zero to distinguish it from the same.

7.1 Adding or Subtracting One from a Register

Incrementing and decrementing a register, the hallmark of counting and loop control, is best accomplished through the dedicated instructions `inc` and `dec`. Each instructions requires only a single register from `r0` through `r31`. Overflow and underflow conditions will generate SREG flag responses that can be monitored with branch instructions.

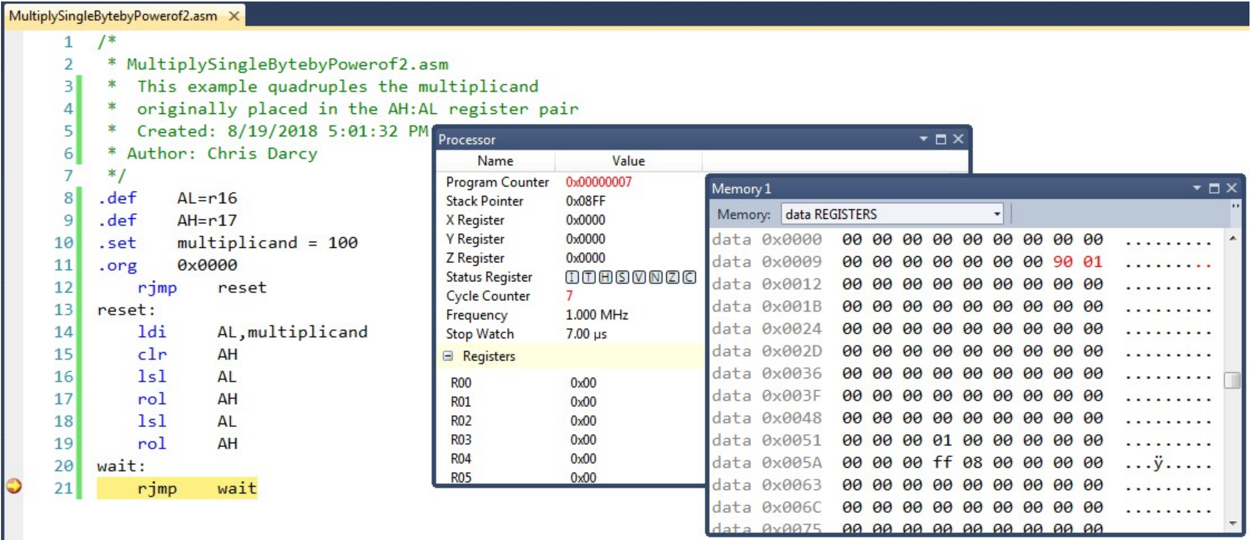
7.2 Multiplying and Dividing a Single Byte by a Power of 2

Just as shifting the digits to the left or right of a decimal number has the effect of multiplying or dividing by a power of 10 so, too, does shifting bits in a binary number have the effect of doing the same for powers of 2. Furthermore, hardware circuits are embedded within the processor's **hardware** to expedite the process. Not surprisingly then, the following instructions figure prominently in low-level multiplication and division routines.

Mnemonics	Operands	Description	Operation	Flags	#Clocks
BIT AND BIT-TEST INSTRUCTIONS					
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0...6	Z,C,N,V	1

7.2.0 Multiplying a Single Byte by a Power of 2

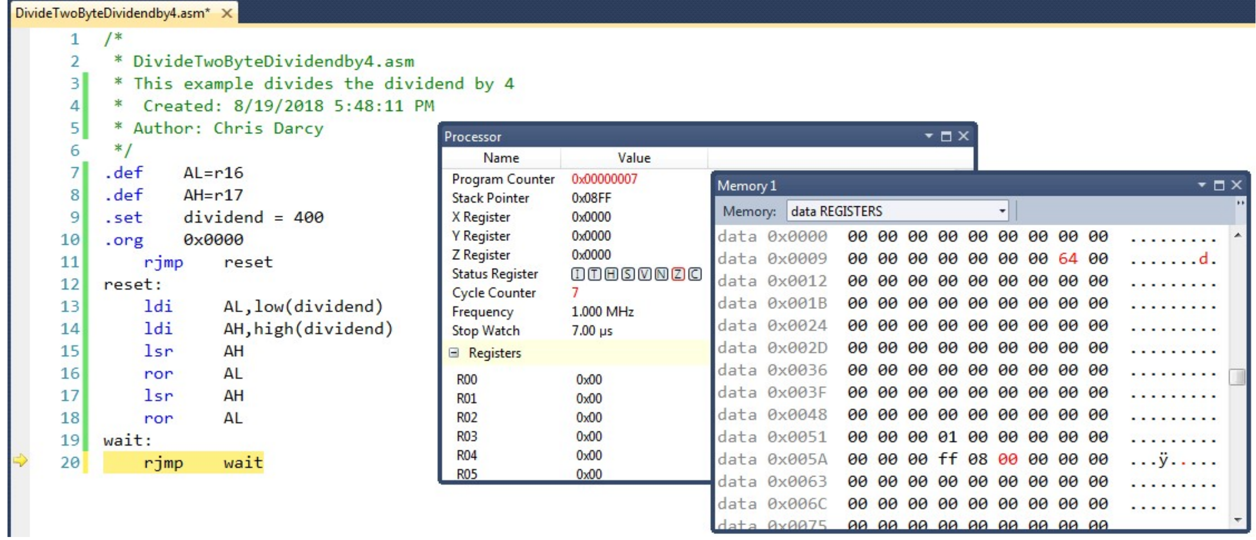
In this example, the intent is to multiple a single byte by **four**. Recognizing this operation could result in two-byte, 16-bit outcome, we designate a registers as the high byte of the eventual product and clear it at the outset. The `lsl` (logical shift left) instruction is used on the lower byte because it will shift the contents one bit to the left, a zero is shifted into the lowest bit and the highest bit is shifted into the carry flag. We immediately employ a `rol` (rotate left though Carry) instruction command on the high byte because it will also shift contents one bit to the left, but it will shift the contents of the Carry Flag into the lowest bit. Every time we shift the multiplicand to the left we are multiplying by two. So, to multiply by four we simply repeat this pair of instruction.



7.2.1 Dividing Two-Byte (Word) Dividend by a Power of 2

The AVR Instruction Set does not contain a divide instruction. This must be accomplished, manually. Later on we'll tackle general divisors but, for now, we'll restrict ourselves to dividing by powers of 2. As with multiplication, division of binary numbers by powers of 2 can be accomplished by shifting bits to the right. To make things more interesting, we'll start with the 16-bit product of our previous example (400) as our initial dividend. It should be apparent that we are simply undoing the multiplication steps.

The `lsr` (logical shift right) instruction is applied to the high byte because it will shift the contents one bit to the right, a zero is shifted into the highest bit and the lowest bit is shifted into the Carry Flag. We then employ the `ror` (rotate right though Carry) command on the low byte because it will also shift contents one bit to the right, but it will shift the contents of the Carry Flag into the highest bit. Every time we shift the dividend to the right we are dividing it by two. So, to divide by four, we simply shift the entire dividend to the right two times.



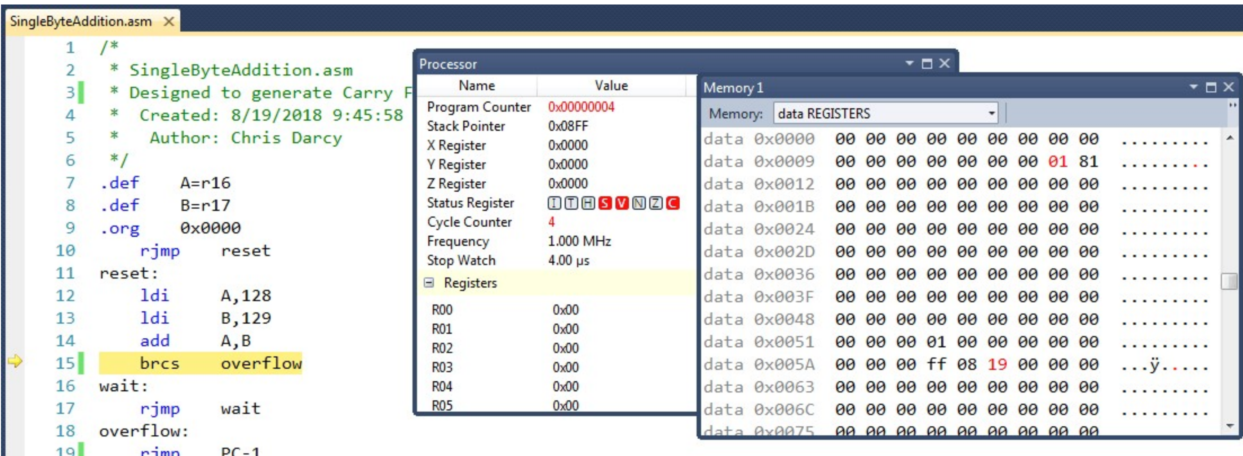
7.3 Byte Arithmetic

Care must be taken when employing arithmetic operations involving single byte registers to appreciate **overflow** and **underflow** situations. When either condition is triggered, the **C** flag within the SREG is set to allow you to recognize and respond to it in some manner.

7.3.0 Byte Addition with Overflow (Carry Flag)

This example serves to demonstrate an **overflow** condition triggered by the addition of two registers in which the sum exceeded 255. The `BRCS` (Branch if Carry Set) instruction MUST immediately follow the instruction that generated the condition. Note that the lower order 8 bits of the sum (in A) remains accurate. Create the project, obtain the course code, and step through a debugging session to experience it for yourself.

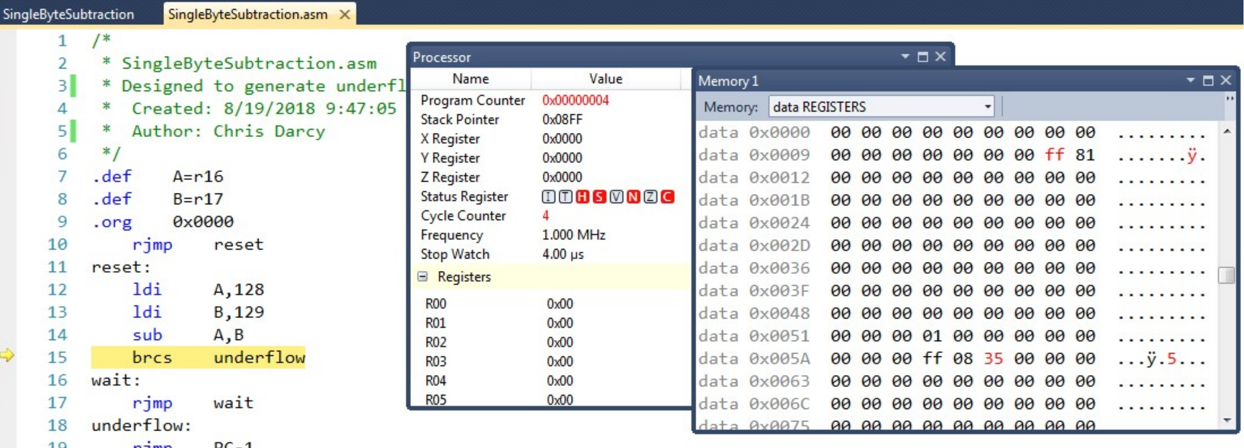
`rsgcaces > AVROptimization > 2_Small_Steps > SingleByteAddition.asm`



7.3.1 Byte Subtraction with Underflow (Carry Flag)

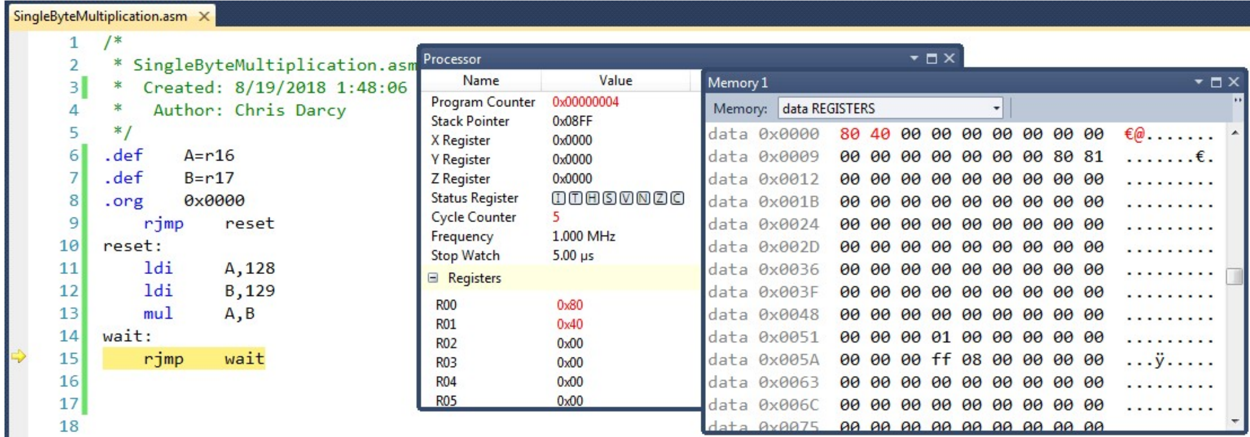
The difference between the two registers yields a value less than (0) triggering an underflow.

`rsgcaces>AVROptimization> 2_Small_Steps > SingleByteSubtraction.asm`



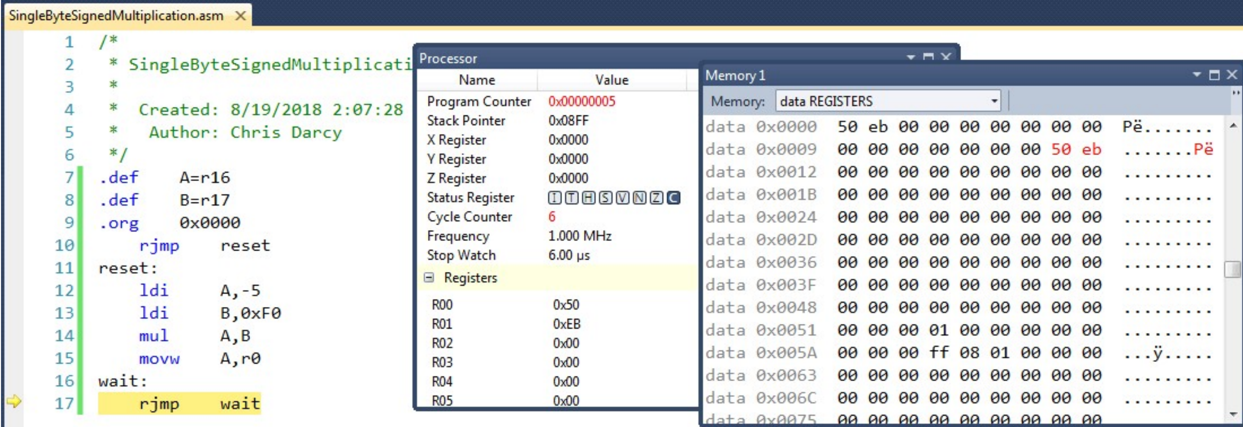
7.3.2 Unsigned Byte Multiplication with the MUL Instruction

The ATmega328P supports the MUL (Multiply Unsigned) Instruction. Any two registers (r0-r31) can be used as operands and are left unaffected as the product is placed in the register pair r1:r0. The instruction takes 2 clock cycles.



7.3.3 Signed Byte Multiplication with the MULS Instruction

In this example, the product of two negative operands yields a positive product that replaces the source operands,



7.3.4 Byte Division

See Algorithms: Byte Division

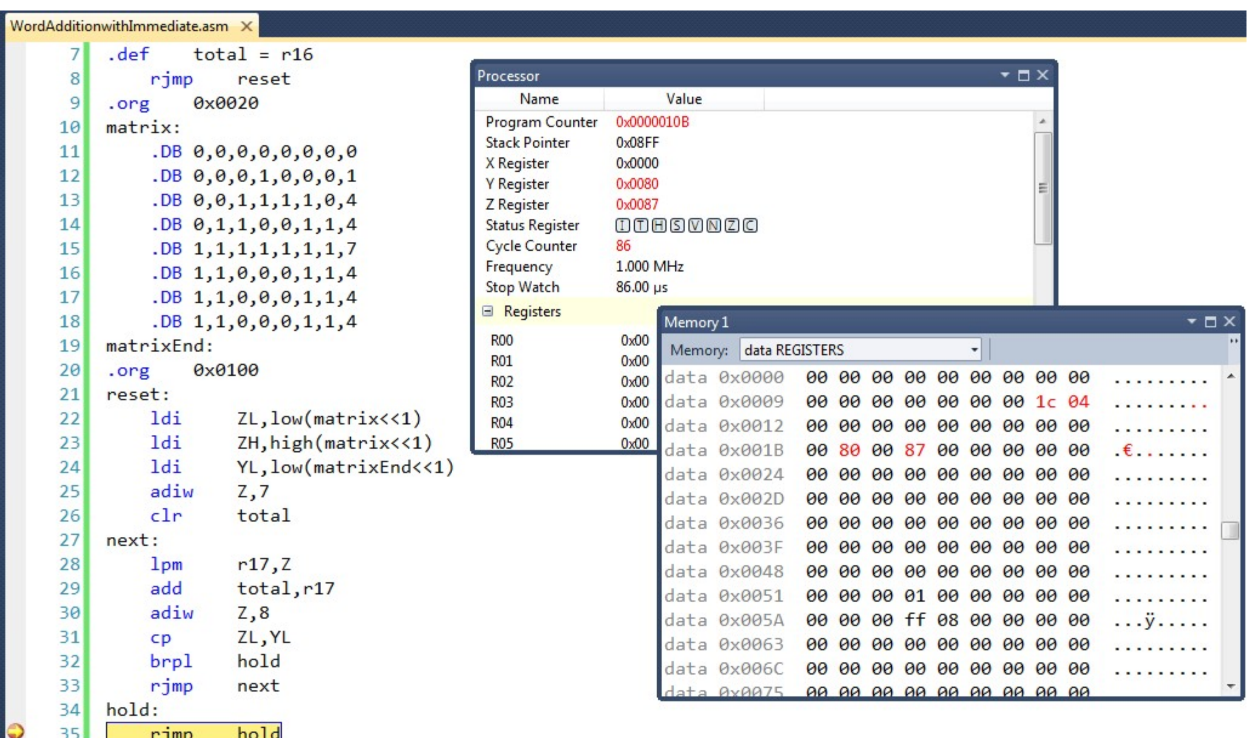
7.4 Arithmetic with Multi-Byte Operands

First, there are two dedicated word instructions for addition and subtraction that should be noted.

7.4.0 Two Dedicated Word Instructions: ADIW and SBIW

Two specialized arithmetic instructions are offered, primarily for the purpose of purpose of manipulating pointers (indices into arrays). Both `adiw` (add immediate to word) and `sbiw` (subtract immediate from word) apply a constant in the interval [0,63] to a register pair, r25:r24, r27:r26, r29:r28, or r31:r30.

In the following example, an 8x8 LED matrix image defining the letter 'A' is placed into program flash. The final column in each row of the matrix contains the number of set bits in the row. The assembly code below runs through the rows tallying the total number of set bits and placed the sum in r16 (total).



Lines 25 and 30 make use of the `adiw` instruction to point to the final column of each row.

Debugging Note. While in a debugging session, clicking in the leftmost gray column sets a breakpoint. Clicking again removes it. With a breakpoint set, you can select **Run to Cursor** from the `Debug > Window Menu` to see the net effect of executing the instructions in between.

3.4.1 Preparing Multi-Byte Operands

If we wish to perform arithmetic operations on integers greater than 255 special preparation must be undertaken to separate multi-byte operands into respective byte registers.

3.4.1.0 Applicable Byte Functions

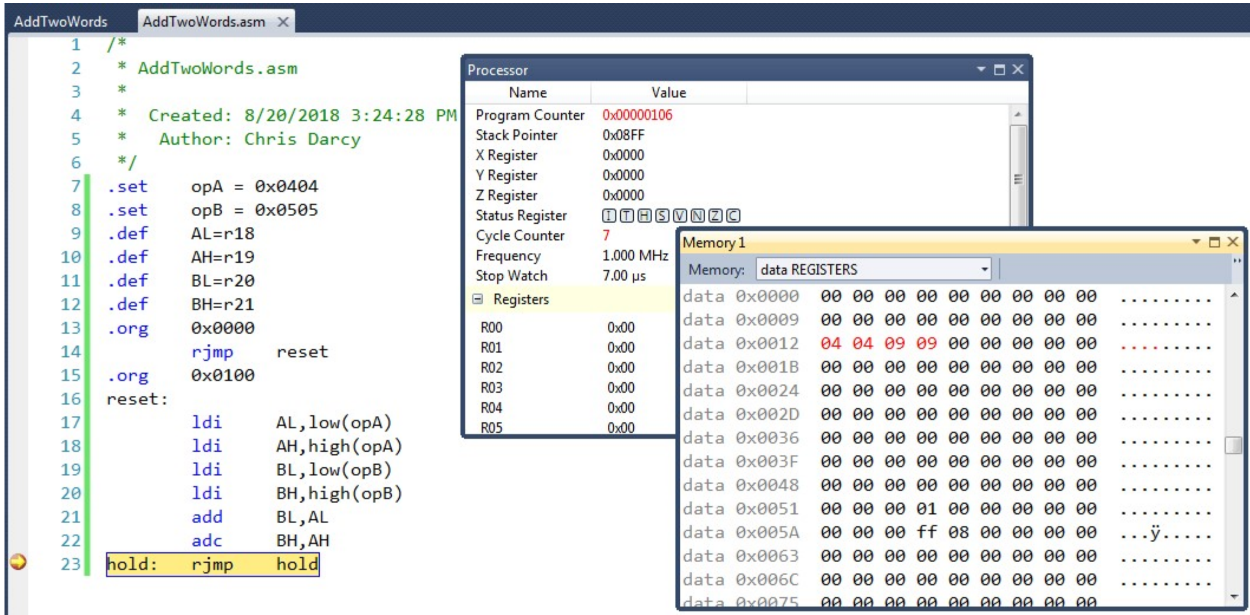
The AVR Assembler recognizes the following set of convenient functions that return bytes separated from words and double words

- `low(expression)` returns the low byte of an expression
- `high(expression)` returns the high byte of an expression
- `byte2(expression)` is the same as high
- `byte3(expression)` returns the third byte of an expression
- `byte4(expression)` returns the fourth byte of an expression

These functions are to be employed to separate operands into respective registers prior to perform arithmetic operations.

3.4.2 Adding Two Words

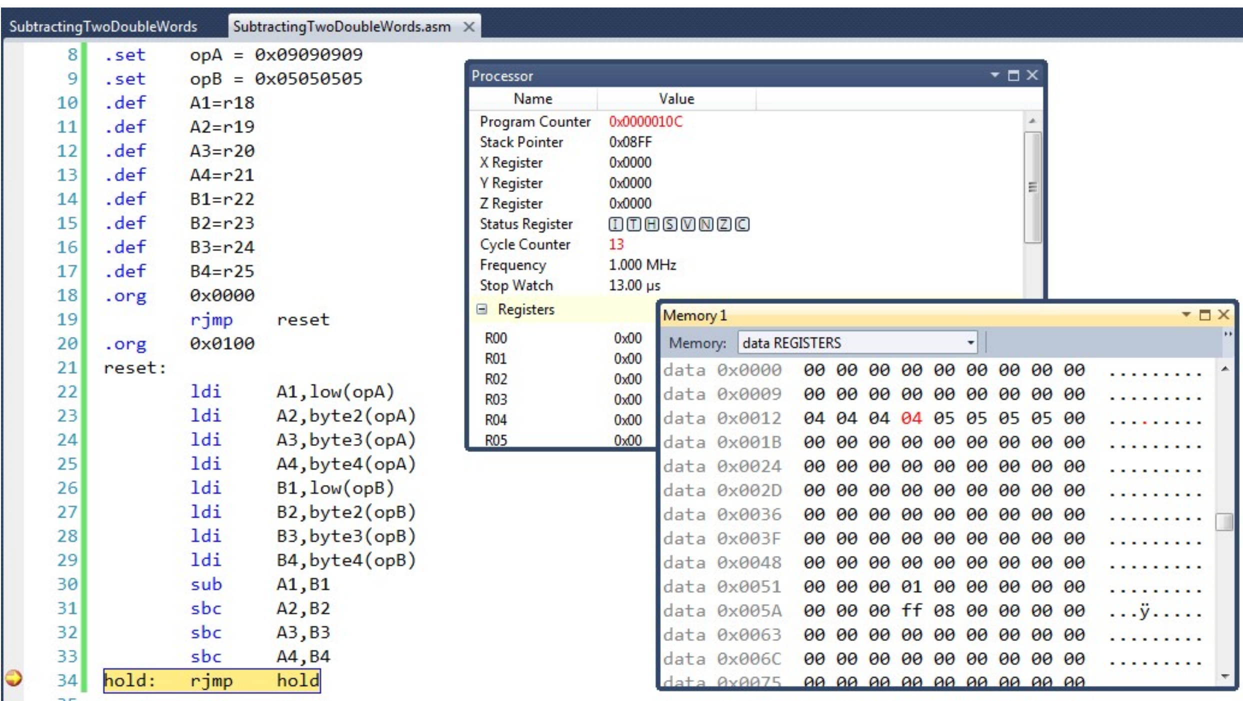
In this example, two 16-bit constants are defined as source operands (Lines 7 and 8), before separating those into two register pairs, `A` and `B` (Lines 17-20). The intent is to implement the assignment statement, `B = A+B`.



The Memory view confirms that after adding the low bytes of the operands with `add` instruction, followed by the addition of the high bytes of the operands with the `adc` instruction, the sum is correct (0x0909).

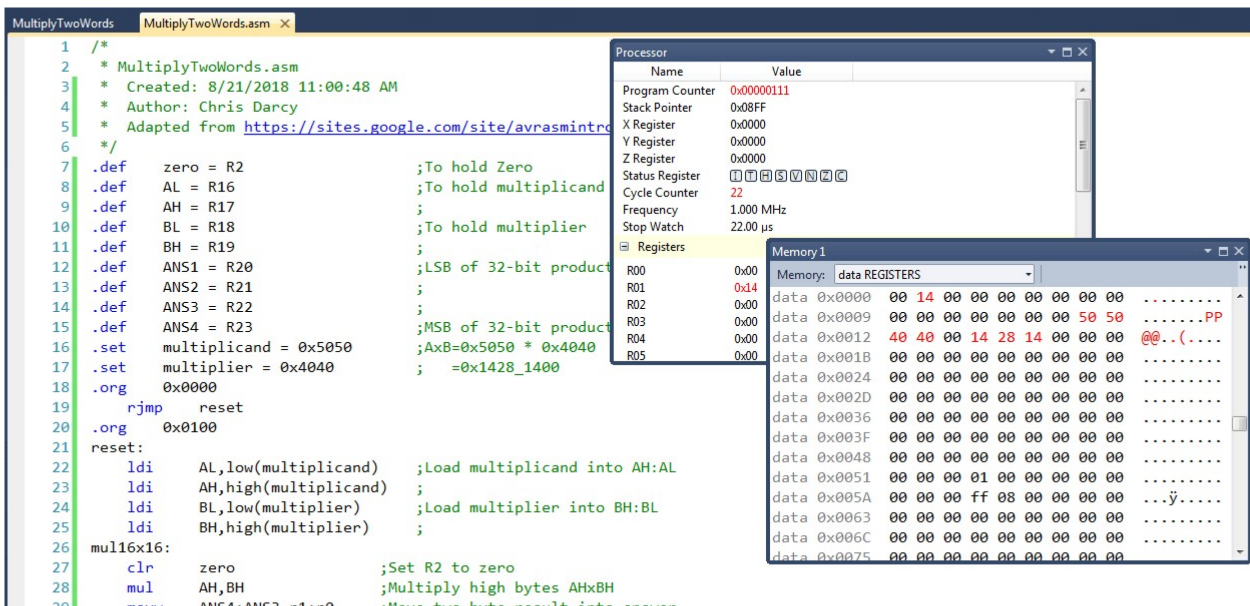
3.4.3 Subtracting Two Double Words

In this somewhat extreme example, the difference between two double-word (4-byte) operands is determined. Each of the operands `opA` and `opB` have their bytes separated into respective registers prior to implementing the equivalent of the assignment statement, `A=A-B`.



Lines 22 through 29 separate the double word. Starting with the lower byte pairs, perform the first subtraction with the `sub` instruction. If the result is negative, the Carry flag is set. This explains why the subsequent subtractions of the remaining byte pairs must be undertaken with the support of the `sbc` instruction.

3.4.4 Multiplying two Words with the MUL Instruction



8 AALP: AVR Assembly Language Programming within the Arduino IDE

8.0 Inline Assembly

Within the Arduino IDE, there are a number of ways to embed assembly code within your Arduino C code. The AVR Inline Assembly Cookbook, dating from 2002, describes a highly cryptic technique that is far too cumbersome for my taste, but you may find it more to your liking:

http://www.nongnu.org/avr-libc/user-manual/inline_asm.html

Hats off to this guy who presents a tutorial making it more palatable:

https://ucexperiment.wordpress.com/2016/03/04/arduino-inline-assembly-tutorial-1/

The use of Special Function Register (SFR) macros allows one to access the registers by name rather than their memory-mapped addresses.

8.0.0 Blink

The technique below is perhaps the simplest.

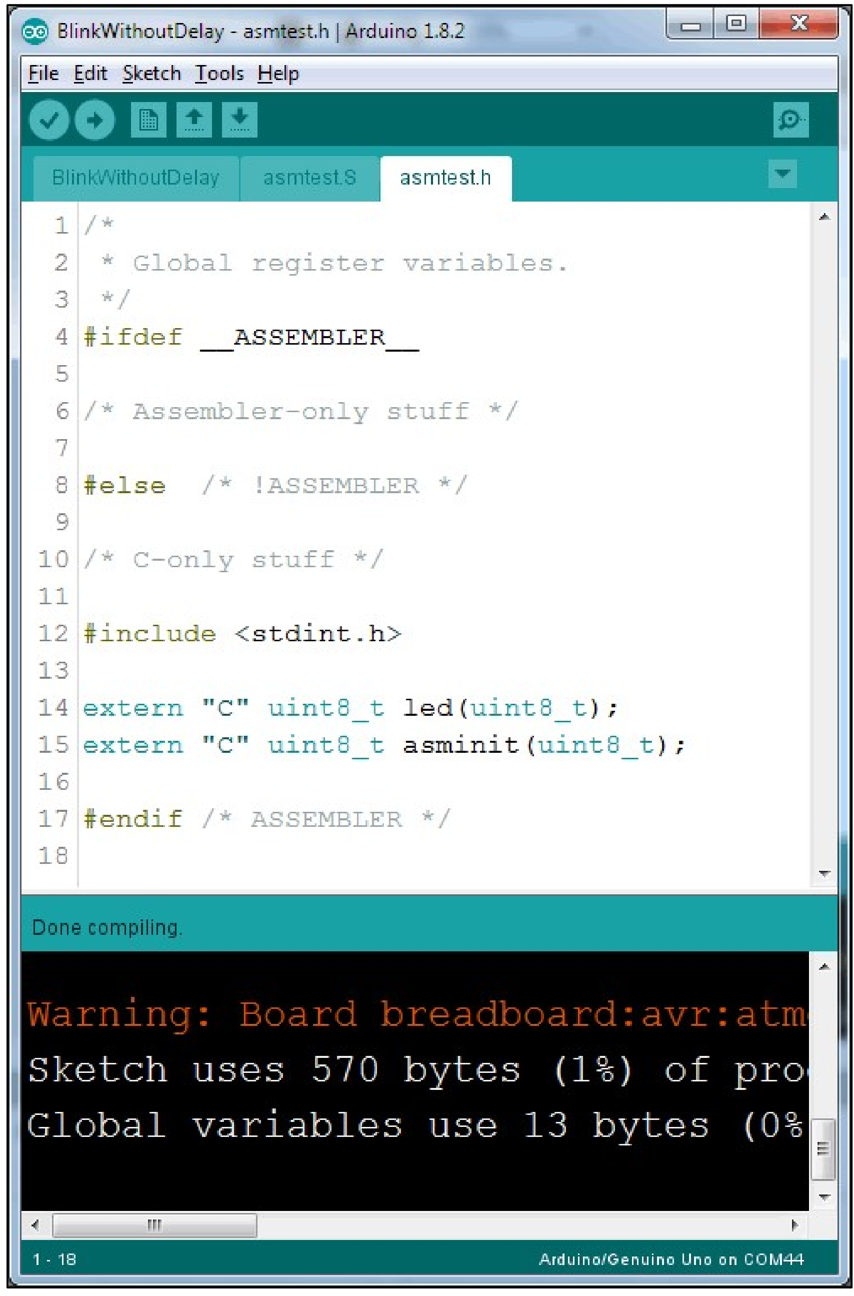
```
1  // Purpose   : Demonstrates the use of inline assembly to Blink pin 13
2  // Author    : C. D'Arcy
3  // Date      : 2017 10 13
4  // Status    : Working
5  void setup() {
6    asm(
7      "ldi r16,0b00100000  \n"    //prepare pin 13 (PB5) for output
8      "sts 0x24,r16        \n"    //do it
9      "ldi r16,0b00100000  \n"    //constant for setting pin 13 high
10     "sts 0x25,r16        \n"    //do it
11   );
12 }
13 void loop() {
14   delay(1000);                  //pause
15   asm(
16     "com  r16            \n"    //invert previous value of r16
17     "sts  0x25,r16       \n"    //toggle PORTB
18     );
19    }
```

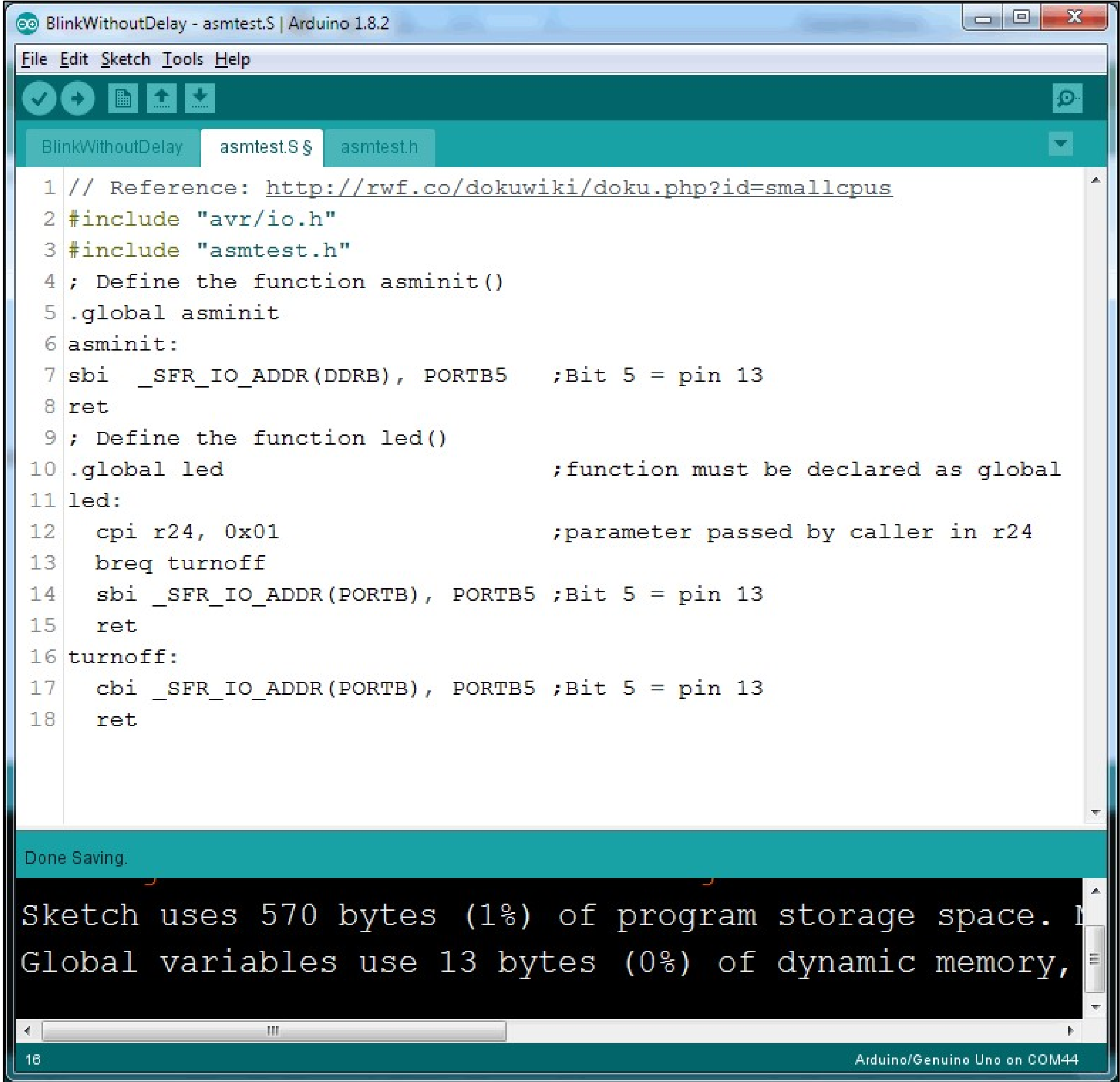
8.0.1 Blink Without Delay

This tutorial documents one user's attempts to pursue inline assembly within the Arduino IDE:

http://rwf.co/dokuwiki/doku.php?id=smallcpus

The two files below are used in conjunction with the driver from Section 2.1.1



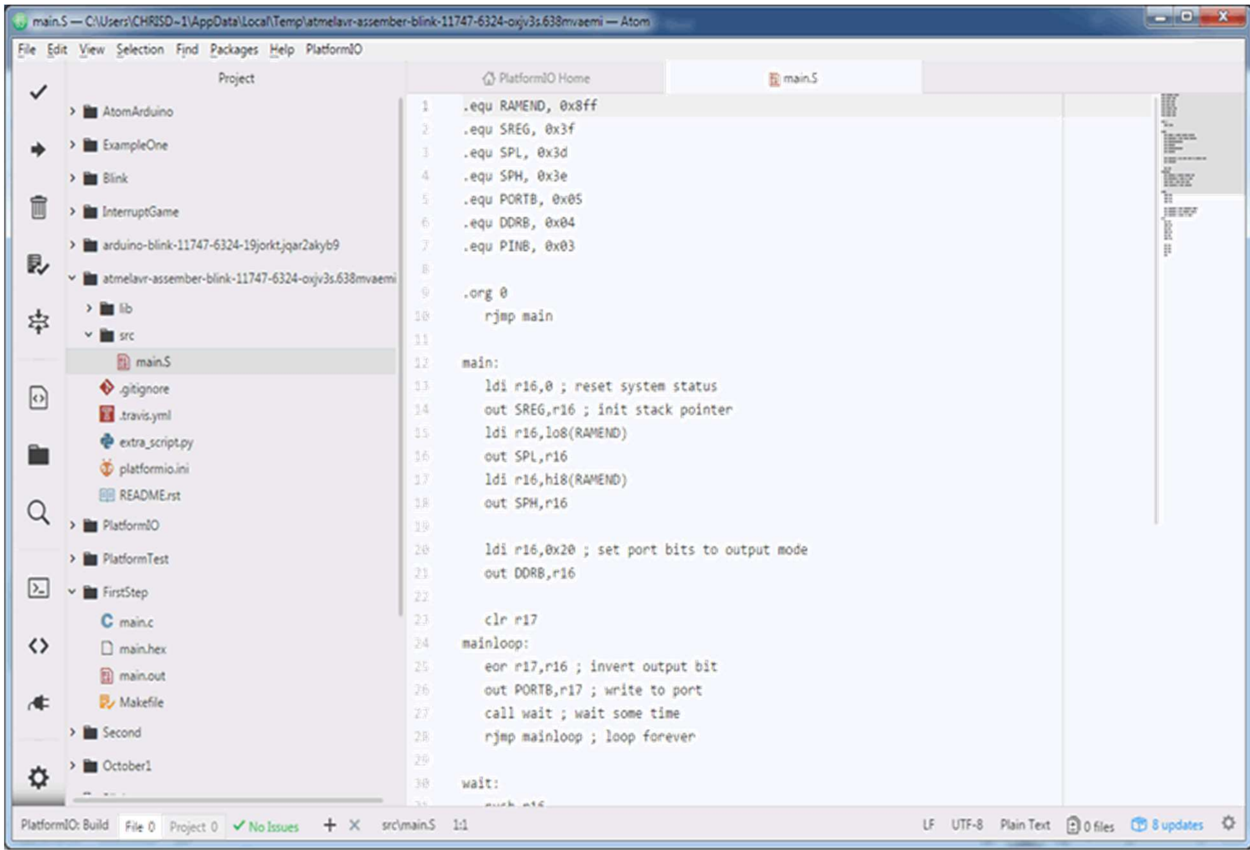

BlinkWithoutDelay - asmtest.S | Arduino 1.8.2
File Edit Sketch Tools Help
BlinkWithoutDelay
asmtest.S §
asmtest.h
// Reference: http://rwf.co/dokuwiki/doku.php?id=smallcpus
#include "avr/io.h"
#include "asmtest.h"
; Define the function asminit()
.global asminit
asminit:
sbi _SFR_IO_ADDR(DDRB), PORTB5 ;Bit 5 = pin 13
ret
; Define the function led()
.global led ;function must be declared as global
led:
cpi r24, 0x01 ;parameter passed by caller in r24
breq turnoff
sbi _SFR_IO_ADDR(PORTB), PORTB5 ;Bit 5 = pin 13
ret
turnoff:
cbi _SFR_IO_ADDR(PORTB), PORTB5 ;Bit 5 = pin 13
ret
Done Saving.
Sketch uses 570 bytes (1%) of program storage space.
Global variables use 13 bytes (0%) of dynamic memory,
16
Arduino/Genuino Uno on COM44

8.1 Pure Assembly

I found it too great a challenge develop pure assembly code within the Arduino IDE that mimicked the way I did it years ago within AVR Studio on PCs. PlatformIO appers to offer something very close on Macs so I'd like to give it a shot this year.

8.1.0 Blink

The pure assembly Blink sketch is provided as an example within the PlatformIO Project Samples.



Appendices

A Development Environments

A Development Environment (DE) consists of a suite of software applications that can run the entire range from converting a programmer's ideas to uploading and running the machine–executable version of those ideas to the target hardware platform or simulator. Tools could include a UML utility, compiler, linker, debugger, uploader, simulator, etc.

A.0 Integrated

An Integrated Development Environment (IDE) provides immediate access to the majority of development tools through and interactive interface.

A.0.0 Arduino IDE

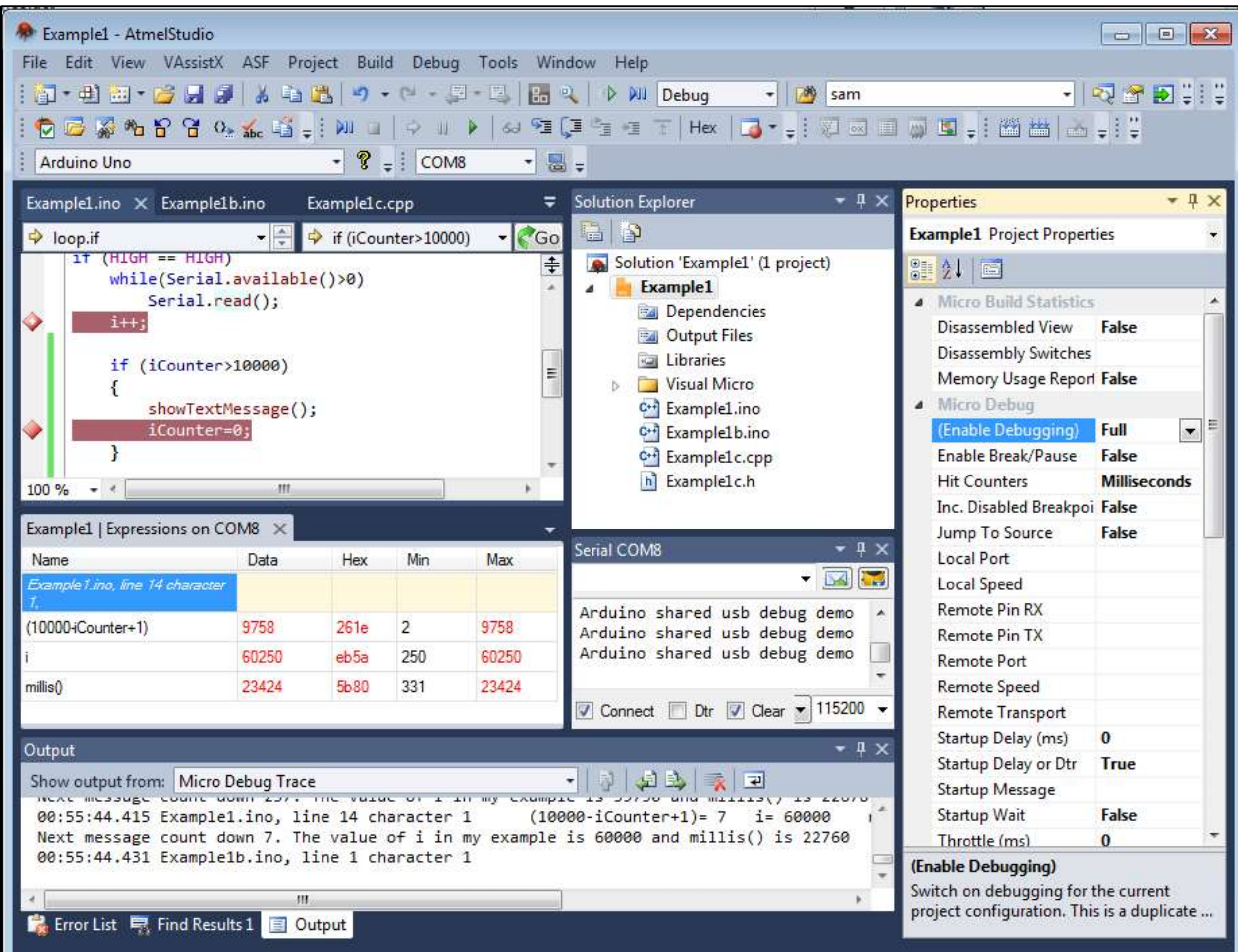
https://www.arduino.cc/en/Main/Software

```
BlinkWithoutDelay | Arduino 1.6.9
File Edit Sketch Tools Help
BlinkWithoutDelay   asmtest.S   asmtest.h

1 #include "asmtest.h"
2
3 int x = 0;
4 int on = 1;
5
6 void setup()
7 {
8   asminit(0);
9 }

Done compiling.
Sketch uses 580 bytes (1%) of program storage space.
Global variables use 13 bytes (0%) of dynamic memory,
1                                   Arduino/Genuino Uno on COM47
```

Due to its familiarity, our first few attempts at AVR Assembly language programming will be undertaken within this environment.

A.0.1 ATMEL Studio 7 (Windows)

http://www.atmel.com/Microsite/atmel-studio/



A.0.2 Crosspack (Mac)

https://www.obdev.at/products/crosspack/index.html

A.0.3 Atom and PlatformIO (Cross-Platform)

Ethan Peterson (ACES '18) brought this IDE to my attention and, to my mind, since it's the closest match to AVR Studio for the Mac, we'll use it for most of our investigations.

A.0.4 WinAVR (Windows)

http://winavr.sourceforge.net/

A.0.5 AVR-Eclipse

http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_GCC_Toolchain

A.1 Standalone

Default ASCII text editors (Mac:TextEdit-Plain Text; Windows:Notepad) can be used to edit your code code.

A.1.0 TextMate (Mac)

The' missing' Mac Editor: https://macromates.com/

A.1.1 Notepad++(Windows)

Useful programming enhancements to Notepad can be found in Notepad++ at https://notepad-plus-plus.org/

A.1.2 Programmers Notepad (Windows)

http://www.pnotepad.org/

B Software: GNU Toolchain

Our study of AVR Assembly Language will make use of the Free Software Foundation's open source GNU Compiler Collection (GCC). Within this broad project, tools are provided for a number of target platforms. Consult the link below for an overview of the toolchain available for the AVR family of microcontrollers:

http://www.nongnu.org/avr-libc/user-manual/overview.html

B.0 GCC

"*GCC focuses on translating a high-level language to the target assembly only. AVR GCC has three available compilers for the AVR: C language, C++, and Ada. The compiler itself does not assemble or link the final code*.

GCC is also known as a "driver" program, in that it knows about, and drives other programs seamlessly to create the final output. The assembler, and the linker are part of another open source project called GNU Binutils. GCC knows how to drive the GNU assembler (gas) to assemble the output of the compiler. GCC knows how to drive the GNU linker (ld) to link all of the object modules into a final executable.

The two projects, GCC and Binutils, are very much interrelated and many of the same volunteers work on both open source projects.

When GCC is built for the AVR target, the actual program names are prefixed with "avr-". So the actual executable name for AVR GCC is: avr-gcc. The name "avr-gcc" is used in documentation and discussion when referring to the program itself and not just the whole AVR GCC system.

See the GCC Web Site and GCC User Manual for more information about GCC."

B.1 GNU Binutils

"The name GNU Binutils stands for "Binary Utilities". It contains the GNU assembler (gas), and the GNU linker (ld), but also contains many other utilities that work with binary files that are created as part of the software development toolchain.

Again, when these tools are built for the AVR target, the actual program names are prefixed with "avr-". For example, the assembler program name, for a native assembler is "as" (even though in documentation the GNU assembler is commonly referred to as "gas"). But when built for an AVR target, it becomes "avr-as". "

B.1.0 avr-as

The assembler. The online reference can be found here:

https://sourceware.org/binutils/docs-2.19/as/

B.1.1 avr-ld

The linker.

B.2 avr-libc

"*GCC and Binutils provides a lot of the tools to develop software, but there is one critical component that they do not provide: a Standard C Library.*

There are different open source projects that provide a Standard C Library depending upon your system time, whether for a native compiler (GNU Libc), for some other embedded system (newlib), or for some versions of Linux (uCLibc). The open source AVR toolchain has its own Standard C Library project: avr-libc.

AVR-Libc provides many of the same functions found in a regular Standard C Library and many additional library functions that is specific to an AVR. Some of the Standard C Library functions that are commonly used on a PC environment have limitations or additional issues that a user needs to be aware of when used on an embedded system.

AVR-Libc also contains the most documentation about the whole AVR toolchain."

B.3 Building Software

"*Even though GCC, Binutils, and avr-libc are the core projects that are used to build software for the AVR, there is another piece of software that ties it all together: Make. GNU Make is a program that makes things, and mainly software. Make interprets and executes a Makefile that is written for a project. A Makefile contains dependency rules, showing which output files are dependent upon which input files, and instructions on how to build output files from input files.*

Some distributions of the toolchains, and other AVR tools such as MFile, contain a Makefile template written for the AVR toolchain and AVR applications that you can copy and modify for your application.

See the GNU Make User Manual for more information."

B.4 AVRDUDE

"*After creating your software, you'll want to program your device. You can do this by using the program AVRDUDE which can interface with various hardware devices to program your processor. AVRDUDE is a very flexible package. All the information about AVR processors and various hardware programmers is stored in a text database. This database can be modified by any user to add new hardware or to add an AVR processor if it is not already listed*

C AVR Assembly Reference

The 8-bit AVR Instruction Set (AVRIS) is detailed in the following pdf:

http://mail.rsgc.on.ca/~cdarcy/Datasheets/doc0856.pdf

C.0 Status Register (Flags), Register and Instruction Operands

(Included) Can be found on pp.1-2 of the AVRIS.

C.1 Program and Addressing Modes

(Included) Can be found on pp.3-10 of the AVRIS.

C.2 Register (GP, I/O & Extended I/O) Summary

(Included) Can be found on pp.9-12 of
http://mail.rsgc.on.ca/~cdarcy/Datasheets/ATmega328PSummary.pdf

C.3 Frequently Used AVR-as Directives

Directives, like many other features, are assembler-dependent (AVRASM vs AVR-as). Since we're using AVR-as the applicable assembler directives can be found here:
https://sourceware.org/binutils/docs-2.19/as/Pseudo-Ops.html#Pseudo-Ops

C.4 Interrupt Vector Table

12.4 Interrupt Vectors in ATmega328 and ATmega328P

Table 12-6. Reset and Interrupt Vectors in ATmega328 and ATmega328P

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Notes: 1. When the BOOTRST Fuse is programmed, the device will jump to the Boot Loader address at reset, see "Boot Loader Support – Read-While-Write Self-Programming" on page 263.
2. When the IVSEL bit in MCUCR is set, Interrupt Vectors will be moved to the start of the Boot Flash Section. The address of each Interrupt Vector will then be the address in this table added to the start address of the Boot Flash Section.

Table 12-7 on page 66 shows reset and Interrupt Vectors placement for the various combinations of BOOTRST and IVSEL settings. If the program never enables an interrupt source, the Interrupt Vectors are not used, and

C.5 Instruction Set

Be the first to clip this slide

Atmega328P Instruction Types

Instruction Type	No. of Instructions
Arithmetic	17
Shift and Rotate	5
Bit-wise Operations	12
Compare Operations	4
Branching	27
Subroutine Calls	6
I/O Instructions	6
Moving Data	29
SREG Bit Operations	18
Program Memory Instructions	11
MCU Control Instructions	6
Total	141

R S Ananda Murthy — Assembler Programming of Atmega328P

C.5.0 Summary of Instructions

(Included) Can be found on pp.13-15 of:

http://mail.rsgc.on.ca/~cdarcy/Datasheets/ATmega328PSummary.pdf

C.5.1 Detailed Instruction Set

See pages 11-157 of http://mail.rsgc.on.ca/~cdarcy/Datasheets/doc0856.pdf