Honours - Computer Interfacing 1

Department of Physics and Electronics
Rhodes University

2013

NAME:________________________________

Contents

Introduction

Welcome the the Computer Interfacing course (a more accurate title would be "Introduction to Micro-controller systems") given by the Department of Physics and Electronics.
By the time you have finished this course you should have acquired a new set of skills applicable to micro-controllers/microprocessors and other embedded computing environments.

Lecture Times

We shall be having 4 lectures a week and you will be expected to complete assignments on your "off' days which re-enforce the topics covered. No handing in is required for these assignments, but the final assessment of the course will involve a task that combines many of the assignments.

Notes and Tools

You are provided with an abridged copy of the ATMega16 datasheet - the full version can be downloaded from the Internet. You may find it usefull to label the following pages in your copy of the datasheets as they will often need to be referenced.

- Pin Configurations - page 2
- Registers - starting at page 9
- Interrupts - page 45
- I/O ports - page 50
  - Port A - page 57
  - Port B - page 58
  - Port C - page 61
  - Port D - page 63
- External Interrupts - page 68
- Timer / counter 0 - page 71
- Timer / counter 1 - page 89
- Timer / counter 2 - page 117
- Analogue Comparator - page 201
- Analogue to Digital converter - page 204
- Electrical Characteristics - page 291
- Register Summary - page 331
- Instruction Set - page 333

Hardware

The equipment provided is as follows:

- Interfacing board (Containing :
  - Programmer / USB to Serial converter
  - LCD
  - Speaker
  - Stepper Motor
  - Breadboard
- Assorted cables.
- Several miscellaneous electronic components provided as the need arises.
- A voltmeter.

1 Getting Started

This course is intended as a first course in micro-controllers, it introduces some of the peripherals available on common micro-controllers with specific reference to the Atmel ATMega16.
The Atmel range of 8bit RISC micro-controllers provides simple 8-pin devices up to 64 pin QPF devices. We will be looking at one of the middle of the range devices in a DIP package.
This course uses assembler since it allows for greater control and efficient code. C provides ease of use and rapid code development, but some of the intricacies are obscured from the programmer.

Software

We will be using AVR Studio 4, as an IDE, simulator and programmer.

Installing

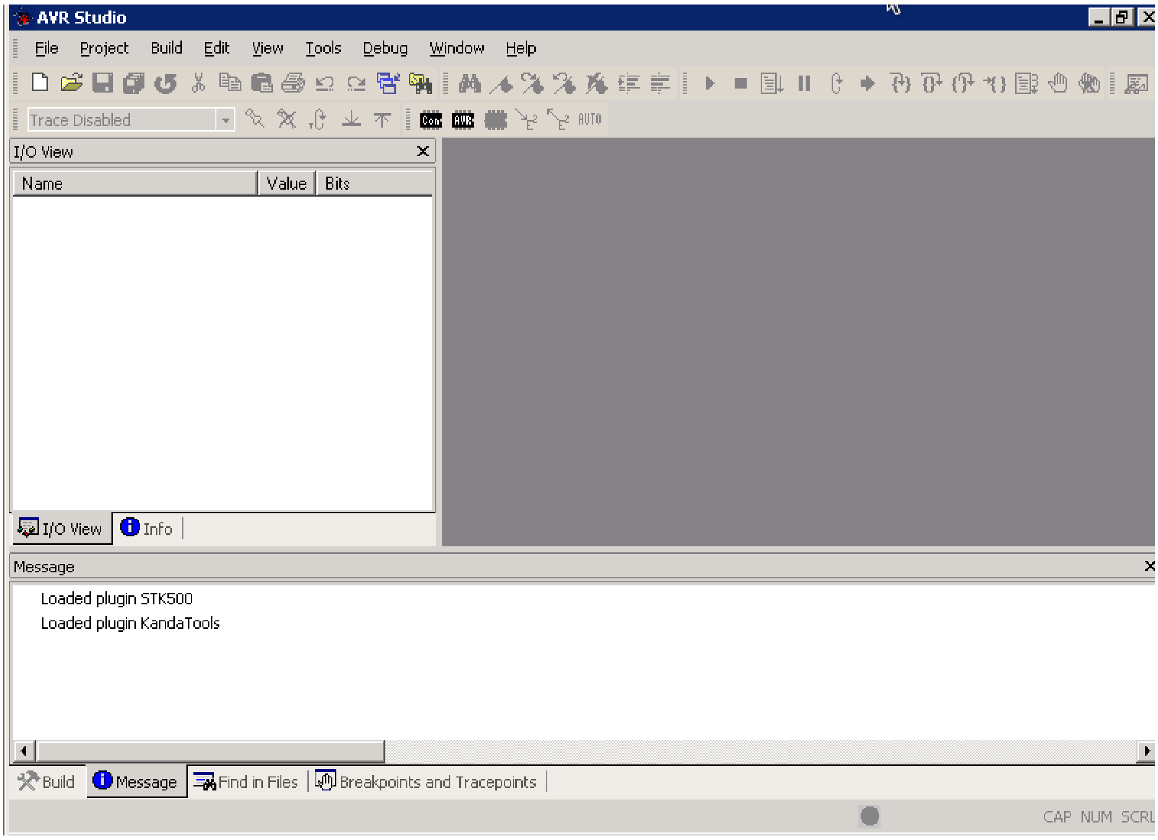
Firstly install AVR Studio 4, followed by the Serial_Install.exe (This allows us to use the programmer and USB to serial converter.

Serial Terminal

You will also need some software to access the serial port of your PC, which we will use later to communicate with the board. Putty is suggested, but any terminal software is sufficient.

AVR Studio

The screen-shot below shows the opening screen of AVR studio. This will be our development environment for this course that we will be using to write, compile and debug code.



Hardware

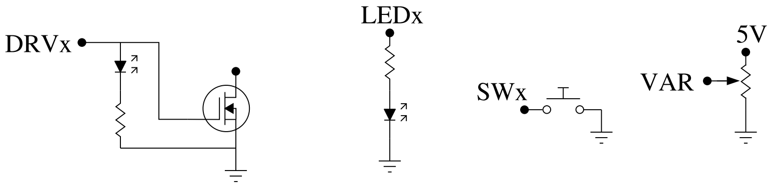
Unlike programming in Computer Science or Information Systems where the programming is essentially abstract, programming for a micro-controller environment means that you must always be aware of the hardware setup, in terms of IO/code space and RAM.

A tour of the board

The development board that we will be using is custom designed to allow for flexibility of hardware setup, all of the IO lines on the micro-controller are taken through to headers where they can be "patched" through to the relevant IO device. The photo below shows the general layout of the board.

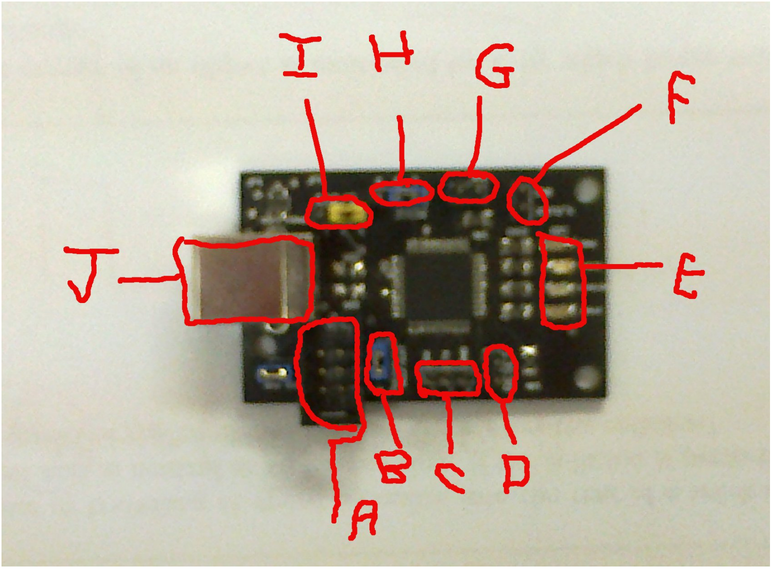
INSERT PHOTO HERE

Below is a schematic representation of the pertinent IO sections.



Programmer

This programmer was designed to provide an easy and cheap method of programming Atmel micro controllers under both Linux and Windows. The programmer mimics an STK500 programmer, except for high voltage serial programming. It also allows the device to act as a USB-to-serial converter to allow serial debugging.

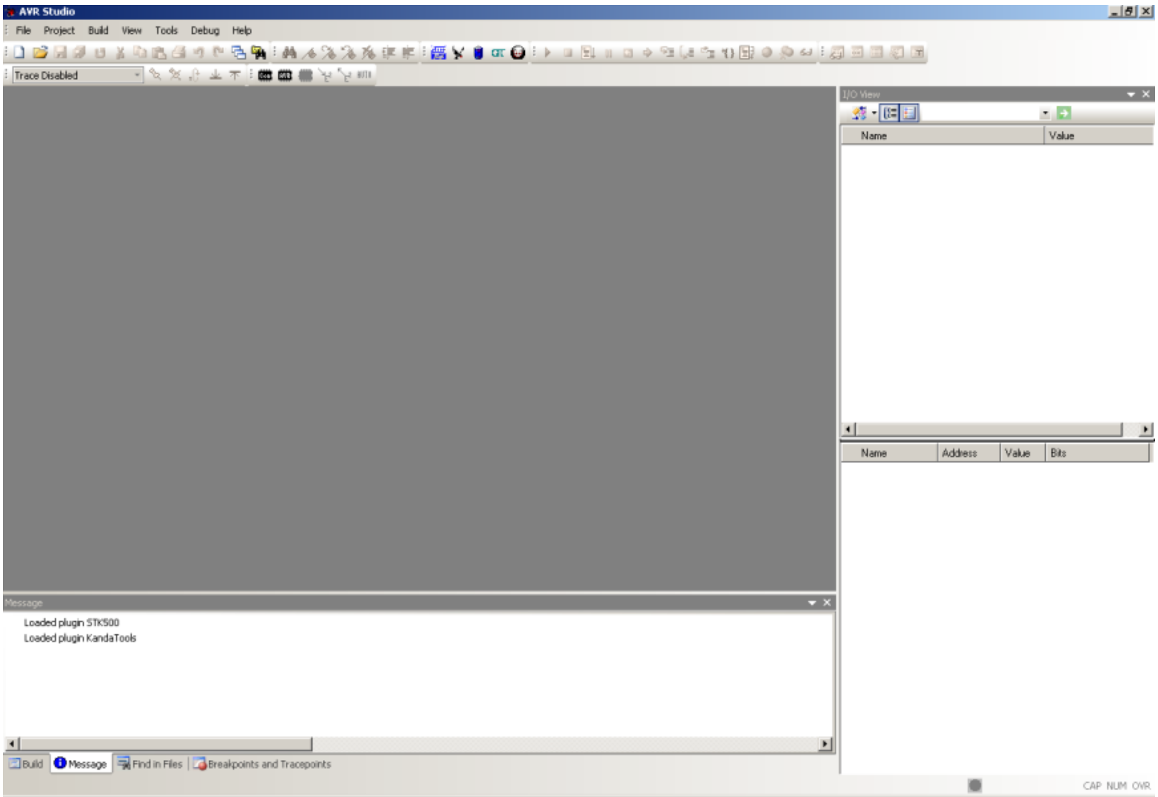


- A: This is the programming header, it is a standard Atmel 6-pin programmer ISP header. The jumper to the left should be on to supply power to the target device or off if the target device has its own power.
- B: This jumper should generally not be moved, it determines whether the programmer's reset pin is connected to the reset line on the programming header (for reprogramming the programmer via ISP or SELF) or if the programmer can control the reset pin on the header (required for programming other devices).
- C: This is the serial header. Please note only 5V serial to be used here. TX is the pin that the programmer will transmit on and should be connected to an RX pin on a target device. RX likewise is the pin that the device will receive data on and should be connected to the TX pin on a target device. The programmer by default is set to 9600,8,n,2.
- D: A jumper can be placed here if you want to reset the device without unplugging the USB cable.
- E: Status indicator LEDs.

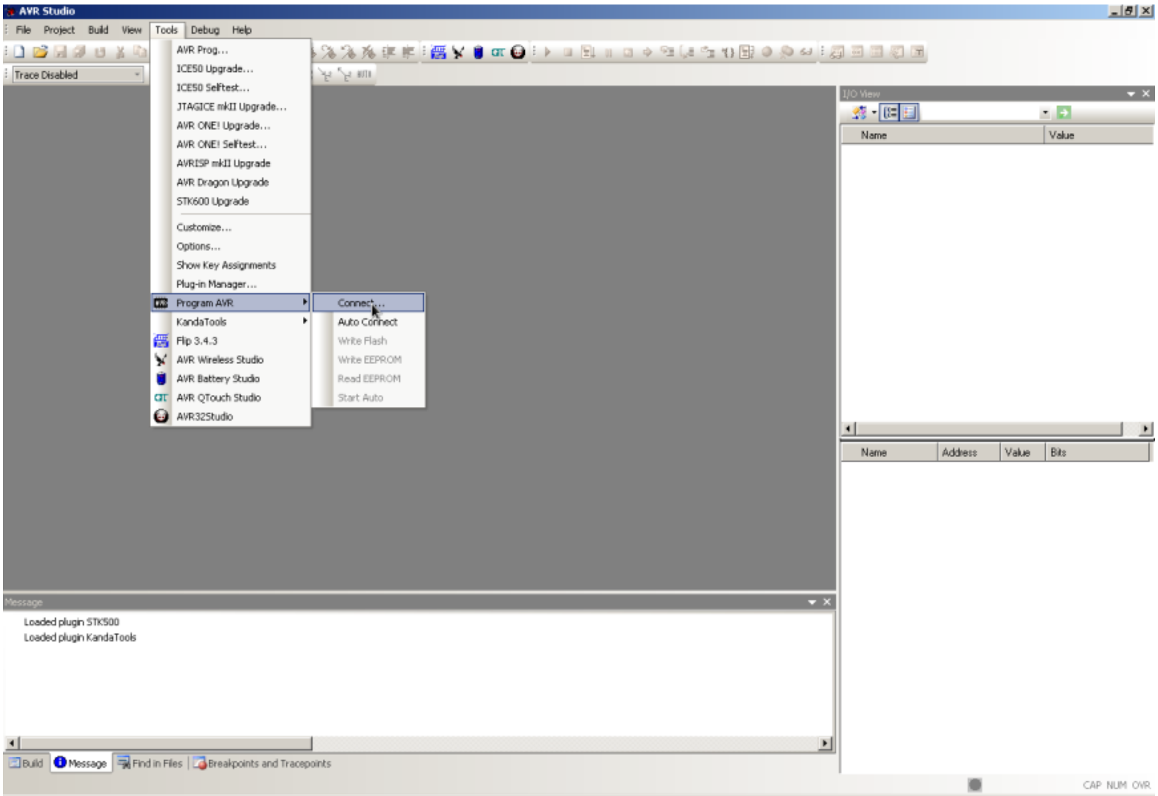
- F: The X_CLOCK line provides a 8MHz signal that can be used to provide a clock signal for programming parts that have not yet been set to internal RC oscillator.
- G: If the Programmer comes out of reset while a jumper is present here it will default to a bootloader allowing firmware updates to be programmed.
- H: The mode selection sets whether the programmer will function as a programmer or a USB to serial converter.
- I: This allows the programmer to be operated at either 3.3 V or 5 V to enable the programmer to be used on 3.3V or 5V parts.
- J: USB connector.

Programming a device using AVR studio

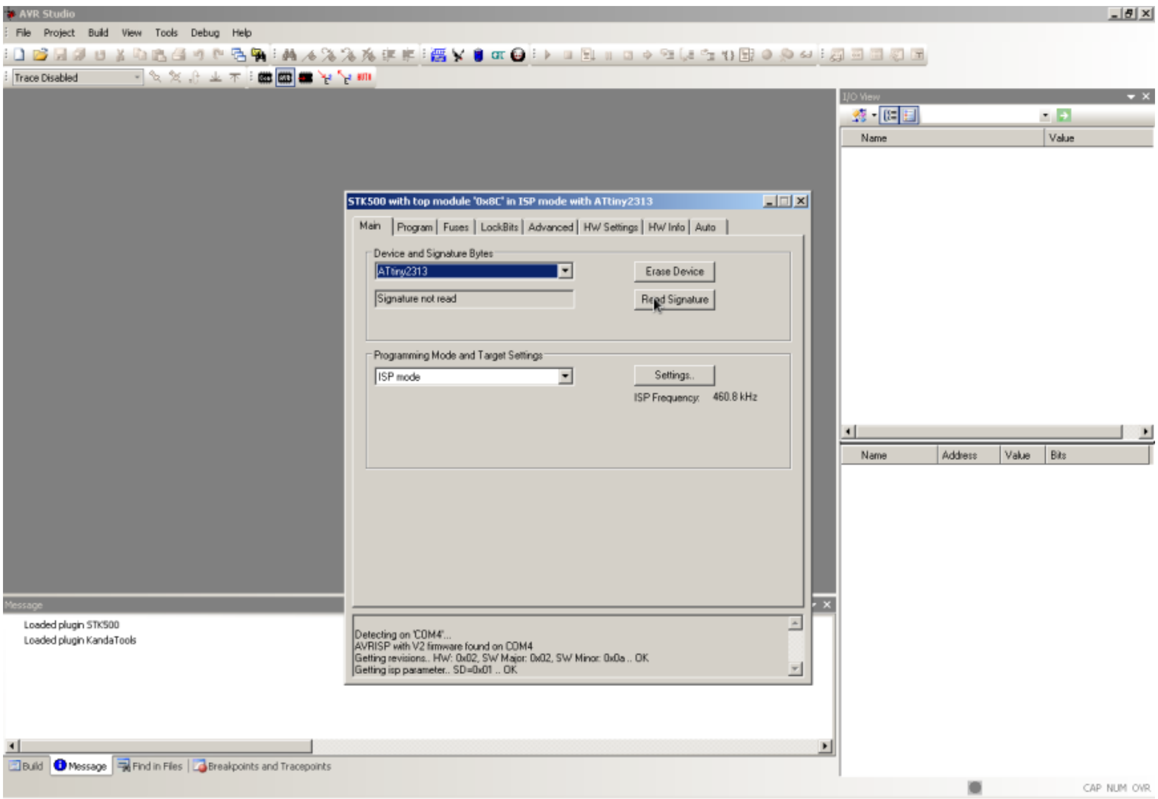
- Open AVR and exit the project wizard.



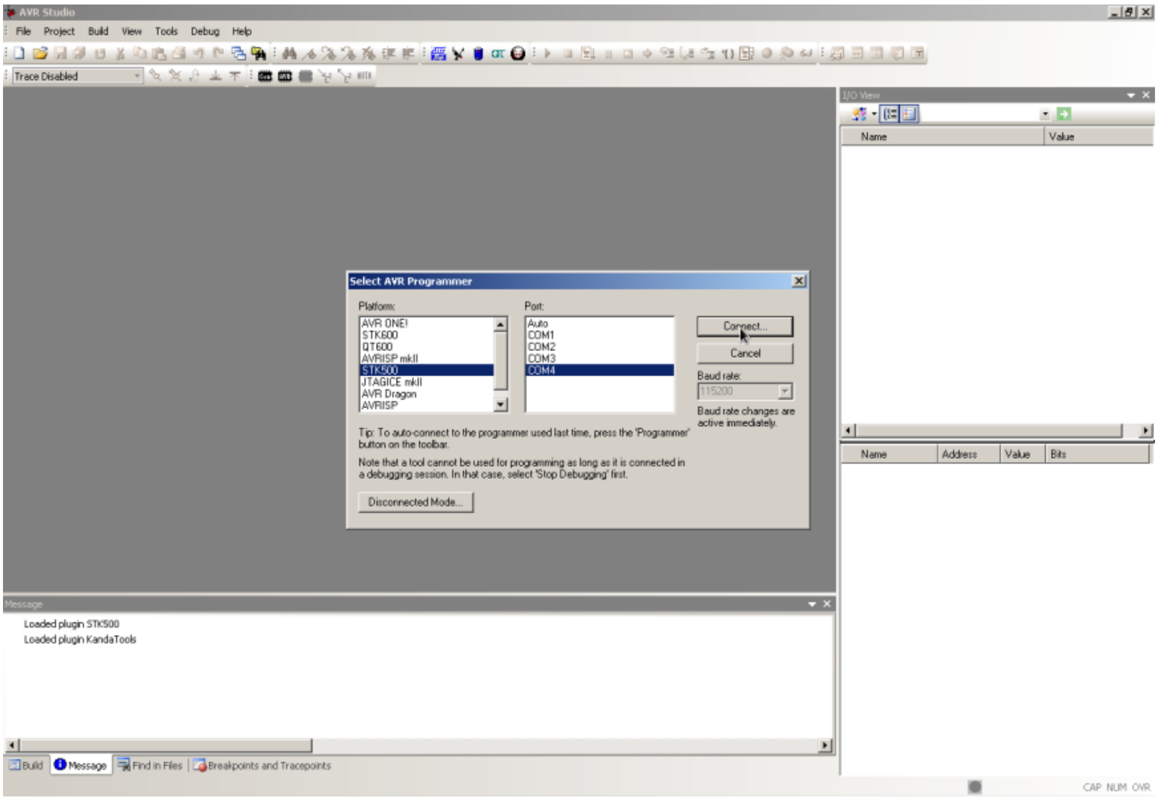
- Select Tools ⇒ Program AVR ⇒ Connect as shown below.



- Select the COM port that Windows has assigned the programmer. (It should be visible under Device manager as a USB serial port.)



- Select the correct target device that you wish to program.
- Select the .hex file to upload to the device.



Playing with the Serial Port

If you change the jumpers on the device to select serial port operation you can then send and receive serial data via USB. Putty is recommended as a serial terminal. To verify serial operation you can jumper RX and TX on the board, characters sent via Putty will then be echoed back.

Switching On

Firstly the following patches should be made to allow the test code to access the peripherals as needed.

These should be in place, but double check the connection.

- LCD Connection:
  - LCDDATA0 - PORTC0
  - . . .
  - LCDDATA7 - PORTC7
  - LCD E - PORTB0
  - LCD R/W - PORTB1
  - LCD R/S - PORTB2
- VAR - PORTA0
- DRV3 - PORTD7
- DRV2 - PORTD6
- DRV1 - PORTD5
- DRV0 - PORTD4
- Programmer Serial TX - PORTD0
- Programmer Serial RX - PORTD1

Start a serial terminal connected to the USB to serial converter. Make sure the programmer is in the correct mode. The jumper should be connecting PINS 2 and 3 on the mode selection.

Now, make sure that the power supply is set to 9 or 12V. Plug it into the power connector on the board and switch the power stitch to the on position.

Follow the instructions provided on the serial terminal.

Writing your first program

We will now write a small "Hello World" type program, it is going to flash some LEDs. Don't worry too much about understanding the code at this point.

```
;LEDFLASH1
; This program toggles portB pin 0

.include "m16def.inc"     ;Can you think why we need this

.def TMP1=R16             ;defines serve the same purpose as in C,
.def TMP2=R17             ;before assembly, defined values are substituted
.def TMP3=R18

.cseg                     ;Tell the assembler that everything below this is in the
   code segment

.org $000                 ;locate code at address $000
rjmp START                ; Jump to the START Label

.org $02A                 ;locate code past the interupt vectors

START:   ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
         out SPL, TMP1
         ldi TMP1, HIGH(RAMEND)
         out SPH, TMP1

         sbi DDRB,0               ; set portB pin 0 to be an output
FLASH:   cbi PORTB, 0             ; set pin to logic 0
         rcall Delay              ; call delay routine
         sbi PORTB, 0             ; set pin to logic 1
         rcall Delay              ; call delay routine
         rjmp FLASH               ; jump to the flash label

Delay:   ser TMP1                 ; TMP1=0xff
Del1:    ser TMP2                 ; TMP2=0xff
Del2:    ldi TMP3, 20             ; TMP=20 decimal
Del3:    dec TMP3                 ; decrement TMP3
                 brne Del3
                 dec TMP2
                 brne Del2
                 dec TMP1
                 brne Del1
                 ret

                 .exit
```

You should be able to compile the code and upload it to the ATMega16 and run it and watch the LED flash.

Don't forget to patch PORTB0 through to an LED or you will not see the flashing light.

2 Microcontroller / Microprocessor Basics

`Read pages 3-17 in the Mega16 datasheet.`

Firstly we should differentiate between a microprocessor and a microcontroller. A microprocessor refers to a processor that has access to RAM and interrupts, a microcontroller contains a microprocessor, RAM and other related peripherals.

The Atmel range of 8-bit RISC processors (of which the ATMega16 is an example) are, as the name suggests, RISC (Reduced Instruction Set Computer) processors. RISC processors have a much smaller instruction set than the x86 or x86_64 architectures that you may be familiar with. The 8-bit refers to the size of a data byte.

Some features of this range of micro-controllers include:

- Mostly fixed execution time instructions.
- 32 general purpose registers.
- Up to 16MHz system cloock.
- 3.3 to 5 V operation.
- a wide varietty of on-chip peripherals.
- Internal program and data memory.
- In-System Programmable.
- Multiple package types.

It should be pointed out at this point that processors are also classified according to their memory architecture. You are probably familiar with Von Neuman or Princeton model, where the memory (program and data) is unified. The model we deal with now is called the Harvard architecture, where program and data memory are kept separate.

(Tut question - Can you think of some advantages for each model?)

Now we come to yet another type of classification, how the micro-controller stores and accesses data internally. The major types are:

- Stack: In a stack machine all operands of an operation are pushed onto the stack, the operation is performed and the result is popped off the stack.
- Accumulator: With this type the accumulator is always one of the operands of the operation. After the operation has completed, the result is available in the accumulator.
- Register-memory: In this model a register is loaded with a value, an operation is then performed with another value and the result is left in the register. The value must then be moved back to a memory location.
- Register-Register: In this architecture two different registers are loaded with values. An operation is then performed ans the result is left in another register. This value must then be moved into memory.

Now that we have had on overview of how the can be classified by various characteristics we are going to have a look at the regular operation of a micro-controller, with specific attention to the Mega16.

2.1 How a microcontroller starts up

When a microcontroller (or microprocessor) starts up they all (without exception) execute the first instruction in their program memory, usually found at location 0. This only happens after voltage levels and the system clock are stable. In terms of the Mega16 these startup parameters are set by programming the fuse bits. (We are not going to be looking too closely at this since it only really impacts hardware design.)

You may recall the section of code shown alongside. The first line tells the compiler to locate the following code at the address 0x000 in program memory. The instruction that is situated there is a jump to the label start. The next line locates the next instruction at location 0x02a, which is then given the label (to the compiler only) START.

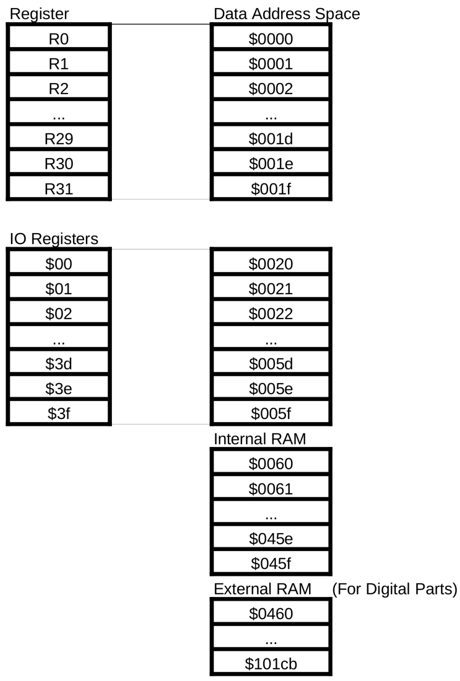
```
.org $000
rjmp START
.org $02A
START: ldi
```

2.2 Memory in the ATMega16

The ATMega16 (and other micro-controllers in the same range) have the basic layout of memory shown alongside. Where the registers and IO-registers are mapped into a single address space that is continuous with the internal SRAM of the microcontroller.

Please note that the electrical design of the registers, IO-registers and SRAM are not identical, they merely occupy a continuous address space.

The external SRAM is shown on the map because the *digital* parts have connections where the designer can add external SRAM (up to a total of 64kB). The part that we are using does not have support for external RAM. We can do all that we need to in 1K of RAM.



2.2.1 Registers

The 32 general purpose registers can be used as operands in operations and will also hold the results of operations. These registers are fast, operations involving only registers will generally take only one clock cycle to complete. The registers are volatile (they do not retain their values on reset or power down).

Special Purpose Registers

Six of the registers (R26/R27,R28/R29,R30/R31) can be used as 16-bit registers for memory access using the ST, STD, LD and LDD instructions (this will be fully explained later when we look at the instruction set).

IO Registers

These are the registers that provide information as to the status of peripherals and allow the control of those peripherals. Without these registers the ATMega16 would effectively be a microprocessor and be unable to interact with the outside world. Two of these are of particular importance and will be covered here.

Status Register:

Basically this register holds information about the most recent arithmetic operation that has been performed. Each bit in this register has special significance. (See page 9 of the ATMega16 data sheet for full description.) (For future reference - this register is not saved on entry to an interrupt routine, you must manually save it.)

Descriptions of the bits are summarized below:

- Bit 7 I: Global Interrupt Enable
  The Global Interrupt Enable bit must be set for the interrupts to be enabled. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

- Bit 6 T: Bit Copy Storage
  The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

- Bit 5 H: Half Carry Flag
  The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry is useful in BCD arithmetic. (Carry between nybbles).

- Bit 4 S: Sign Bit, $S = N \oplus V$
  The S-bit is always an exclusive or between the Negative Flag N and the Twos Complement Overflow Flag V.

- Bit 3 V: Twos Complement Overflow Flag
  The Twos Complement Overflow Flag V supports twos complement arithmetics.

- Bit 2 N: Negative Flag The Negative Flag N indicates a negative result in an arithmetic or logic operation.

- Bit 1 Z: Zero Flag
  The Zero Flag Z indicates a zero result in an arithmetic or logic operation.

- Bit 0 C: Carry Flag
  The Carry Flag C indicates a carry in an arithmetic or logic operation.

Stack pointer:

The stack is mainly used for storing temporary data, usually when entering interrupt routines and sub-routines. The stack register points to the “top” of the stack. (Note that the “bottom” of the stack is actually the highest memory location, so that as you push data onto the stack it grows “downward” into your SRAM.) You must always set up the stack in your code before subroutines are used or interrupts occur. This also means that excessive use of nested subroutines within interrupts could allow the stack to over write data stored in SRAM.

2.2.2 RAM

This is the 1024 bytes (8-bits wide) of storage space that can be used to store variables and data. The RAM is also volatile. Instructions involving the on-chip SRAM generally take 2 or more clock cycles to complete. Non-volatile RAM can be added to micro-processor systems, but it is generally slower and more costly.

2.2.3 Program Memory

The program memory (16K bytes) is organised as 8K words each word being 16 bits. Program memory is non-volatile and is retained while the power is off. Program memory is flash based (same electrical characteristics as your USB flash-stick), this type of memory typically has at least 10000 write/erase cycles before errors creep in. (Remember flash sticks do not last forever).

2.2.4 EEPROM

EEPROM on the Atmel range that we deal with is not mapped into address space in any way, but is rather accessed as a peripheral, so we will cover its use when we get to peripherals.

2.3 System Clock

The microcontroller is a complex synchronous state machine. It responds to program steps in a sequential manner as dictated by a user-written program. The micro-controller sequences through a predictable fetch–decode–execute sequence. Each unique assembly language program instruction issues a series of signals to control the micro-controller hardware to accomplish instruction related operations. The speed at which a micro-controller sequences through these actions is controlled by a precise time base called the clock. The clock source is routed throughout the micro-controller to provide a time base for all peripheral subsystems. The ATMega16 may be clocked internally, using a user-selectable resistor capacitor (RC) time base, or externally. We will be using an internal 8MHz RC Oscillator to provide our clock source.

2.4 Assembler Directives

`Read the ‘‘Assembler Help.pdf’’ for a more complete language reference.`

The Atmel assembler supports a number of directives, these are not assembled into operation codes (machine language), rather they provide an easy way of changing locations in memory, defining macros, and initialising memory.

BYTE - Reserve bytes to a variable

The BYTE directive reserves memory resources in the SRAM or EEPROM. In order to be able to refer to the reserved location, the BYTE directive should be preceded by a label. The directive takes one parameter, which is the number of bytes to reserve. The directive can not be used within a Code segment (see directives CSEG, DSEG, and ESEG). Note that a parameter must be given. The allocated bytes are not initialized.

Example:

```
.DSEG
var1:   .BYTE 1          ; reserve 1 byte to var1
table:  .BYTE tab_size   ; reserve tab_size bytes
.CSEG
        ldi r30,low(var1) ; Load Z register low
        ldi r31,high(var1) ; Load Z register high
        ld r1,Z          ; Load VAR1 into register 1
```

CSEG, DSEG and ESEG - Setting memory segments

These three directives are used to locate code and variables. Certain other directives are limited to which segments they can be used in.
Example:

```
.ESEG
eevarlst:        .DW 0,0xffff,10
.DSEG ; Start data segment
var1:            .BYTE 1 ; reserve 1 byte to var1
table:           .BYTE tab_size ; reserve tab_size bytes.
.CSEG
        ldi r30,low(var1) ; Load Z register low
        ldi r31,high(var1) ; Load Z register high
        ld r1,Z ; Load var1 into register 1
```

DB - Define constant bytes in the code and EEPROM segments

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 expressionlist contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. If the expressionlist 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 assemby 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.

Example:

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
const2: .DB 1,2,3
```

DW - Define constant word in the code and EEPROM segments

The DW directive reserves memory resources in the program memory or the EEPROM memory. 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 memory location.
Example:

```
.CSEG
varlist:         .DW 0, 0xffff, 0b1001110001010101, -32768, 65535
.ESEG
eevarlst:        .DW 0,0xffff,10
```

DEF - Define a symbolic name for a register

The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used in the rest of the program to refer to the register it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.
Example:

```
.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
```

EQU - Set a symbol equal to an expression

The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined.
Example:

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2
.CSEG                           ; Start code segment
        clr r2                  ; Clear register 2
        out porta,r2            ; Write to Port A
```

Include - Include another file

The INCLUDE directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until end of file (EOF) or an EXIT directive is encountered. An included file may itself contain INCLUDE directives.
Example:

```
; iodefs.asm:
.EQU sreg = 0x3f          ; Status register
.EQU sphigh = 0x3e        ; Stack pointer high
.EQU splow = 0x3d         ; Stack pointer low

; incdemo.asm
.INCLUDE iodefs.asm       ; Include I/O definitions
        in r0,sreg        ; Read status register
```

Macros

The MACRO directive tells the Assembler that this is the start of a Macro. The MACRO directive takes the Macro name as parameter. When the name of the Macro is written later in the program, the Macro definition is expanded at the place it was used. A Macro can take up to 10 parameters. These parameters are referred to as @0-@9 within the Macro definition. When issuing a Macro call, the parameters are given as a comma separated list. The Macro definition is terminated by an ENDMACRO directive. By default, only the call to the Macro is shown on the listfile generated by the Assembler. In order to include the macro expansion in the listfile, a LISTMAC directive must be used. A macro is marked with a + in the opcode field of the listfile.
Example:

```
.MACRO SUBI16                          ; Start macro definition
        subi r16,low(@0)               ; Subtract low byte
        sbci r17,high(@0)              ; Subtract high byte
.ENDMACRO

.CSEG                                  ; Start code segment
SUBI16 0x1234,r16,r17                  ; Sub.0x1234 from r17:r16
```

ORG - Set location in memory

The ORG directive sets the location counter to an absolute value. The value to set is given as a parameter. If an ORG directive is given within a Data Segment, then it is the SRAM location counter which is set, if the directive is given within a Code Segment, then it is the Program memory counter which is set and if the directive is given within an EEPROM Segment, it is the EEPROM location counter which is set. The default values of the Code and the EEPROM location counters are zero, and the default value of the SRAM location counter is the address immediately following the end of I/O address space (0x60 for devices without extended I/O, 0x100 or more for devices with extended I/O) when the assembling is started. Note that the SRAM and EEPROM location counters count bytes whereas the Program memory location counter counts words. Also note that some devices lack SRAM and/or EEPROM.
Example:

```
.DSEG                       ; Start data segment
.ORG 0x120                  ; Set SRAM address to hex 120
variable: .BYTE 1           ; Reserve a byte at SRAM adr. 0x120
.CSEG
.ORG 0x10                   ; Set Program Counter to hex 10
        mov r0,r1           ; Do something
```

SET - Set a symbol equal to an expression

The SET directive assigns a value to a label. This label can then be used in later expressions. Unlike the .EQU directive, a label assigned to a value by the SET directive can be changed (redefined) later in the program.
Example:

```
.SET FOO = 0x114            ; set FOO to point to an SRAM location
lds r0, FOO                 ; load location into r0
.SET FOO = FOO + 1          ; increment (redefine) FOO. This would be illegal if using .
    EQU
lds r1, FOO                 ; load next location into r1
```

2.5 Instruction Set

The full instruction set summary is given in the extracts from the ATMega16 datasheet. We will look at one or two illustrative examples and explain how to read the summary provided.

Arithmetic Instructions

These operations, as their name suggests, involve arithmetic operations on one or more registers. You cannot perform arithmetic operations on bytes in RAM, they must first be read into a register and then later stored in RAM again.

Branch Instructions

These operations change the flow of the program based on certain criteria or simple jump to another location regardless.

Data Transfer Instructions

These operations deal with the transfer of data between registers, between a register and RAM or between registers and IO registers.

Bit and Bit Test Instructions

These instructions perform bitwise operations on ports and flags as well as bit operations of registers.

MCU Control Instructions

These instructions (NOP, SLEEP, WDR and BREAK) control the processor's function

2.6 Structure of an assembly language program

Simply put there is no definitive structure to an assembly language program. All of the fancy directives and includes could be left out, but the program would be exceedingly difficult to write and to read. We follow a few basic guidelines that make coding easier and reading the program possible:

- Use the predefined processor includes for the processor that you are using so that you can refer to things by the names given in the register summary. i.e. PORTD instead of the memory location 0x32.
- Set up the code correctly. Set the stack pointer in the beginning, avoid placing code in the interrupt vectors. Set up dummy interrupt routines for unused interrupt vectors so that if you accidentally enable them it will not adversely affect the running of your code.
- Use intelligible labels for code sections.
- Use meaningful register definitions.
- Don't alter registers in interrupt code if you have not preserved their value, unless you don't use that register in interruptable code.
- Comments are essential, otherwise the code is next to impossible to decipher.

2.7 IO Basics

We will be covering IO ports in more detail a bit later, but for now we will cover enough to handle simple IO.
There are 3 registers that cover the use of a "PORT" (PORTA, PORTB etc) the actual IO pins on the micro-controller. The three controlling IO-registers are PORTx, PINx, and DDRx. The DDRx register controls whether the port in question is set up as an input or output. If the bit in DDRx corresponding to an IO-pin is "0" then the pin is configured as an input (the default value). Conversely if the bit is a "1" then the pin will function as an output. This means that on one port you can have both inputs and outputs. To read a binary value in from the pins you must read the value from the PINx IO-register. A "0" corresponding to 0V and a "1" corresponding to 5V. You can read the PINx register even if the port is configured as an output. To output a value to the pins you must write a value to the PORTx register.
There is more to this which we will see later, but for now all you need to know is that to make the LEDs on the board light up, you must write a logical "0" to the corresponding bit on the port.

3 Peripherals

Without using the peripherals of the micro-controller we have no way of getting data into or out of our code, we also have no way of affecting the world outside the processor.

We will briefly cover the use of the peripherals here, but these notes are no substitute for the data sheet when is comes to the finer points of each peripheral.

3.1 IO Ports

`Read pages 50-67 of the datasheet.`
The IO ports (PORTA...PORTD) also have alternative functions, which will be discussed in the relevant sections below.
As discussed before, the operation of the general IO ports (A-D) are each controlled by three registers. What was not mentioned was the effect of changing the PORTx values when the port is configured as an input. If a particular pin is set as an input and a logical "1" is written to the correxponding bit in the PORT register, a "pull-up" resistor is activated, which means that if the pin is not tied to a specific voltage (i.e. 0V) it will be pulled up to 5V. When the corresponding PORT bit is a logical "0" the pins become "tristated" or high impedance.
Leaving pins as high impedance or tristate will affect the hardware design as well as power consumption, which will affect overall cost of design, both monetary and power (which is critical in designs operating from batteries).

3.2 EEPROM

`Read pages 18-23 of the datasheet.`
We have seen that we can locate constants in EEPROM using the `.ESEG` assembler directive. This is actually written to the micro-controller in a separate file (.eep) when uploading code. You may wonder why we bother with EEPROM when we can alter program memory from within a running piece of code? Well firstly it is possible to corrupt program memory this way (if the voltage is not kept at the correct level when writing), EEPROM memories also have a greater lifetime in terms of write/erase cycles.
EEPROM provides a convinient way to change constants in a design without changing the code space. It also allows for retention of data during power outages.

Data can be stored in EEPROM via the `.DB` or `.DW` directives at compile time and read out of EEPROM using an ISP. This would be a bit useless if you could not access it from within your code as well.
Writing to and reading from the EEPROM is controlled via 4 registers (EEARH, EEARL, EEDR, EECR). (See page 19 of the data sheet for full description.)

How to read from EEPROM

- Make sure that a write is not in progress
- Write the EEPROM address to be read to the EEAR register (high byte and then low byte).
- Trigger a read by writing a "1" to the EEPROM Read Enable bit in the EECR.
- Read the value that was retrieved from EEPROM from EEDR.

How to write to EEPROM

- Make sure that a write is not in progress

- Write the EEPROM address to be written to the EEAR register (high byte and then low byte).
- Write the Data to EEDR.
- Trigger a write by writing a "1" to the EEPROM Write Enable bit in the EECR.

3.3 Serial

Serial communications interfaces are popular since they require a minimum of data lines (compared to parallel interfaces). We will look at a few of the popular ones.

3.3.1 SPI

`Look at pages 135-143 of the electronic version of the datasheet if you want to know more..`
We are not going to be looking at this in practice (due to a lack of SPI devices to interface with), but in terms of number of supported devices this is an industry leader. What sets it aside from the RS232 interface is that all data transfers are controlled by the micro-controller.
An SPI system consists of a master device and any number of slave devices. The master can select the device to be read from / written to and data is then clocked between them.
Data is written from the master to the slave on the MOSI (Master Out Slave In) line, while at the the same time a byte of data is transmitted from the slave to the master using the MISO (Master In Slave Out) line. The SCK (Serial Clock) line provides a means to synchronise bit timing.

3.3.2 RS-232

`Read pages 144-171 of the datasheet.`
RS232 is a legacy standard that is still in use today, mainly owing to its simplicity and robustness. The physical specification (in its simplest form) calls for 2 data lines (there is also an implicit ground connection), one line for each direction of data transfer. The data lines use +12V and -12V as signal levels. This allows for a high degree of noise immunity. There is no clock line to synchronise data, this means that all data bits are of a set duration and the transmitter must ensure that timing is maintained once a transmission is started. The bit-timing is effectively controlled by setting the "Baud Rate", which is given in bits per second. This means that each bit has a duration of $\frac{1}{\text{baud rate}}$

Using the USART on the Mega16

- Firstly the baud rate must be set using the UBRR (USART Baud Rate Register).
- The USART must be enabled for Rx and Tx as well as the data formatn the control registers, USCRA/B/C
- Data can then be written to UDR (USART Data Register) which will be written out onto the lines and incomming data can be read from the UDR as well.

3.3.3 Dallas 1-wire

There is no actual peripheral on the AVR micro controller that supports the Dallas 1-wire interface, but this is a popular interface and we will look at using an IO port and writing the software to read these devices. The full specification for the 1-wire protocol can be found on the web, but we will summarise it here.

- The data line is held high (+5V) via 4.7kΩ resistor, with respect to the ground line.
- When an iButton is attached to these lines, it charges up an internal power source that i will use later.

- If the Master (micro-controller in this case) doe nothing the iButton will remain inert.
- The master then sends a reset pulse, by actively pulling the data line low for a set time. It then releases the line. The 1-wire device will then pull the device low to indicate its presence.
- the master detects this presence by monitoring the status of the line.
- The Master can then read and write to the slave device.
- We will only consider the reading of the serial number in this course (it is the only operation supported by the iButtons in use on campus).
- The master then writes the code 0x33 onto the bus, which its the instruction to send the serial number of the device.

  Data is written to the device by means of holding the data line low for a set amount of time for each bit. A high bit has different duration to a low bit.
- The master then reads the number returned by the device.

  *The master reads the data back by actively pulling the line low. The slave then holds the line low for a certain amount of time, based on whether it is sending a high or low bit.

  *The master has to time how long the line was held low by the slave to deduce whether it was sending a 1 or a 0.

3.4 ADC

`Read pages 204-221 of the datasheet.`

An ADC converts an analog voltage on its inputs to a digital value based on a reference voltage (A_{REF}). The minimum value 0, is given by the 0V rail and the maximum value 1023 is given by the voltage present on the A_{REF} pin minus one LSB. Therefore a binary 1 represents the value $\frac{A_{REF}}{1024}$.

The ADC on the ATMega16 is a successive approximation ADC, this is not the fastest type of ADC but for an on-chip peripheral it balances the need for speed against complexity and cost.

The ATMega has 8 single ended inputs to the ADC (selectable by appropriate ADMUX settings) or 4 differential inputs with differing gains (also selectable by appropriate ADMUX settings.

The ADC can operate in one of 2 modes, Single Conversion or Free-Running mode, each mode has its uses.

The ADC also features a settable prescaler, which allows the user to change the speed of the conversion. Faster speeds can lead to inaccuracy, but a full 10-bit accuracy is not always needed.

There are methods to decrease the effect of digital noise on the ADC, these are done either using solid design and layout of the circuit as well as putting the processor to sleep (eliminating a major source of digital noise) while the conversion takes place. (Familiarise yourself with these methods as given in the ATMega16 datasheets).

3.5 Interrupts

`Read pages 45-49 and 68-70 of the datasheet.`

The easiest way to describe interrupts to a windows programmer is to liken them to events in Visual Basic. Interrupt code is run when a particular hardware event happens (assuming that particular interrupt event is enabled).

A list of the interrupts and their associated vectors for the ATMega16 is given in the table below:

Vector No.	Program Address	Source	Interrupt Definition
1	0x000	RESET	Reset from any source
2	0x002	INT0	External Interrupt Request 0
3	0x004	INT1	External Interrupt Request 1
4	0x006	TIMER2 COMP	Timer/Counter2 Compare Match
5	0x008	TIMER2 OVF	Timer/Counter2 Overflow
6	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	0x00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	0x00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	0x010	TIMER1 OVF	Timer/Counter1 Overflow
10	0x012	TIMER0 OVF	Timer/Counter0 Overflow
11	0x014	SPI, STC	Serial Transfer Complete
12	0x016	USART, RXC	USART, Rx Complete
13	0x018	USART, UDRE	USART Data Register Empty
14	0x01A	USART, TXC USART	Tx Complete
15	0x01C	ADC	ADC Conversion Complete
16	0x01E	EE_RDY	EEPROM Ready
17	0x020	ANA_COMP	Analog Comparator
18	0x022	TWI	Two-wire Serial Interface
19	0x024	INT2	External Interrupt Request 2
20	0x026	TIMER0 COMP	TimerCounter0 Compare Match
21	0x028	SPM_RDY	Store Program Memory Ready

When an interrupt occurs the current program counter is stored on the stack and then code execution commences from the program address corresponding to the interrupt. When the RETI instruction is called the program counter is read off the top of the stack and execution of the originally running code continues. Note that the Status Register is not automatically stored when

entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software. You must also manually save any registers that you use at the beginning of your interrupt code and restore them in the correct order at the end.
It is generally bad form to enable interrupts within interrupts as the hardware stack can grow to fill RAM rather quickly.
Most of the peripherals on the ATMega have a corrsponding interrupt so that the peripheral can be triggered, other code can be run and then when the peripheral is finished with its task the interrupt code can be execute to manipulate data or program flow based on the needs.

3.6 Timers / Counters

`Read pages 71-134 of the datasheet.`
The timers/counters on the ATMega provide an accurate timing or counting solution. All of the functions depend on a digital counter with a fixed (setable) clock source. There operation and use is best understood by looking at code examples and following the information provided by the ATMega datasheet.

3.6.1 Pulse Width Modulation

Pulse Width Modulation (besides a method for encoding information) is used to control the average power dissipated by a circuit element. We will use it mostly for controlling the brightness of an LED, but it can be used to control motors, heaters or almost any other passive device.

Instead of applying a changing voltage (and hence supplying a changing current and power to the device) a constant voltage is either supplied or not supplied with a varying mark-space ratio, thereby delivering a varying average power.

The timer/counters in the ATMega16 provide an easy method to produce pulse width modulation, by running their counters and comparing the value of the running counter to a user specified value.

3.7 Analog Comparator

`Read pages 201-203 of the datasheet.`
The analog comparator is just what is says, it compares the voltage on two different pins and determines which is higher. We will use this to make a successive approximation ADC (the same type of ADC built into the ATMega16).

3.8 Watchdog Timer

`Read pages 37-44 of the datasheet.`

3.9 LCD

Liquid Crystal Displays are used in the world of embedded electronics to provide a display for human interaction. The LCD modules that we are going to be using actually have a microcontroller on board to handle character generation and display, we are merely going to be accessing the functions of this module through a parallel interface.

Giving instructions to the LCD

When RS is low, any data written to the LCD is effectively a command to do something or set something in the LDC display. This enables us to ,move the cursor around the screen, move the screen around the characters in the display RAM, reset the display etc.

Writing to (DDRAM) the display

When you want to put a character on the display, you write the character to Data Display RAM, the LCD module then actually displays that character. The position of the character written depends on where the cursor is currently positioned. The cursor position can be altered by updating the DDRAM address.
A basic set of driver functions is provided in the LCD.asm file. Make sure that you understand these driver functions.

```
.MACRO LCD_WRITE
 CBI PORTB, 1
.ENDMACRO
.MACRO LCD_READ
 SBI PORTB, 1
.ENDMACRO
.MACRO LCD_E_HI
 SBI PORTB, 0
.ENDMACRO
.MACRO LCD_E_LO
 CBI PORTB, 0
.ENDMACRO
.MACRO LCD_RS_HI
 SBI PORTB, 2
.ENDMACRO
.MACRO LCD_RS_LO
 CBI PORTB, 2
.ENDMACRO

;This is a one millisecond delay
Delay:          push r16
                ldi r16, 11
Delayloop1:     push r16
                ldi r16, 239 ; for an 8MHz xtal
Delayloop2:     dec r16
                brne Delayloop2
                pop r16
                dec r16
                brne Delayloop1
                pop r16
                ret
; waits 800 clock cycles (0.1ms on 8MHz clock)
Waittenth:      push r16
                ldi r16, 255
decloop:        dec r16
                nop
                nop
                brne decloop
                pop r16
                ret

; return when the lcd is not busy
Check_busy:     push r16
                ldi r16, 0b00000000
                out DDRC, r16     ; portc lines input
                LCD_RS_LO         ;RS lo
                LCD_READ          ;read
Loop_Busy:      rcall Delay       ; wait 1ms
                LCD_E_HI          ; E hi
                rcall Delay
                in r16, PINC      ; read portc
                LCD_E_LO          ; make e low
                sbrc r16, 7       ; check the busy flag in bit 7
                rjmp Loop_busy
                LCD_WRITE         ;
                LCD_RS_LO         ; rs lo
                pop r16
```

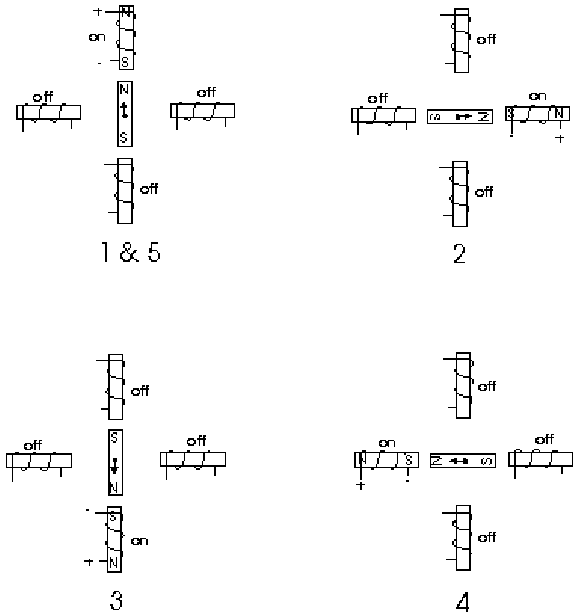
```
                ret

; write char in r16 to LCD
Write_char:     ;rcall Check_busy
        push r17
                rcall Check_busy
                LCD_WRITE
                LCD_RS_HI
                ser r17
                out ddrc, r17    ; c output
                out portc, R16
                LCD_E_HI
                LCD_E_LO
                clr r17
                out ddrc, r17
                ;rcall delay
                pop r17
                ret
;write instruction in r16 to LCD
Write_instruc:
        push r17
        rcall Check_busy
                LCD_WRITE
                LCD_RS_LO
                ser r17
                out ddrc, r17    ; c output
                out portc, R16
                ;rcall delay
                LCD_E_HI
        LCD_E_LO
                clr r17
                out ddrc, r17
        ;rcall delay
                pop r17
                ret


Init_LCD:       push r16
                clr r16
                out ddrc, r16
                out portc, r16
                sbi ddrb, 2      ;reg sel output
                sbi ddrb, 0      ; enable output
                sbi portb, 2
                sbi portb, 0
                sbi ddrb, 1      ; rw output
                ldi r16, 0x38
                rcall Write_instruc
                ldi r16, 0x0c
                rcall Write_instruc
                ldi r16, 0x06
                rcall Write_instruc
                ldi r16, 0x01
                rcall Write_instruc
        pop r16
                ret
```

3.10 Motors

Controlling a motor is one of the basics of interfacing. We will look briefly at three different types of motors.

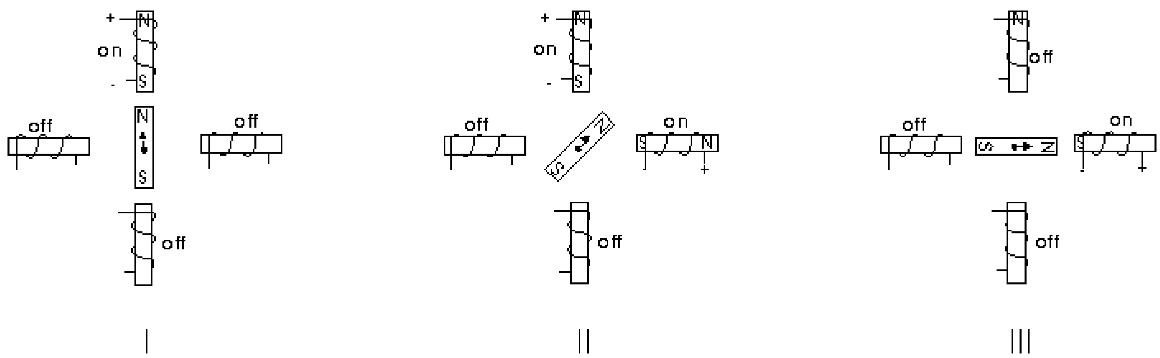
3.10.1 Stepper Motors

Stepper motors consist of a permanent magnet rotating shaft, called the rotor, and electromagnets on the stationary portion that surrounds the motor, called the stator. Figure 1 illustrates one complete rotation of a stepper motor. At position 1, we can see that the rotor is beginning at the upper electromagnet, which is currently active (has voltage applied to it). To move the rotor clockwise (CW), the upper electromagnet is deactivated and the right electromagnet is activated, causing the rotor to move 90 degrees CW, aligning itself with the active magnet. This process is repeated in the same manner at the south and west electromagnets until we once again reach the starting position.



n the above example, we used a motor with a resolution of 90 degrees or demonstration purposes. In reality, this would not be a very practical motor for most applications. The average stepper motor's resolution – the amount of degrees rotated per pulse – is much higher than this. For example, a motor with a resolution of 5 degrees would move its rotor 5 degrees per step, thereby requiring 72 pulses (steps) to complete a full 360 degree rotation.

You may double the resolution of some motors by a process known as "half-stepping". Instead of switching the next electromagnet in the rotation on one at a time, with half stepping you turn on both electromagnets, causing an equal attraction between, thereby doubling the resolution. As you can see in Figure 2, in the first position only the upper electromagnet is active, and the rotor is drawn completely to it. In position 2, both the top and right electromagnets are active, causing the rotor to position itself between the two active poles. Finally, in position 3, the top magnet is deactivated and the rotor is drawn all the way right. This process can then be repeated for the entire rotation.



3.10.2 DC Motors

DC Motors are the simplest to understand at a high level, you apply a voltage and the motor turns, the higher the voltage, the faster the motor turns. The speed of these can be controlled using PWM reasonably easily.

3.10.3 AC Motors

These are motors designed to run off a mains 220V AC supply. Turning them on and off with a triac is reasonably simple, PWM is considerably more tricky. Due to this and the inherent danger of 220V power we will not be looking at AC motors.

4 Usefull Bags of tricks

Some of the operations that you have always taken for granted are not that simple on a microcontroller, most notably the ability to do "complex" mathematics simply. There are also some usefull tricks that you have probably never needed before when coding.

4.1 Maths

The following code should be run in the simulator and the outputs noted, this provides a decent understanding of the 2's complement maths performed by the processor as well as the multiply instruction.

```
.include "m16def.inc"    ;Can you think why we need this

.def num1=R16            ;
.def num2=R17            ;
.def num3=R18
.def num4=R19
.def tmp1=r20
.def tmp2=r21
.def bignum1L=r30
.def bignum1H=r31
.def bignum2L=r28
.def bignum2H=r29

.cseg                    ;Tell the assembler that everything below this is in the
   code segment

.org $000                ;locate code at address $000
rjmp muls_8bit           ; Jump to the START Label

.org $02A                ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

; simple subtraction and illustration of 2's complement
        ldi num1, 0
        ldi num2, 5
        sub num2, num1 ; 5-0
        sub num1, num2 ; 0-5
        neg num1                 ; 2's complement of the result gives us 5 again
        nop

;16-bit addition and subtraction
        ldi bignum1L, 254        ;0b1111 1110
        ldi bignum1H, 3          ;0b0000 0011
        ;16-bit register holds the number 0x03fe =1022
        ; now add 10 to it
        adiw bignum1L, 10
        ; 16 bit register now holds 1032 = 0x0408
        nop
        ; now subtract 11
        sbiw bignum1L, 11
        ; 16 bit reg now holds 1021 = 0x03fd
        nop

;8bit unsigned multiplication
        ; let us now try 3 * 130
        ldi num1, 3              ;0x03
        ldi num2, 130    ;0x82
        mul num1, num2
```

```
        nop                                 ;0x0186

;8bit signed multiplication
        ; let us now see what happens when we use signed
        ; multiplication on those numbers
        ldi num1, 3             ;0x03
        ldi num2, 130    ;0x82
        muls num1, num2 ;0xfe86=65158
        ; this is because both numbers were treated as signed
        ; so what we had was 3 * -126 = -378, so if we take the 2's
        ; complement of this we will get 378
        ; 0x82 is the 2's complement of 126
        movw num1, r0
        com num1          ;1's complement
        com num2
        ldi tmp1, 1
        ldi tmp2, 0
        add num1, tmp1
        adc num2, tmp2
        ;=0x017a = 378
        nop



end_loop:
        nop
        nop
        rjmp end_loop
```

Adding and Subtracting

All Maths operations are done using twos complement maths. When only addition is used this is straight forward. Addition of positive and 2's complement numbers works the same as straight addition of unsigned binary numbers. You should note that when using 8-bit signed numbers, the largest representation is 128, not 255, this is the same as a signed and unsigned char in C.

Multiplication and Division

The ATMega provides a hardware multiplication of two 8-bit registers to provide a 16-bit result (in R0:1). Be carefull when using signed multiplication as both operands are treated as signed.

4.2 Lookup Tables

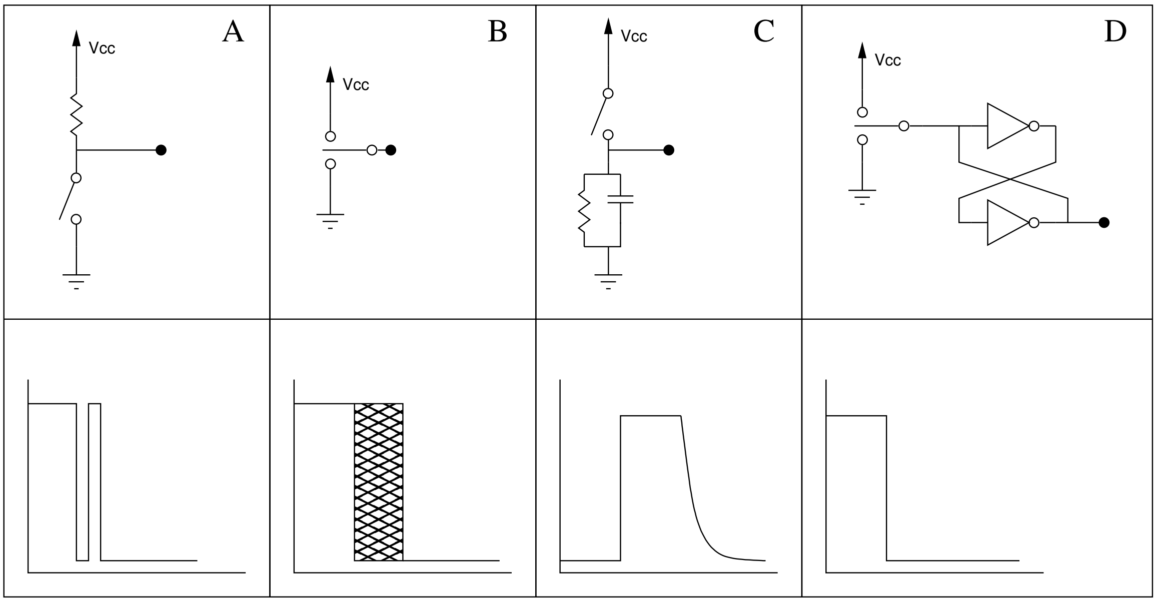
Lookup tables are an efficient means of returning a pre-determined value based on an input value. The input value is treated as an index to an array (offset from a memory location), the value to be returned for that offset is simply stored in the location that would be indicated.
This is one way of converting binary numbers to values for an output port with a seven segment display.
Consider the table below showing memory addresses and the contents.

Address	Offset	Contents
0x0340	0	0
0x0341	1	1
0x0342	2	4
0x0343	3	9
0x0344	4	16
0x0345	5	25
0x0346	6	36
0x0347	7	49
0x0348	8	64
0x0349	9	81
0x034A	10	100
0x034B	11	121
0x034C	12	144
0x034D	13	169
0x034E	14	196
0x034F	15	225

This table could be used to lookup the square of a number (this would only really be usefull if there was no multiply instruction available). The input is just the offset from the beginning of the table and the contents of the memory location given by the base address (0x0340) + offset would give the square of the offset.

4.3 Interfacing to a switch

This is often overlooked by many people not used to dealing with electronics. The simple action of pressing a switch down does not always produce a clear signal for a micro controller. Consider the circuits below:



- Circuit A: This circuit at first glance seems perfectly good, except for the fact that switches are not perfect and the moving contact does not always make contact and remain in contact with the other side. The moving part may bounce. This means that the output voltage may fluctuate between 0V and V_{CC} a few times before settling. The microcrontroller can see this as multiple presses.

- Circuit B: This would be the next choice for most people. Surely the voltage cannot fluctuate if the switch has not yet made contact with the other side? Unfortunately those people who study electronics should (hopefully) be able to point out that if the input port is the GATE of a MOS-based transistor and no-pull resistors are activated, then the gate voltage will float and can have any value present.

- Circuit C: This is the first solution that can work effectively. The capacitor is "instantly" charged when the switch makes contact, and will only slowly discharge (through the resistor) when the switch breaks contact. Thus the output voltage will be a sharp step up and then present a decaying exponential voltage. Through correct selection of component values you can ensure that the output voltage will not decay to the logic threshold voltage in the times that the switch may be bouncing. This circuit does however suffer from the drawback that the high to low transition is not instantaneous, which may cause trouble if the timing is critical.

- Circuit D: This is the ideal way of debouncing switched in hardware. The two logic inverters effectively form one bit of memory. The input state is remembered until it is asserted one way or the other. The only down side to this is that an extra logic chip is required in the design.

The alternative to this is to "debounce" the switch in software, by instituting a delay to avoid detecting possible bounces. This requires no hardware, but can sometimes cause code to become cumbersome.

4.4 Power Management and Sleep Modes

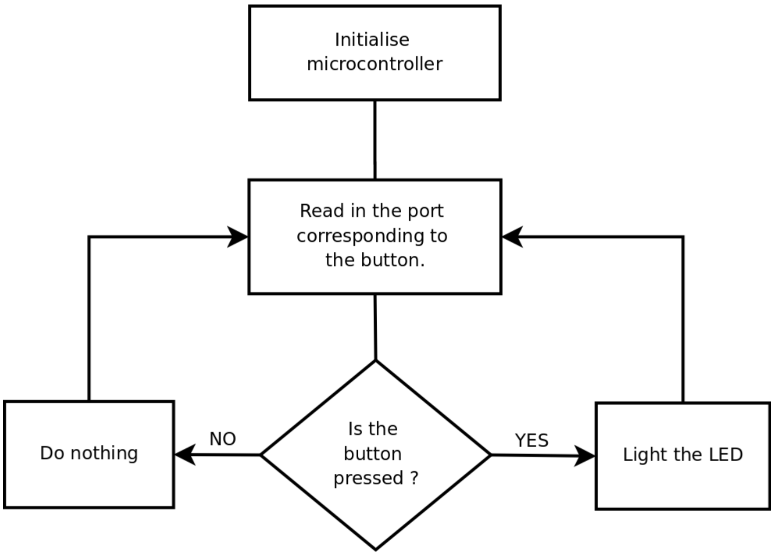
`Read pages 32-36 of the datasheet.`

5 Solving Some Real-World Problems

We will introduce the ATMega a little bit at a time, by solving "problems" using it. Some of these problems do appear manufactured - but they are there to illustrate a particular aspect of the microcontroller.
The source code and a .hex file will be made available as we cover this in class.

5.1 A glorified switch

The problem: We would like an led to light up whenever a switch is held down.
The solution: As this is our first (and rather simple program) the solution is rather simple. The diagram below shows the basic operation of the program.



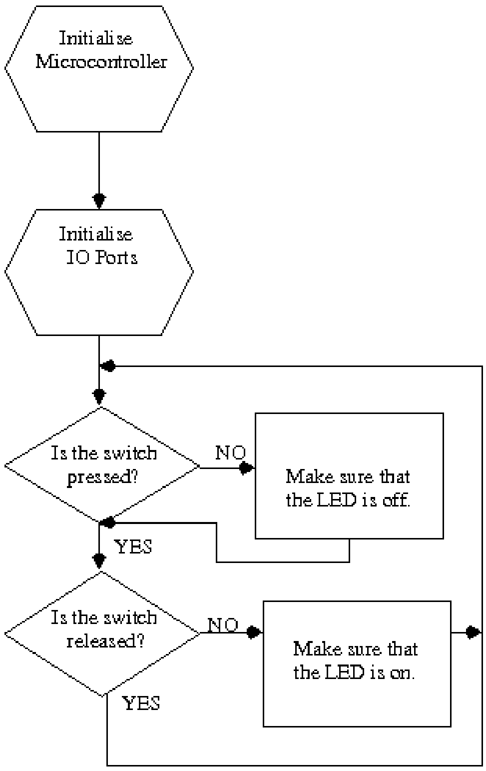
Conceptual flow of the program.

Now we look at a code solution to this problem.

```
;LED–SWITCH − Project1
; This program toggles portB pin 0
; Remember to patch PortB0 to an LED
; and PORTD0 to Switch0
.include "m16def.inc"    ;Can you think why we need this

.def TMP1=R16            ;defines serve the same purpose as in C,
.def TMP2=R17            ;before assembly, defined values are substituted
.def TMP3=R18

.cseg                    ;Tell the assembler that everything below this is in the
    code segment

.org $000                ;locate code at address $000
rjmp START               ; Jump to the START Label

.org $02A                ;locate code past the interupt vectors

START:   ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
         out SPL, TMP1
         ldi TMP1, HIGH(RAMEND)
         out SPH, TMP1

         RCALL INITIALISE ; Call the subroutine INITIALISE

MAIN_LOOP:
         ; Due to instructions we deviate slightly from
         ;        our flow diagram
```

```
        SBIC PIND, 0      ;skip next instruction if d:0 is low
        CBI  PORTB, 0     ;if d:0 is high (nopress) make b:0 low (off)
        SBIS PIND, 0      ;skip if d:0 set (no press)
        SBI      PORTB, 0        ; if pressed take b:0 high (led on)

        NOP
        NOP
        RJMP MAIN_LOOP






INITIALISE:
        SBI DDRB, 0                       ; Make pin0 of portB output
        CBI PORTB, 0              ; Set the initial state of pin to low=LED-off

        ;Strictly speaking the port value should be set
        ;  before making it an output, since this way
        ;       the LED will go on for the time it takes to
        ;       execute the next instruction.

        CBI DDRD, 0       ;make D:0 an input
        SBI PORTD, 0      ; enable pull-up on D:0
        RET                                       ;Return from subroutine
```

You should note that we do not follow the flow diagram given (it would actually make the code longer). The flow diagram for the code above is actually:



Tasks:

1. Modify (or rewrite) the code so that switches 0-2 operate LEDs 0-2 in the same manner. i.e. when a button is pressed the corresponding LED lights up.
2. Now change your code so that when switch 3 is pressed LED 3 will only light up if switch two is pressed as well.

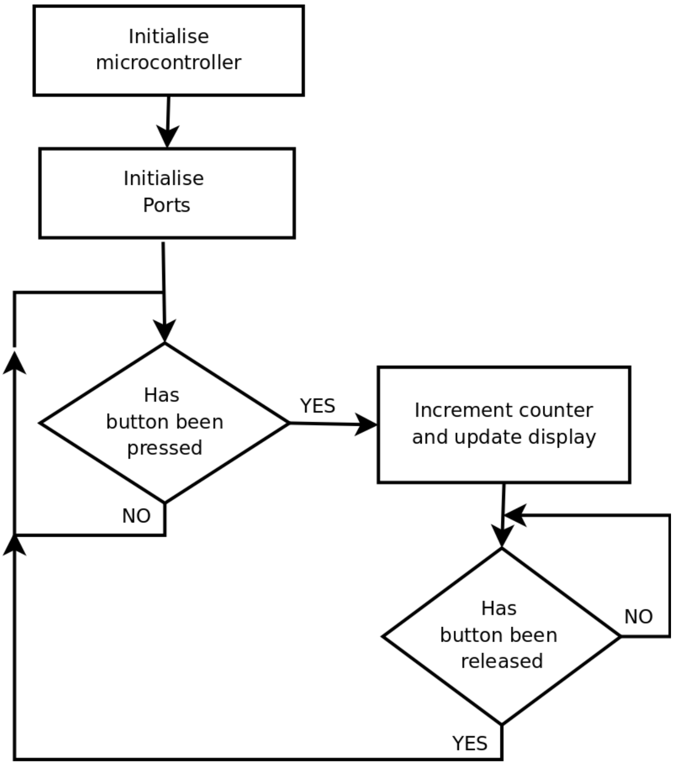
5.2 Counting Button Presses

The problem: We would like to count the number of times that a switch is pressed (and display the result).

The solution: The first problem that we encounter here, is that of switch bounce. (See Usefull Bags of tricks). Let us actually assume perfect switches at first and we will (hopefully) see the effect of switch bounce. To detect a switch press we have to observe the pin chnage from a logical 1 to a logical 0, so we cannot simply assume that every time we observe a 0 on the pin that it corresponds to a press, because the microcontroller runs orders of magnitude faster than our fastest finger press.

We are then going to display the count of presses on the LEDs as a binary number. (LEDx on represents 2^x.

The flow diagram for the code is:



The code for this is:

```
;button_count
; This program counts button presses on
; portd0 and displays them on portb leds
; Don't forget to connect portb0-7 to leds 0-7
; and connect portd0 to switch 0
.include "m16def.inc"    ;Can you think why we need this

.def TMP1=R16            ;
.def TMP2=R17            ;
.def TMP3=R18
.def COUNT=R19   ;store the count in this register

.cseg                    ;Tell the assembler that everything below this is in the
    code segment

.org $000                ;locate code at address $000
rjmp START               ; Jump to the START Label

.org $02A                ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
```

```
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

        RCALL INITIALISE

SAMPLE_DOWN:
        SBIC PIND,0   ; skip if button pushed
        RJMP SAMPLE_DOWN ;
        ;If we get here the button has been pushed
        INC COUNT         ;increment the counter;
        MOV TMP1, COUNT
        OUT PORTB, TMP1
SAMPLE_UP:
        SBIS PIND, 0 ; skip if button released
        RJMP SAMPLE_UP
        RJMP SAMPLE_DOWN

INITIALISE:
        ; Setup port b as output for LEDs
        ; and inital state to all off and set counter to zero
        SER TMP1;
        OUT DDRB, TMP1
    CLR TMP1
        OUT PORTB, TMP1 ; 0x00 = all off
        CLR COUNT
        OUT DDRD, TMP1 ; make d inputs (tmp1 is a handy clear register)
    SER TMP1
        OUT PORTD, TMP1 ; activate pullups on PORTD
        RET
```

Tasks:

1. Modify (or rewrite) the code so that switch 0 increments the counter and switch 1 decrements the counter. Only register a button press if the previous button has been released.

2. Modify your code so that instead of displaying the count, pressing a button moves a lit LED up or down the display. (Start with LED0 lit).

5.3 Counting button presses with an interrupt

We are now going to implement our last problem using interrupts, instead of keeping the processor busy needlessly sampling the buttons.

The problem: We would like to count the number of times that a switch is pressed (and display the result) using interrupts, int0 is going to be used to increment the counter on a press (switch 0) and int1 is going to be used to decrement the counter on a button release (switch 1).

The solution: To do this we must make INT0 trigger on a falling edge and INT1 trigger on a rising edge. We can then write separate interrupt routines to handle each case (increment and decrement). We must also be carefull not to "damage" data that may be in use in the main program loop.
The solution code is:

```
;button_count
; This program counts button presses on
; portb0 and displays them on portd leds
; using interrupts

.include "m16def.inc"    ;Can you think why we need this

.def TMP1=R16            ;
.def TMP2=R17            ;
.def TMP3=R18
.def COUNT=R19   ;store the count in this register

.cseg                    ;Tell the assembler that everything below this is in the
    code segment

.org 0x000               ;locate code at address $000
rjmp START               ; Jump to the START Label
.org INT0addr
rjmp INT0_ROUTINE
.org INT1addr
rjmp INT1_ROUTINE

.org $02A                ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

        RCALL INITIALISE
        SEI
MAIN_LOOP:
        NOP
        NOP
        NOP
        RJMP MAIN_LOOP

INITIALISE:
        ; Setup port b as output for LEDs
        ; and inital state to all off and set counter to zero
        CLR TMP1;
        OUT DDRD, TMP1   ; ensure that portdis an input
        SBI PORTD,2              ; enable pull-ups on int0
        SBI PORTD,3              ; enable pull-ups on int1
        CLR TMP1
        OUT PORTB, TMP1 ; port B all low, so leds off
    SER TMP1
        OUT DDRB, TMP1   ; portb output
        CLR COUNT                ; clear the counter
```

```
        LDI TMP1, 0x0e   ; int0 falling , int1 rising
        OUT     MCUCR, TMP1
        LDI TMP1, 0xc0
        OUT GICR, TMP1
        RET

INT0_ROUTINE:
;int 0 is to increment the counter
        PUSH TMP1                ;save tmp1
        IN TMP1, SREG
        PUSH TMP1                ; save sreg
        INC COUNT                ; decrement the counter
        MOV TMP1, COUNT
        OUT PORTB, TMP1 ; output
        POP TMP1
        OUT SREG, TMP1  ;resotre sreg
        POP TMP1                 ;resotr tmp1
        RETI

INT1_ROUTINE:
;int 1 is to decrement the counter
        PUSH TMP1
        IN TMP1, SREG
        PUSH TMP1
        DEC COUNT
        MOV TMP1, COUNT
        COM TMP1
        OUT PORTB, TMP1
        POP TMP1
        OUT SREG, TMP1
        POP TMP1
        RETI
```

Tasks:

1. Alter the program so that pressing button0 increments the count on a rising and falling edge.
2. Alter the program so that when button2 (connected to portd4) is pressed it de-activates button1. When pressed again it re-activates button1.

5.4 Using timers to time events

To introduce timers we are going to look at a reaction timer, that times how fast you can push a button after an LED has been lit.
The current program just displays the high byte of counter 1 (16-bit counter) on the LEDs.

```
; reaction timer
; switch0 - int0 on pd2
; switch1 - int1 on pd3
; portb0-7 to leds 0-7

.include "m16def.inc"

.def TMP1=R16              ;
.def TMP2=R17              ;
.def TMP3=R18

.cseg
.org $000                  ;locate code at address $000
rjmp START                 ; Jump to the START Label
.org INT0addr
rjmp INT0_ISR
.org INT1addr
rjmp INT1_ISR
.org OVF1addr
rjmp TIMER1_OVF_ISR
.org $02A                  ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

        RCALL INITIALISE_PORTS
        RCALL INITIALISE_TIMER
        RCALL INITIALISE_EXTERNAL_INTERRUPTS
        SEI
MAIN_LOOP:
        NOP
        NOP
        RJMP MAIN_LOOP

INITIALISE_PORTS:
        ; portb output LEDs
        LDI TMP1, 0x00
        out PORTA, tmp1
        LDI TMP1, 0xFF
        OUT DDRA, TMP1
        ;portd inputs
        LDI TMP1, 0x00
        OUT DDRD, TMP1
        ;enable pullups on int pins
        sbi PORTD, 2
        sbi PORTD, 3
        RET

INITIALISE_TIMER:
        ;enable timer 1 interrupt
        clr tmp1
        ldi tmp1, 0x04
        out TIMSK, tmp1
        clr tmp1
        ;set initial value in timer
        out TCNT1H, tmp1
        out TCNT1L, tmp1
        RET
```

```
INITIALISE_EXTERNAL_INTERRUPTS:
        ;enable  int0
        ldi tmp1, 0x40
        out GICR, tmp1
        ; interrupt on falling edge.
        ldi tmp1, 0x0a
        out MCUCR, tmp1
        ret

INT0_ISR:
        ;wait a while and then light the test led
;disable int 0 and enable int1
        ldi tmp1, 0x80
        out GICR, tmp1
        sbi porta,0
    ldi tmp1, 0x60
loop1: ser tmp2
loop2: ser tmp3
loop3: ;
        dec tmp3
        cpi tmp3, 0
        BRNE loop3
        dec tmp2
        BRNE loop2
        dec tmp1
        BRNE loop1
        ;we have now finished a bit of a delay.
        sbi porta, 1 ;turn led on
        cbi porta,0
        ;start timer going
        ldi tmp1, 0x05
        out tccr1b, tmp1
        reti

INT1_ISR:
; stop timer1
        ldi tmp1, 0x00
        out tccr1b, tmp1
;disable int 1 and enable int0
        ldi tmp1, 0x40
        out GICR, tmp1
;read both timer bytes and display (we will only display the high byte}
    IN tmp2, TCNT1L
        IN tmp1, TCNT1H
        out PORTA,tmp1
        reti

TIMER1_OVF_ISR:
; too slow
ldi tmp1, 0x00
        out tccr1b, tmp1
        out PORTA, tmp1
        reti
```

Tasks:

1. Change the code so that the number of whole seconds that have elapsed are displayed in binary on the led display. i.e. 0 to 255 seconds.

5.5 Making *Random* Dice

"Random" is a word thrown around quite often in computing, but any statistician or cryptographer (and hopefully computer scientist) will tell you that a truely random number is an elusive beast. That being said we are going to create an "apparently" random number.

Tasks:

1. Write code that initialises an 8-bit timer/counter and starts it counting with no pre-scaling.
2. On a button press, read in the value of the counter and output it as a value between 0 and 9 on the led display.
3. Satisfy yourself that the number displayed is "Random".

5.6 Using the Stepper Motor - 1

We want to make the stepper motor turn at a variable rate.

Consider the code below:

```
; Stepper Motor 1
; PD2 - SW0
; PD3 - SW1
; PD4 - DRV0
; ..
; PD7 - DRV3
; Also make sure that the jumper by the motor
; Connection is set to 2-3 (+5v)


.include "m16def.inc"

.def TMP1=R16            ;
.def TMP2=R17            ;
.def TMP3=R18
.def mask=R19
.cseg
.org $000                ;locate code at address $000
rjmp START               ; Jump to the START Label
.org INT0addr
rjmp INT0_ISR
.org INT1addr
rjmp INT1_ISR
.org OVF0addr
rjmp T0_OVF_ISR
.org $02A                ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

  ; Set PORT D to correct IO
  ldi TMP1, 0b11110000
  OUT DDRD, TMP1
  ; Set initial state on port D
  LDI TMP1, 0b00001100
  OUT PORTD, TMP1
  ; setup timer 1 for overflow
  LDI TMP1, 0x00
  out TCNT0, tmp1
  ldi tmp1, 0x01
  out TIMSK, tmp1
  ldi tmp1, 0x05
  out TCCR0, tmp1
  ldi MASK, 0b00010000
  out portd, mask
  SEI

loop:
  nop
  nop
  nop
  rjmp loop



INT0_ISR:
        RETI

INT1_ISR:
```

```
        RETI

T0_OVF_ISR:
        ; single step through pd4-7
    LSL MASK
;inc mask
    BREQ SET_PD4
    RJMP do_out
set_pd4:
    LDI MASK, 0b00010000
do_out:
    in TMP1, PORTD;
        andi tmp1, 0x0f
        or tmp1, mask
        out PORTD, mask
        RETI
```

Tasks:

1. Use the code as a starting point and alter it so that pressing SW0 (connected to INT0) will double the speed on each press up to some maximum.
2. Pressing SW1 (connected to INT1) will halve the speed on each press.

HINT: Read over what the Output Compare function on the timers can do.

5.7 Reading values in from the ADC

This code shows how to read data in from the ADC in free-running mode. The output is shown on the PORTB LEDs.

```
; Portc0-7 connected to leds0-7
; ADC0 - VAR

.include "m16def.inc"     ;Can you think why we need this

.def TMP1=R16             ;defines serve the same purpose as in C,
.def TMP2=R17             ;before assembly, defined values are substituted
.def TMP3=R18

.cseg                     ;Tell the assembler that everything below this is in the
   code segment

.org $000                 ;locate code at address $000
rjmp START                ; Jump to the START Label
.org ADCCaddr
rjmp ADC_ISR
.org $02A                 ;locate code past the interupt vectors

START:  ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

;setup ADC in free running mode, with interrupt
        ldi tmp1, 0b01100000
        out ADMUX, tmp1
        ldi tmp1, 0xff
        out adcsra, tmp1

; portc output
        ldi tmp1, 0xff
        out portc, tmp1
        out ddrc, tmp1

;enable interrupts
        SEI

main_loop:
        nop
        nop
        rjmp main_loop

ADC_ISR:
        push tmp1
        in tmp1, ADCH
        out portc, tmp1
        pop tmp1
        RETI
```

Tasks:

1. Alter the code so that the ADC operates in single conversion mode, with the conversion being triggered by the button linked to the external interrupt INT0.
2. Read all 10bits in, but ignore the two least significant bits.
3. Display your output on the PORTB LEDs, but not as a binary number. Display the output as a bargraph, with all LEDs on representing the value 0xff and no LEDs on representing the value 0x00 and 4 LEDs on representing the value 0x7F.

5.8 Using the stepper Motor 2

Now we are going to look at using a look-up table for controlling the stepper motor.

Tasks:

1. Set up a lookup table in EEPROM (or program memory is you are feeling adventurous) to provide the half-steps necessary to control the stepper motor.
2. Read the lookup table into RAM on start-up
3. Pressing SW0 should change direction.
4. Pressing SW1 should start/stop the motor.
5. Use the potentiometer connected to ADC0 to control the speed.

5.9 Using the LCD

The problem: Now the time comes to do a slightly more familiar "Hello World" program.

```
; LCD
;Portc0-7 connected to LCD_data0-7
;PB0 - E
;PB1 - rw
;PB2 - rs

.include "m16def.inc"



.def TMP1=R16           ;
.def TMP2=R17           ;
.def TMP3=R18

.cseg
.org $000               ;locate code at address $000
rjmp START              ; Jump to the START Label

.org $02A               ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)   ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1
        call Init_LCD
        ldi r16, 0x41
        call Write_char
        ldi r16, 0x84
        call Write_instruc
        rcall delay
        ldi r16, 0x7e
        call Write_char
;       ldi r16, 0x55
;       call Write_char

MAIN_LOOP:
        NOP
        NOP
        RJMP MAIN_LOOP

.include "LCD.asm"
```

Tasks:

1. Use the code above as a starting point for a program that prints "Hello World" on the LCD.
2. Generalise this to a program that writes a message, stored in EEPROM, to the LCD when button 1 is pressed.
3. You can pre-define a message (null terminated) in EEPROM by using .org, .db and .eseg.

5.10 A Real-Time Clock

We are going to use Timer2 and a 32kHz crystal to generate a 1 second interrupt for the purposes of time-keeping. The code below, turns LED0 on for one second and then off for one second.

```
; real time clock
; 32khz xtal between PC6 PC7
; PB0 - LED0
.include "m16def.inc"

.def TMP1=R16           ;
.def TMP2=R17           ;
.def TMP3=R18

.cseg
.org $000               ;locate code at address $000
rjmp START              ; Jump to the START Label
.org OVF2addr
rjmp TIMER2_OVF_ISR
.org $02A               ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1
        sbi ddrb, 0 ; set bit 0 to output
        cbi portb, 0; initially on
        RCALL INITIALISE_TIMER

        SEI
MAIN_LOOP:
        NOP
        NOP
        RJMP MAIN_LOOP

INITIALISE_TIMER:
        ;use tcnt2 for external pins
        ldi tmp1, 0x08
        out assr, tmp1 ;clock from external source
;       ldi Tmp1, 0x00
;       out TCNT2, Tmp2
        ldi Tmp1, 0b00000101
        out TCCR2, Tmp1 ; divide by 128
        ldi Tmp1, 0b01000000
        out tifr, tmp1;
        out TIMSK, Tmp1 ; enable overflow interrupt

        RET

TIMER2_OVF_ISR:
        ;flip portb0
        in tmp1, portb
        ldi tmp2, 0x01
        eor tmp1, tmp2
        out portb, tmp1
        reti
```

Tasks:

1. Write a program that effectively counts the number of seconds since the code started execution. Display this as a binary number on the PORTB LEDs.

5.11 Using The USART

We are going to use the USART to send data to the PC and recieve data from the PC. (Semember to connect the serial cable provided between the board and your PC.) The code below sets up the USART for reception of data and then on receiving a character, sends a short (polite) message to the PC.

```
; USART
; Connect RX and TX from usb-serial converter to TX and RX on board
; PB0 to led0
.include "m16def.inc"

.def TMP1=R16             ;
.def TMP2=R17             ;
.def TMP3=R18
.def MESSAGE_offset=r19
.dseg
MESSAGE: .byte 10 ; reserve 10bytes for the message
.cseg
.org $000                 ;locate code at address $000
rjmp START                ; Jump to the START Label
.org URXCaddr
rjmp URXC_ISR
.org UDREaddr
rjmp UDRE_ISR
.org UTXCaddr
rjmp UTXC_ISR

.org $02A                 ;locate code past the interupt vectors

START:
        ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1
        call Init_UART
        ;set portb output for LEDs
        ser Tmp1
        out ddrb, Tmp1
        out portb, Tmp1
        SEI
MAIN_LOOP:
        NOP
        NOP
        RJMP MAIN_LOOP

Init_UART:
;set baud rate (9600,8,n,2)
        ldi Tmp1, 51
        ldi Tmp2, 0x00
        out UBRRH, Tmp2
        out     UBRRL, Tmp1
;set rx and tx enable
        sbi UCSRB, RXEN
        sbi UCSRB, TXEN
; enable uart interrupts
        sbi UCSRB, RXCIE

        RET

; Interrupt code for when UDR empty
UDRE_ISR:
        RETI
; Code for TX complete
UTXC_ISR:
;increment message offset
```

```
        inc MESSAGE_OFFSET
;setup RAM pointer to the variable message
        LDI R30, low(MESSAGE)
        LDI R31, high(MESSAGE)
; increase by message offset.
        ADD R30, MESSAGE_OFFSET
        BRCC SEND_NEXT
        inc R31 ; there was an overflow, so increment the high byte
SEND_NEXT:
        LD Tmp1, Z
        cpi tmp1, 0x00
        breq message_finished
        out UDR, Tmp1
        RETI
message_finished:
        ;reenable rx
        sbi UCSRB, RXCIE
        ;clear led
        sbi portb, 0
        reti
; Code for RX complete
URXC_ISR:
        ; led 0 off
        in tmp1, udr
        cbi portb, 0
        call SEND_MESSAGE
        RETI

SEND_MESSAGE:
        clr MESSAGE_OFFSET;
        LDI R30, low(MESSAGE)
        LDI R31, high(MESSAGE)
        LDI Tmp1, 'H'
        ST Z+, Tmp1
        LDI Tmp1, 'e'
        ST Z+, Tmp1
        LDI Tmp1, 'l'
        ST Z+, Tmp1
        LDI Tmp1, 'l'
        ST Z+, Tmp1
        LDI Tmp1, 'o'
        ST Z+, Tmp1
        LDI Tmp1, 0x00
        ST Z+, Tmp1
        LDI R30, low(MESSAGE)
        LDI R31, high(MESSAGE)
        LD Tmp1, Z
        out UDR, Tmp1 ; tx first char
        SBI UCSRB, TXCIE ; enable txci
        cbi UCSRB, RXCIE ; disable reception
        RET
```

Tasks:

1. Store all messages in EEPROM.
2. Alter the code so that when a digit is received, the response is the English word for that digit. When any other character is received, the response 'Goodbye' is sent.
3. If any other character is received (not a number) then LED 1 should be flashed with a frequency of 1Hz. Until a digit is received.
4. Remember to use null terminated strings in RAM and EEPROM. (It is good programming practice.)

5.12 Controlling the Brightness of an LED

The code below shows how to set timer/counter 0 up as a pulse width modulator.

```
; Connect pb2 to led0
; pb3 (OC0) to led1

.include "m16def.inc"    ;Can you think why we need this

.def TMP1=R16            ;defines serve the same purpose as in C,
.def TMP2=R17            ;before assembly, defined values are substituted
.def TMP3=R18

.cseg                    ;Tell the assembler that everything below this is in the
   code segment

.org $000                ;locate code at address $000
rjmp START               ; Jump to the START Label

.org $02A                ;locate code past the interupt vectors

START:  ldi TMP1, LOW(RAMEND)    ;initialise the stack pointer
        out SPL, TMP1
        ldi TMP1, HIGH(RAMEND)
        out SPH, TMP1

        ldi tmp1, 0b00001100
        sbi PORTB, 2
        out ddrb, tmp1 ;set OC0 output
        ldi tmp1, 0xea
        out ocr0, tmp1  ; set the compare value
        ldi tmp1, 0b01110001
        out tccr0, tmp1 ; set tcnt0 operation

main_loop:
        nop
        nop
        rjmp main_loop
```

<u>**Tasks:**</u>

1. Alter the code so that a value from the ADC is continuously read in and used as the compare value for the counter so that changing the applied voltage to PORTA:0 changes the brightness of the LED on the OC0 pin.

2. Add to your code so that the LED on PORTB:2 behaves exactly oppositely to the LED on OC0. i.e. when the LED on OC0 gets brighter, LED0 gets dimmer.

5.13 Making a self controlled light dimmer.

We are going to make a self controlled light dimmer that will turn off the lights as the environment gets brighter.

Tasks:

1. Write a program that reads in a value from the ADC0 pin that has a LDR connected to it. As the LDR picks up more light LED0 should be made dimmer. Use your finger to block the LDR to get the ADC value that corresponds to the LED being completely on. Use the unobstructed ambient lighting to give a value for when the LED should be totally off.

Appendices

ASCII Table

The ASCII Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	`
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	'	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

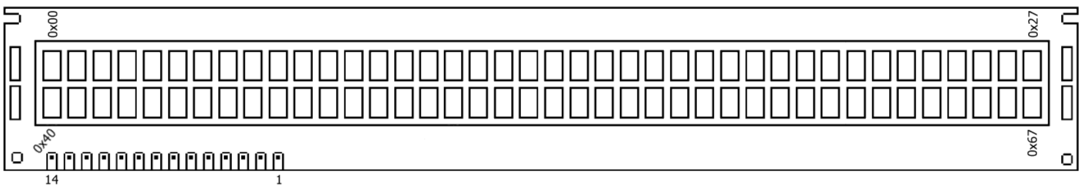
The *Extended Concise* LCD Data Sheet for HD44780

Version: 25.6.1999

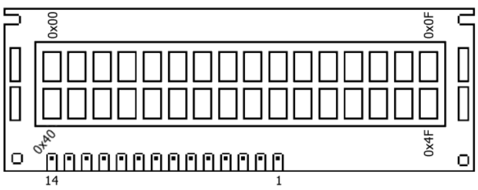
Instruction	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0	Description	Clock-Cycles
NOP	0	0	0	0	0	0	0	0	0	0	No Operation	0
Clear Display	0	0	0	0	0	0	0	0	0	1	Clear display & set address counter to zero	165
Cursor Home	0	0	0	0	0	0	0	0	1	x	Set adress counter to zero, return shifted display to original position. DD RAM contents remains unchanged.	3
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Set cursor move direction (I/D) and specify automatic display shift (S).	3
Display Control	0	0	0	0	0	0	1	D	C	B	Turn display (D), cursor on/off (C), and cursor blinking (B).	3
Cursor / Display shift	0	0	0	0	0	1	S/C	R/L	x	x	Shift display or move cursor (S/C) and specify direction (R/L).	3
Function Set	0	0	0	0	1	DL	N	F	x	x	Set interface data width (DL), number of display lines (N) and character font (F).	3
Set CGRAM Address	0	0	0	1	CGRAM Address						Set CGRAM address. CGRAM data is sent afterwards.	3
Set DDRAM Address	0	0	1	DDRAM Address							Set DDRAM address. DDRAM data is sent afterwards.	3
Busy Flag & Address	0	1	BF	Address Counter							Read busy flag (BF) and address counter	0
Write Data	1	0	Data								Write data into DDRAM or CGRAM	3
Read Data	1	1	Data								Read data from DDRAM or CGRAM	3

x : Don't care	I/D	1 0	Increment Decrement	R/L	1 0	Shift to the right Shift to the left
	S	1 0	Automatic display shift	DL	1 0	8 bit interface 4 bit interface
	D	1 0	Display ON Display OFF	N	1 0	2 lines 1 line
	C	1 0	Cursor ON Cursor OFF	F	1 0	5x10 dots 5x7 dots
	B	1 0	Cursor blinking	DDRAM : Display Data RAM CGRAM : Character Generator RAM		
	S/C	1 0	Display shift Cursor move			

LCD Display with 2 lines x 40 characters :



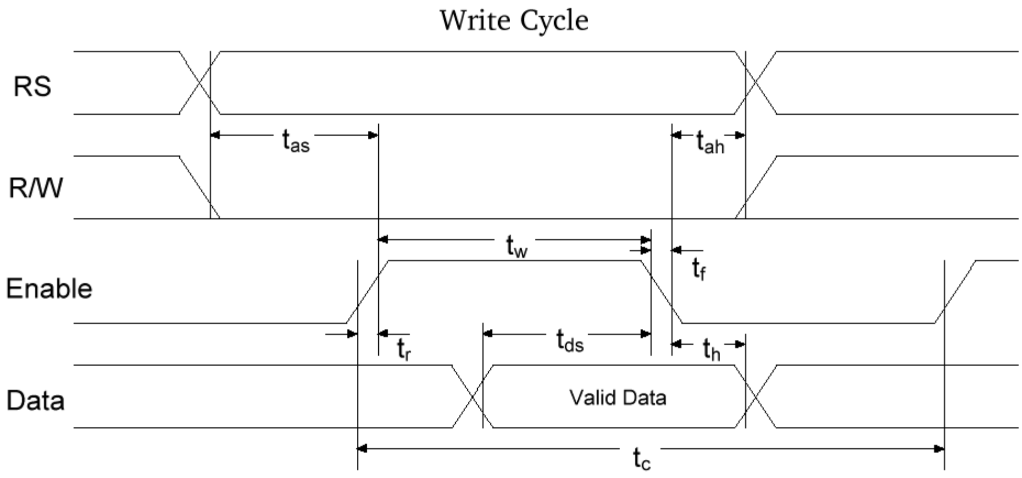
LCD Display with 2 lines x 16 characters :



Pin No	Name	Function	Description
1	Vss	Power	GND
2	Vdd	Power	+ 5 V
3	Vee	Contrast Adj.	(-2) 0 - 5 V
4	RS	Command	Register Select
5	R/W	Command	Read / Write
6	E	Command	Enable (Strobe)
7	D0	I/O	Data LSB
8	D1	I/O	Data
9	D2	I/O	Data
10	D3	I/O	Data
11	D4	I/O	Data
12	D5	I/O	Data
13	D6	I/O	Data
14	D7	I/O	Data MSB

Bus Timing Characteristics

(Ta = - 20 to + 75°C)



Write-Cycle	V_{DD}	2.7 - 4.5 V (2)	4.5 - 5.5 V (2)		2.7 - 4.5 V (2)	4.5 - 5.5 V (2)	
Parameter	**Symbol**	**Min(1)**		**Typ(1)**	**Max(1)**		**Unit**
Enable Cycle Time	t_c	1000	500	-	-	-	ns
Enable Pulse Width (High)	t_w	450	230	-	-	-	ns
Enable Rise/Fall Time	t_r, t_f	-	-	-	25	20	ns
Address Setup Time	t_{as}	60	40	-	-	-	ns
Address Hold Time	t_{ah}	20	10	-	-	-	ns
Data Setup Time	t_{ds}	195	80	-	-	-	ns
Data Hold Time	t_h	10	10	-	-	-	ns

(1) *The above specifications are indications only (based on Hitachi HD44780). Timing will vary from manufacturer to manufacturer.*

(2) *Power Supply : HD44780 S : V_{DD} = 4.5 - 5.5 V*
HD44780 U : V_{DD} = 2.7 - 5.5 V

This data sheet refers to specifications for the Hitachi HD44780 LCD Driver chip, which is used for most LCD modules.

Common types are :
1 line x 20 characters
2 lines x 16 characters
2 lines x 20 characters
2 lines x 40 characters
4 lines x 20 characters
4 lines x 40 characters

Mega16 include file

```
;***** THIS IS A MACHINE GENERATED FILE - DO NOT EDIT ********************
;***** Created: 2008-11-07 12:39 ******* Source: ATmega16.xml ************
;*************************************************************************
;* A P P L I C A T I O N   N O T E   F O R   T H E   A V R   F A M I L Y
;*
;* Number              : AVR000
;* File Name           : "m16def.inc"
;* Title               : Register/Bit Definitions for the ATmega16
;* Date                : 2008-11-07
;* Version             : 2.31
;* Support E-mail      : avr@atmel.com
;* Target MCU          : ATmega16
;*
;* DESCRIPTION
;* When including this file in the assembly program file, all I/O register
;* names and I/O register bit names appearing in the data book can be used.
;* In addition, the six registers forming the three data pointers X, Y and
;* Z have been assigned names XL - ZH. Highest RAM address for Internal
;* SRAM is also defined
;*
;* The Register names are represented by their hexadecimal address.
;*
;* The Register Bit names are represented by their bit number (0-7).
;*
;* Please observe the difference in using the bit names with instructions
;* such as "sbr"/"cbr" (set/clear bit in register) and "sbrs"/"sbrc"
;* (skip if bit in register set/cleared). The following example illustrates
;* this:
;*
;* in    r16,PORTB             ;read PORTB latch
;* sbr   r16,(1<<PB6)+(1<<PB5) ;set PB6 and PB5 (use masks, not bit#)
;* out   PORTB,r16             ;output to PORTB
;*
;* in    r16,TIFR              ;read the Timer Interrupt Flag Register
;* sbrc  r16,TOV0              ;test the overflow flag (use bit#)
;* rjmp  TOV0_is_set           ;jump if set
;* ...                         ;otherwise do something else
;*************************************************************************

#ifndef _M16DEF_INC_
#define _M16DEF_INC_


#pragma partinc 0

; ***** SPECIFY DEVICE ***************************************************
.device ATmega16
#pragma AVRPART ADMIN PART_NAME ATmega16
.equ    SIGNATURE_000   = 0x1e
.equ    SIGNATURE_001   = 0x94
.equ    SIGNATURE_002   = 0x03

#pragma AVRPART CORE CORE_VERSION V2E

; ***** I/O REGISTER DEFINITIONS *****************************************
; NOTE:
; Definitions marked "MEMORY MAPPED" are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ    SREG    = 0x3f
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
.equ    OCR0    = 0x3c
.equ    GICR    = 0x3b
.equ    GIFR    = 0x3a
```

```
.equ    TIMSK   = 0x39
.equ    TIFR    = 0x38
.equ    SPMCSR  = 0x37
.equ    TWCR    = 0x36
.equ    MCUCR   = 0x35
.equ    MCUCSR  = 0x34
.equ    TCCR0   = 0x33
.equ    TCNT0   = 0x32
.equ    OSCCAL  = 0x31
.equ    OCDR    = 0x31
.equ    SFIOR   = 0x30
.equ    TCCR1A  = 0x2f
.equ    TCCR1B  = 0x2e
.equ    TCNT1L  = 0x2c
.equ    TCNT1H  = 0x2d
.equ    OCR1AL  = 0x2a
.equ    OCR1AH  = 0x2b
.equ    OCR1BL  = 0x28
.equ    OCR1BH  = 0x29
.equ    ICR1L   = 0x26
.equ    ICR1H   = 0x27
.equ    TCCR2   = 0x25
.equ    TCNT2   = 0x24
.equ    OCR2    = 0x23
.equ    ASSR    = 0x22
.equ    WDTCR   = 0x21
.equ    UBRRH   = 0x20
.equ    UCSRC   = 0x20
.equ    EEARL   = 0x1e
.equ    EEARH   = 0x1f
.equ    EEDR    = 0x1d
.equ    EECR    = 0x1c
.equ    PORTA   = 0x1b
.equ    DDRA    = 0x1a
.equ    PINA    = 0x19
.equ    PORTB   = 0x18
.equ    DDRB    = 0x17
.equ    PINB    = 0x16
.equ    PORTC   = 0x15
.equ    DDRC    = 0x14
.equ    PINC    = 0x13
.equ    PORTD   = 0x12
.equ    DDRD    = 0x11
.equ    PIND    = 0x10
.equ    SPDR    = 0x0f
.equ    SPSR    = 0x0e
.equ    SPCR    = 0x0d
.equ    UDR     = 0x0c
.equ    UCSRA   = 0x0b
.equ    UCSRB   = 0x0a
.equ    UBRRL   = 0x09
.equ    ACSR    = 0x08
.equ    ADMUX   = 0x07
.equ    ADCSRA  = 0x06
.equ    ADCH    = 0x05
.equ    ADCL    = 0x04
.equ    TWDR    = 0x03
.equ    TWAR    = 0x02
.equ    TWSR    = 0x01
.equ    TWBR    = 0x00

; ***** BIT DEFINITIONS **************************************************
; ***** TIMER_COUNTER_0 **************
; TCCR0 - Timer/Counter Control Register
.equ    CS00    = 0     ; Clock Select 1
.equ    CS01    = 1     ; Clock Select 1
.equ    CS02    = 2     ; Clock Select 2
```

```
.equ    WGM01   = 3      ; Waveform Generation Mode 1
.equ    CTC0    = WGM01  ; For compatibility
.equ    COM00   = 4      ; Compare match Output Mode 0
.equ    COM01   = 5      ; Compare Match Output Mode 1
.equ    WGM00   = 6      ; Waveform Generation Mode 0
.equ    PWM0    = WGM00  ; For compatibility
.equ    FOC0    = 7      ; Force Output Compare

; TCNT0 - Timer/Counter Register
.equ    TCNT0_0 = 0      ;
.equ    TCNT0_1 = 1      ;
.equ    TCNT0_2 = 2      ;
.equ    TCNT0_3 = 3      ;
.equ    TCNT0_4 = 4      ;
.equ    TCNT0_5 = 5      ;
.equ    TCNT0_6 = 6      ;
.equ    TCNT0_7 = 7      ;

; OCR0 - Output Compare Register
.equ    OCR0_0  = 0      ;
.equ    OCR0_1  = 1      ;
.equ    OCR0_2  = 2      ;
.equ    OCR0_3  = 3      ;
.equ    OCR0_4  = 4      ;
.equ    OCR0_5  = 5      ;
.equ    OCR0_6  = 6      ;
.equ    OCR0_7  = 7      ;

; TIMSK - Timer/Counter Interrupt Mask Register
.equ    TOIE0   = 0      ; Timer/Counter0 Overflow Interrupt Enable
.equ    OCIE0   = 1      ; Timer/Counter0 Output Compare Match Interrupt register

; TIFR - Timer/Counter Interrupt Flag register
.equ    TOV0    = 0      ; Timer/Counter0 Overflow Flag
.equ    OCF0    = 1      ; Output Compare Flag 0

; SFIOR - Special Function IO Register
.equ    PSR10   = 0      ; Prescaler Reset Timer/Counter1 and Timer/Counter0

; ***** TIMER_COUNTER_1 **************
; TIMSK - Timer/Counter Interrupt Mask Register
.equ    TOIE1   = 2      ; Timer/Counter1 Overflow Interrupt Enable
.equ    OCIE1B  = 3      ; Timer/Counter1 Output CompareB Match Interrupt Enable
.equ    OCIE1A  = 4      ; Timer/Counter1 Output CompareA Match Interrupt Enable
.equ    TICIE1  = 5      ; Timer/Counter1 Input Capture Interrupt Enable

; TIFR - Timer/Counter Interrupt Flag register
.equ    TOV1    = 2      ; Timer/Counter1 Overflow Flag
.equ    OCF1B   = 3      ; Output Compare Flag 1B
.equ    OCF1A   = 4      ; Output Compare Flag 1A
.equ    ICF1    = 5      ; Input Capture Flag 1

; TCCR1A - Timer/Counter1 Control Register A
.equ    WGM10   = 0      ; Waveform Generation Mode
.equ    PWM10   = WGM10  ; For compatibility
.equ    WGM11   = 1      ; Waveform Generation Mode
.equ    PWM11   = WGM11  ; For compatibility
.equ    FOC1B   = 2      ; Force Output Compare 1B
.equ    FOC1A   = 3      ; Force Output Compare 1A
.equ    COM1B0  = 4      ; Compare Output Mode 1B, bit 0
.equ    COM1B1  = 5      ; Compare Output Mode 1B, bit 1
.equ    COM1A0  = 6      ; Compare Ouput Mode 1A, bit 0
.equ    COM1A1  = 7      ; Compare Output Mode 1A, bit 1

; TCCR1B - Timer/Counter1 Control Register B
.equ    CS10    = 0      ; Prescaler source of Timer/Counter 1
.equ    CS11    = 1      ; Prescaler source of Timer/Counter 1
```

```
.equ    CS12    = 2      ; Prescaler source of Timer/Counter 1
.equ    WGM12   = 3      ; Waveform Generation Mode
.equ    CTC10   = WGM12  ; For compatibility
.equ    CTC1    = WGM12  ; For compatibility
.equ    WGM13   = 4      ; Waveform Generation Mode
.equ    CTC11   = WGM13  ; For compatibility
.equ    ICES1   = 6      ; Input Capture 1 Edge Select
.equ    ICNC1   = 7      ; Input Capture 1 Noise Canceler

; ***** EXTERNAL_INTERRUPT ***********
; GICR - General Interrupt Control Register
.equ    GIMSK   = GICR   ; For compatibility
.equ    IVCE    = 0      ; Interrupt Vector Change Enable
.equ    IVSEL   = 1      ; Interrupt Vector Select
.equ    INT2    = 5      ; External Interrupt Request 2 Enable
.equ    INT0    = 6      ; External Interrupt Request 0 Enable
.equ    INT1    = 7      ; External Interrupt Request 1 Enable

; GIFR - General Interrupt Flag Register
.equ    INTF2   = 5      ; External Interrupt Flag 2
.equ    INTF0   = 6      ; External Interrupt Flag 0
.equ    INTF1   = 7      ; External Interrupt Flag 1

; MCUCR - General Interrupt Control Register
.equ    ISC00   = 0      ; Interrupt Sense Control 0 Bit 0
.equ    ISC01   = 1      ; Interrupt Sense Control 0 Bit 1
.equ    ISC10   = 2      ; Interrupt Sense Control 1 Bit 0
.equ    ISC11   = 3      ; Interrupt Sense Control 1 Bit 1

; MCUCSR - MCU Control And Status Register
.equ    ISC2    = 6      ; Interrupt Sense Control 2

; ***** EEPROM ***********************
; EEDR - EEPROM Data Register
.equ    EEDR0   = 0      ; EEPROM Data Register bit 0
.equ    EEDR1   = 1      ; EEPROM Data Register bit 1
.equ    EEDR2   = 2      ; EEPROM Data Register bit 2
.equ    EEDR3   = 3      ; EEPROM Data Register bit 3
.equ    EEDR4   = 4      ; EEPROM Data Register bit 4
.equ    EEDR5   = 5      ; EEPROM Data Register bit 5
.equ    EEDR6   = 6      ; EEPROM Data Register bit 6
.equ    EEDR7   = 7      ; EEPROM Data Register bit 7

; EECR - EEPROM Control Register
.equ    EERE    = 0      ; EEPROM Read Enable
.equ    EEWE    = 1      ; EEPROM Write Enable
.equ    EEMWE   = 2      ; EEPROM Master Write Enable
.equ    EEWEE   = EEMWE  ; For compatibility
.equ    EERIE   = 3      ; EEPROM Ready Interrupt Enable

; ***** CPU **************************
; SREG - Status Register
.equ    SREG_C  = 0      ; Carry Flag
.equ    SREG_Z  = 1      ; Zero Flag
.equ    SREG_N  = 2      ; Negative Flag
.equ    SREG_V  = 3      ; Two's Complement Overflow Flag
.equ    SREG_S  = 4      ; Sign Bit
.equ    SREG_H  = 5      ; Half Carry Flag
.equ    SREG_T  = 6      ; Bit Copy Storage
.equ    SREG_I  = 7      ; Global Interrupt Enable

; MCUCR - MCU Control Register
;.equ   ISC00   = 0      ; Interrupt Sense Control 0 Bit 0
;.equ   ISC01   = 1      ; Interrupt Sense Control 0 Bit 1
;.equ   ISC10   = 2      ; Interrupt Sense Control 1 Bit 0
;.equ   ISC11   = 3      ; Interrupt Sense Control 1 Bit 1
.equ    SM0     = 4      ; Sleep Mode Select
```

```
.equ    SM1     = 5       ; Sleep Mode Select
.equ    SE      = 6       ; Sleep Enable
.equ    SM2     = 7       ; Sleep Mode Select

; MCUCSR - MCU Control And Status Register
.equ    MCUSR   = MCUCSR          ; For compatibility
.equ    PORF    = 0       ; Power-on reset flag
.equ    EXTRF   = 1       ; External Reset Flag
.equ    EXTREF  = EXTRF   ; For compatibility
.equ    BORF    = 2       ; Brown-out Reset Flag
.equ    WDRF    = 3       ; Watchdog Reset Flag
.equ    JTRF    = 4       ; JTAG Reset Flag
.equ    JTD     = 7       ; JTAG Interface Disable

; OSCCAL - Oscillator Calibration Value
.equ    CAL0    = 0       ; Oscillator Calibration Value Bit0
.equ    CAL1    = 1       ; Oscillator Calibration Value Bit1
.equ    CAL2    = 2       ; Oscillator Calibration Value Bit2
.equ    CAL3    = 3       ; Oscillator Calibration Value Bit3
.equ    CAL4    = 4       ; Oscillator Calibration Value Bit4
.equ    CAL5    = 5       ; Oscillator Calibration Value Bit5
.equ    CAL6    = 6       ; Oscillator Calibration Value Bit6
.equ    CAL7    = 7       ; Oscillator Calibration Value Bit7

; SFIOR - Special function I/O register
;.equ   PSR10   = 0       ; Prescaler reset
.equ    PSR2    = 1       ; Prescaler reset
.equ    PUD     = 2       ; Pull-up Disable

; ***** TIMER_COUNTER_2 **************
; TIMSK - Timer/Counter Interrupt Mask register
.equ    TOIE2   = 6       ; Timer/Counter2 Overflow Interrupt Enable
.equ    OCIE2   = 7       ; Timer/Counter2 Output Compare Match Interrupt Enable

; TIFR - Timer/Counter Interrupt Flag Register
.equ    TOV2    = 6       ; Timer/Counter2 Overflow Flag
.equ    OCF2    = 7       ; Output Compare Flag 2

; TCCR2 - Timer/Counter2 Control Register
.equ    CS20    = 0       ; Clock Select bit 0
.equ    CS21    = 1       ; Clock Select bit 1
.equ    CS22    = 2       ; Clock Select bit 2
.equ    WGM21   = 3       ; Waveform Generation Mode
.equ    CTC2    = WGM21   ; For compatibility
.equ    COM20   = 4       ; Compare Output Mode bit 0
.equ    COM21   = 5       ; Compare Output Mode bit 1
.equ    WGM20   = 6       ; Waveform Genration Mode
.equ    PWM2    = WGM20   ; For compatibility
.equ    FOC2    = 7       ; Force Output Compare

; TCNT2 - Timer/Counter2
.equ    TCNT2_0 = 0       ; Timer/Counter 2 bit 0
.equ    TCNT2_1 = 1       ; Timer/Counter 2 bit 1
.equ    TCNT2_2 = 2       ; Timer/Counter 2 bit 2
.equ    TCNT2_3 = 3       ; Timer/Counter 2 bit 3
.equ    TCNT2_4 = 4       ; Timer/Counter 2 bit 4
.equ    TCNT2_5 = 5       ; Timer/Counter 2 bit 5
.equ    TCNT2_6 = 6       ; Timer/Counter 2 bit 6
.equ    TCNT2_7 = 7       ; Timer/Counter 2 bit 7

; OCR2 - Timer/Counter2 Output Compare Register
.equ    OCR2_0  = 0       ; Timer/Counter2 Output Compare Register Bit 0
.equ    OCR2_1  = 1       ; Timer/Counter2 Output Compare Register Bit 1
.equ    OCR2_2  = 2       ; Timer/Counter2 Output Compare Register Bit 2
.equ    OCR2_3  = 3       ; Timer/Counter2 Output Compare Register Bit 3
.equ    OCR2_4  = 4       ; Timer/Counter2 Output Compare Register Bit 4
.equ    OCR2_5  = 5       ; Timer/Counter2 Output Compare Register Bit 5
```

```
.equ    OCR2_6   = 6      ; Timer/Counter2 Output Compare Register Bit 6
.equ    OCR2_7   = 7      ; Timer/Counter2 Output Compare Register Bit 7

; ASSR - Asynchronous Status Register
.equ    TCR2UB   = 0      ; Timer/counter Control Register2 Update Busy
.equ    OCR2UB   = 1      ; Output Compare Register2 Update Busy
.equ    TCN2UB   = 2      ; Timer/Counter2 Update Busy
.equ    AS2      = 3      ; Asynchronous Timer/counter2

; SFIOR - Special Function IO Register
;.equ   PSR2     = 1      ; Prescaler Reset Timer/Counter2

; ***** SPI *************************
; SPDR - SPI Data Register
.equ    SPDR0    = 0      ; SPI Data Register bit 0
.equ    SPDR1    = 1      ; SPI Data Register bit 1
.equ    SPDR2    = 2      ; SPI Data Register bit 2
.equ    SPDR3    = 3      ; SPI Data Register bit 3
.equ    SPDR4    = 4      ; SPI Data Register bit 4
.equ    SPDR5    = 5      ; SPI Data Register bit 5
.equ    SPDR6    = 6      ; SPI Data Register bit 6
.equ    SPDR7    = 7      ; SPI Data Register bit 7

; SPSR - SPI Status Register
.equ    SPI2X    = 0      ; Double SPI Speed Bit
.equ    WCOL     = 6      ; Write Collision Flag
.equ    SPIF     = 7      ; SPI Interrupt Flag

; SPCR - SPI Control Register
.equ    SPR0     = 0      ; SPI Clock Rate Select 0
.equ    SPR1     = 1      ; SPI Clock Rate Select 1
.equ    CPHA     = 2      ; Clock Phase
.equ    CPOL     = 3      ; Clock polarity
.equ    MSTR     = 4      ; Master/Slave Select
.equ    DORD     = 5      ; Data Order
.equ    SPE      = 6      ; SPI Enable
.equ    SPIE     = 7      ; SPI Interrupt Enable

; ***** USART ***********************
; UDR - USART I/O Data Register
.equ    UDR0     = 0      ; USART I/O Data Register bit 0
.equ    UDR1     = 1      ; USART I/O Data Register bit 1
.equ    UDR2     = 2      ; USART I/O Data Register bit 2
.equ    UDR3     = 3      ; USART I/O Data Register bit 3
.equ    UDR4     = 4      ; USART I/O Data Register bit 4
.equ    UDR5     = 5      ; USART I/O Data Register bit 5
.equ    UDR6     = 6      ; USART I/O Data Register bit 6
.equ    UDR7     = 7      ; USART I/O Data Register bit 7

; UCSRA - USART Control and Status Register A
.equ    USR      = UCSRA  ; For compatibility
.equ    MPCM     = 0      ; Multi-processor Communication Mode
.equ    U2X      = 1      ; Double the USART transmission speed
.equ    UPE      = 2      ; Parity Error
.equ    PE       = UPE    ; For compatibility
.equ    DOR      = 3      ; Data overRun
.equ    FE       = 4      ; Framing Error
.equ    UDRE     = 5      ; USART Data Register Empty
.equ    TXC      = 6      ; USART Transmitt Complete
.equ    RXC      = 7      ; USART Receive Complete

; UCSRB - USART Control and Status Register B
.equ    UCR      = UCSRB  ; For compatibility
.equ    TXB8     = 0      ; Transmit Data Bit 8
.equ    RXB8     = 1      ; Receive Data Bit 8
.equ    UCSZ2    = 2      ; Character Size
.equ    CHR9     = UCSZ2  ; For compatibility
```

```
.equ    TXEN    = 3     ; Transmitter Enable
.equ    RXEN    = 4     ; Receiver Enable
.equ    UDRIE   = 5     ; USART Data register Empty Interrupt Enable
.equ    TXCIE   = 6     ; TX Complete Interrupt Enable
.equ    RXCIE   = 7     ; RX Complete Interrupt Enable

; UCSRC - USART Control and Status Register C
.equ    UCPOL   = 0     ; Clock Polarity
.equ    UCSZ0   = 1     ; Character Size
.equ    UCSZ1   = 2     ; Character Size
.equ    USBS    = 3     ; Stop Bit Select
.equ    UPM0    = 4     ; Parity Mode Bit 0
.equ    UPM1    = 5     ; Parity Mode Bit 1
.equ    UMSEL   = 6     ; USART Mode Select
.equ    URSEL   = 7     ; Register Select

.equ    UBRRHI  = UBRRH ; For compatibility

; ***** TWI **************************
; TWBR - TWI Bit Rate register
.equ    I2BR    = TWBR  ; For compatibility
.equ    TWBR0   = 0     ;
.equ    TWBR1   = 1     ;
.equ    TWBR2   = 2     ;
.equ    TWBR3   = 3     ;
.equ    TWBR4   = 4     ;
.equ    TWBR5   = 5     ;
.equ    TWBR6   = 6     ;
.equ    TWBR7   = 7     ;

; TWCR - TWI Control Register
.equ    I2CR    = TWCR  ; For compatibility
.equ    TWIE    = 0     ; TWI Interrupt Enable
.equ    I2IE    = TWIE  ; For compatibility
.equ    TWEN    = 2     ; TWI Enable Bit
.equ    I2EN    = TWEN  ; For compatibility
.equ    ENI2C   = TWEN  ; For compatibility
.equ    TWWC    = 3     ; TWI Write Collition Flag
.equ    I2WC    = TWWC  ; For compatibility
.equ    TWSTO   = 4     ; TWI Stop Condition Bit
.equ    I2STO   = TWSTO ; For compatibility
.equ    TWSTA   = 5     ; TWI Start Condition Bit
.equ    I2STA   = TWSTA ; For compatibility
.equ    TWEA    = 6     ; TWI Enable Acknowledge Bit
.equ    I2EA    = TWEA  ; For compatibility
.equ    TWINT   = 7     ; TWI Interrupt Flag
.equ    I2INT   = TWINT ; For compatibility

; TWSR - TWI Status Register
.equ    I2SR    = TWSR  ; For compatibility
.equ    TWPS0   = 0     ; TWI Prescaler
.equ    TWS0    = TWPS0 ; For compatibility
.equ    I2GCE   = TWPS0 ; For compatibility
.equ    TWPS1   = 1     ; TWI Prescaler
.equ    TWS1    = TWPS1 ; For compatibility
.equ    TWS3    = 3     ; TWI Status
.equ    I2S3    = TWS3  ; For compatibility
.equ    TWS4    = 4     ; TWI Status
.equ    I2S4    = TWS4  ; For compatibility
.equ    TWS5    = 5     ; TWI Status
.equ    I2S5    = TWS5  ; For compatibility
.equ    TWS6    = 6     ; TWI Status
.equ    I2S6    = TWS6  ; For compatibility
.equ    TWS7    = 7     ; TWI Status
.equ    I2S7    = TWS7  ; For compatibility

; TWDR - TWI Data register
```

```
.equ    I2DR    = TWDR  ; For compatibility
.equ    TWD0    = 0     ; TWI Data Register Bit 0
.equ    TWD1    = 1     ; TWI Data Register Bit 1
.equ    TWD2    = 2     ; TWI Data Register Bit 2
.equ    TWD3    = 3     ; TWI Data Register Bit 3
.equ    TWD4    = 4     ; TWI Data Register Bit 4
.equ    TWD5    = 5     ; TWI Data Register Bit 5
.equ    TWD6    = 6     ; TWI Data Register Bit 6
.equ    TWD7    = 7     ; TWI Data Register Bit 7

; TWAR - TWI (Slave) Address register
.equ    I2AR    = TWAR  ; For compatibility
.equ    TWGCE   = 0     ; TWI General Call Recognition Enable Bit
.equ    TWA0    = 1     ; TWI (Slave) Address register Bit 0
.equ    TWA1    = 2     ; TWI (Slave) Address register Bit 1
.equ    TWA2    = 3     ; TWI (Slave) Address register Bit 2
.equ    TWA3    = 4     ; TWI (Slave) Address register Bit 3
.equ    TWA4    = 5     ; TWI (Slave) Address register Bit 4
.equ    TWA5    = 6     ; TWI (Slave) Address register Bit 5
.equ    TWA6    = 7     ; TWI (Slave) Address register Bit 6

; ***** ANALOG_COMPARATOR ************
; SFIOR - Special Function IO Register
.equ    ACME    = 3     ; Analog Comparator Multiplexer Enable

; ACSR - Analog Comparator Control And Status Register
.equ    ACIS0   = 0     ; Analog Comparator Interrupt Mode Select bit 0
.equ    ACIS1   = 1     ; Analog Comparator Interrupt Mode Select bit 1
.equ    ACIC    = 2     ; Analog Comparator Input Capture Enable
.equ    ACIE    = 3     ; Analog Comparator Interrupt Enable
.equ    ACI     = 4     ; Analog Comparator Interrupt Flag
.equ    ACO     = 5     ; Analog Compare Output
.equ    ACBG    = 6     ; Analog Comparator Bandgap Select
.equ    ACD     = 7     ; Analog Comparator Disable

; ***** AD_CONVERTER *****************
; ADMUX - The ADC multiplexer Selection Register
.equ    MUX0    = 0     ; Analog Channel and Gain Selection Bits
.equ    MUX1    = 1     ; Analog Channel and Gain Selection Bits
.equ    MUX2    = 2     ; Analog Channel and Gain Selection Bits
.equ    MUX3    = 3     ; Analog Channel and Gain Selection Bits
.equ    MUX4    = 4     ; Analog Channel and Gain Selection Bits
.equ    ADLAR   = 5     ; Left Adjust Result
.equ    REFS0   = 6     ; Reference Selection Bit 0
.equ    REFS1   = 7     ; Reference Selection Bit 1

; ADCSRA - The ADC Control and Status register
.equ    ADCSR   = ADCSRA        ; For compatibility
.equ    ADPS0   = 0     ; ADC  Prescaler Select Bits
.equ    ADPS1   = 1     ; ADC  Prescaler Select Bits
.equ    ADPS2   = 2     ; ADC  Prescaler Select Bits
.equ    ADIE    = 3     ; ADC Interrupt Enable
.equ    ADIF    = 4     ; ADC Interrupt Flag
.equ    ADATE   = 5     ; When this bit is written to one,the Timer/Counter2
    prescaler will be reset.The bit will be cleared by hardware after the operation
    is performed.Writing a zero to this bit will have no effect.This bit will always
    be read as zero if Timer/Counter2 is clocked by the internal CPU clock.If this
    bit is written when Timer/Counter2 is operating in asynchronous mode,the bit will
     remain one until the prescaler has been reset.
.equ    ADFR    = ADATE ; For compatibility
.equ    ADSC    = 6     ; ADC Start Conversion
.equ    ADEN    = 7     ; ADC Enable

; ADCH - ADC Data Register High Byte
.equ    ADCH0   = 0     ; ADC Data Register High Byte Bit 0
.equ    ADCH1   = 1     ; ADC Data Register High Byte Bit 1
.equ    ADCH2   = 2     ; ADC Data Register High Byte Bit 2
```

```
.equ    ADCH3   = 3     ; ADC Data Register High Byte Bit 3
.equ    ADCH4   = 4     ; ADC Data Register High Byte Bit 4
.equ    ADCH5   = 5     ; ADC Data Register High Byte Bit 5
.equ    ADCH6   = 6     ; ADC Data Register High Byte Bit 6
.equ    ADCH7   = 7     ; ADC Data Register High Byte Bit 7

; ADCL - ADC Data Register Low Byte
.equ    ADCL0   = 0     ; ADC Data Register Low Byte Bit 0
.equ    ADCL1   = 1     ; ADC Data Register Low Byte Bit 1
.equ    ADCL2   = 2     ; ADC Data Register Low Byte Bit 2
.equ    ADCL3   = 3     ; ADC Data Register Low Byte Bit 3
.equ    ADCL4   = 4     ; ADC Data Register Low Byte Bit 4
.equ    ADCL5   = 5     ; ADC Data Register Low Byte Bit 5
.equ    ADCL6   = 6     ; ADC Data Register Low Byte Bit 6
.equ    ADCL7   = 7     ; ADC Data Register Low Byte Bit 7

; SFIOR - Special Function IO Register
.equ    ADTS0   = 5     ; ADC Auto Trigger Source 0
.equ    ADTS1   = 6     ; ADC Auto Trigger Source 1
.equ    ADTS2   = 7     ; ADC Auto Trigger Source 2

; ***** JTAG ************************
; OCDR - On-Chip Debug Related Register in I/O Memory
.equ    OCDR0   = 0     ; On-Chip Debug Register Bit 0
.equ    OCDR1   = 1     ; On-Chip Debug Register Bit 1
.equ    OCDR2   = 2     ; On-Chip Debug Register Bit 2
.equ    OCDR3   = 3     ; On-Chip Debug Register Bit 3
.equ    OCDR4   = 4     ; On-Chip Debug Register Bit 4
.equ    OCDR5   = 5     ; On-Chip Debug Register Bit 5
.equ    OCDR6   = 6     ; On-Chip Debug Register Bit 6
.equ    OCDR7   = 7     ; On-Chip Debug Register Bit 7
.equ    IDRD    = OCDR7 ; For compatibility

; MCUCSR - MCU Control And Status Register
;.equ   JTRF    = 4     ; JTAG Reset Flag
;.equ   JTD     = 7     ; JTAG Interface Disable

; ***** BOOT_LOAD *******************
; SPMCSR - Store Program Memory Control Register
.equ    SPMCR   = SPMCSR        ; For compatibility
.equ    SPMEN   = 0     ; Store Program Memory Enable
.equ    PGERS   = 1     ; Page Erase
.equ    PGWRT   = 2     ; Page Write
.equ    BLBSET  = 3     ; Boot Lock Bit Set
.equ    RWWSRE  = 4     ; Read While Write section read enable
.equ    ASRE    = RWWSRE        ; For compatibility
.equ    RWWSB   = 6     ; Read While Write Section Busy
.equ    ASB     = RWWSB ; For compatibility
.equ    SPMIE   = 7     ; SPM Interrupt Enable

; ***** PORTA ***********************
; PORTA - Port A Data Register
.equ    PORTA0  = 0     ; Port A Data Register bit 0
.equ    PA0     = 0     ; For compatibility
.equ    PORTA1  = 1     ; Port A Data Register bit 1
.equ    PA1     = 1     ; For compatibility
.equ    PORTA2  = 2     ; Port A Data Register bit 2
.equ    PA2     = 2     ; For compatibility
.equ    PORTA3  = 3     ; Port A Data Register bit 3
.equ    PA3     = 3     ; For compatibility
.equ    PORTA4  = 4     ; Port A Data Register bit 4
.equ    PA4     = 4     ; For compatibility
.equ    PORTA5  = 5     ; Port A Data Register bit 5
.equ    PA5     = 5     ; For compatibility
.equ    PORTA6  = 6     ; Port A Data Register bit 6
.equ    PA6     = 6     ; For compatibility
.equ    PORTA7  = 7     ; Port A Data Register bit 7
```

```
.equ    PA7     = 7     ; For compatibility

; DDRA - Port A Data Direction Register
.equ    DDA0    = 0     ; Data Direction Register, Port A, bit 0
.equ    DDA1    = 1     ; Data Direction Register, Port A, bit 1
.equ    DDA2    = 2     ; Data Direction Register, Port A, bit 2
.equ    DDA3    = 3     ; Data Direction Register, Port A, bit 3
.equ    DDA4    = 4     ; Data Direction Register, Port A, bit 4
.equ    DDA5    = 5     ; Data Direction Register, Port A, bit 5
.equ    DDA6    = 6     ; Data Direction Register, Port A, bit 6
.equ    DDA7    = 7     ; Data Direction Register, Port A, bit 7

; PINA - Port A Input Pins
.equ    PINA0   = 0     ; Input Pins, Port A bit 0
.equ    PINA1   = 1     ; Input Pins, Port A bit 1
.equ    PINA2   = 2     ; Input Pins, Port A bit 2
.equ    PINA3   = 3     ; Input Pins, Port A bit 3
.equ    PINA4   = 4     ; Input Pins, Port A bit 4
.equ    PINA5   = 5     ; Input Pins, Port A bit 5
.equ    PINA6   = 6     ; Input Pins, Port A bit 6
.equ    PINA7   = 7     ; Input Pins, Port A bit 7

; ***** PORTB ************************
; PORTB - Port B Data Register
.equ    PORTB0  = 0     ; Port B Data Register bit 0
.equ    PB0     = 0     ; For compatibility
.equ    PORTB1  = 1     ; Port B Data Register bit 1
.equ    PB1     = 1     ; For compatibility
.equ    PORTB2  = 2     ; Port B Data Register bit 2
.equ    PB2     = 2     ; For compatibility
.equ    PORTB3  = 3     ; Port B Data Register bit 3
.equ    PB3     = 3     ; For compatibility
.equ    PORTB4  = 4     ; Port B Data Register bit 4
.equ    PB4     = 4     ; For compatibility
.equ    PORTB5  = 5     ; Port B Data Register bit 5
.equ    PB5     = 5     ; For compatibility
.equ    PORTB6  = 6     ; Port B Data Register bit 6
.equ    PB6     = 6     ; For compatibility
.equ    PORTB7  = 7     ; Port B Data Register bit 7
.equ    PB7     = 7     ; For compatibility

; DDRB - Port B Data Direction Register
.equ    DDB0    = 0     ; Port B Data Direction Register bit 0
.equ    DDB1    = 1     ; Port B Data Direction Register bit 1
.equ    DDB2    = 2     ; Port B Data Direction Register bit 2
.equ    DDB3    = 3     ; Port B Data Direction Register bit 3
.equ    DDB4    = 4     ; Port B Data Direction Register bit 4
.equ    DDB5    = 5     ; Port B Data Direction Register bit 5
.equ    DDB6    = 6     ; Port B Data Direction Register bit 6
.equ    DDB7    = 7     ; Port B Data Direction Register bit 7

; PINB - Port B Input Pins
.equ    PINB0   = 0     ; Port B Input Pins bit 0
.equ    PINB1   = 1     ; Port B Input Pins bit 1
.equ    PINB2   = 2     ; Port B Input Pins bit 2
.equ    PINB3   = 3     ; Port B Input Pins bit 3
.equ    PINB4   = 4     ; Port B Input Pins bit 4
.equ    PINB5   = 5     ; Port B Input Pins bit 5
.equ    PINB6   = 6     ; Port B Input Pins bit 6
.equ    PINB7   = 7     ; Port B Input Pins bit 7

; ***** PORTC ************************
; PORTC - Port C Data Register
.equ    PORTC0  = 0     ; Port C Data Register bit 0
.equ    PC0     = 0     ; For compatibility
.equ    PORTC1  = 1     ; Port C Data Register bit 1
.equ    PC1     = 1     ; For compatibility
```

```
.equ    PORTC2  = 2     ; Port C Data Register bit 2
.equ    PC2     = 2     ; For compatibility
.equ    PORTC3  = 3     ; Port C Data Register bit 3
.equ    PC3     = 3     ; For compatibility
.equ    PORTC4  = 4     ; Port C Data Register bit 4
.equ    PC4     = 4     ; For compatibility
.equ    PORTC5  = 5     ; Port C Data Register bit 5
.equ    PC5     = 5     ; For compatibility
.equ    PORTC6  = 6     ; Port C Data Register bit 6
.equ    PC6     = 6     ; For compatibility
.equ    PORTC7  = 7     ; Port C Data Register bit 7
.equ    PC7     = 7     ; For compatibility

; DDRC - Port C Data Direction Register
.equ    DDC0    = 0     ; Port C Data Direction Register bit 0
.equ    DDC1    = 1     ; Port C Data Direction Register bit 1
.equ    DDC2    = 2     ; Port C Data Direction Register bit 2
.equ    DDC3    = 3     ; Port C Data Direction Register bit 3
.equ    DDC4    = 4     ; Port C Data Direction Register bit 4
.equ    DDC5    = 5     ; Port C Data Direction Register bit 5
.equ    DDC6    = 6     ; Port C Data Direction Register bit 6
.equ    DDC7    = 7     ; Port C Data Direction Register bit 7

; PINC - Port C Input Pins
.equ    PINC0   = 0     ; Port C Input Pins bit 0
.equ    PINC1   = 1     ; Port C Input Pins bit 1
.equ    PINC2   = 2     ; Port C Input Pins bit 2
.equ    PINC3   = 3     ; Port C Input Pins bit 3
.equ    PINC4   = 4     ; Port C Input Pins bit 4
.equ    PINC5   = 5     ; Port C Input Pins bit 5
.equ    PINC6   = 6     ; Port C Input Pins bit 6
.equ    PINC7   = 7     ; Port C Input Pins bit 7

; ***** PORTD ************************
; PORTD - Port D Data Register
.equ    PORTD0  = 0     ; Port D Data Register bit 0
.equ    PD0     = 0     ; For compatibility
.equ    PORTD1  = 1     ; Port D Data Register bit 1
.equ    PD1     = 1     ; For compatibility
.equ    PORTD2  = 2     ; Port D Data Register bit 2
.equ    PD2     = 2     ; For compatibility
.equ    PORTD3  = 3     ; Port D Data Register bit 3
.equ    PD3     = 3     ; For compatibility
.equ    PORTD4  = 4     ; Port D Data Register bit 4
.equ    PD4     = 4     ; For compatibility
.equ    PORTD5  = 5     ; Port D Data Register bit 5
.equ    PD5     = 5     ; For compatibility
.equ    PORTD6  = 6     ; Port D Data Register bit 6
.equ    PD6     = 6     ; For compatibility
.equ    PORTD7  = 7     ; Port D Data Register bit 7
.equ    PD7     = 7     ; For compatibility

; DDRD - Port D Data Direction Register
.equ    DDD0    = 0     ; Port D Data Direction Register bit 0
.equ    DDD1    = 1     ; Port D Data Direction Register bit 1
.equ    DDD2    = 2     ; Port D Data Direction Register bit 2
.equ    DDD3    = 3     ; Port D Data Direction Register bit 3
.equ    DDD4    = 4     ; Port D Data Direction Register bit 4
.equ    DDD5    = 5     ; Port D Data Direction Register bit 5
.equ    DDD6    = 6     ; Port D Data Direction Register bit 6
.equ    DDD7    = 7     ; Port D Data Direction Register bit 7

; PIND - Port D Input Pins
.equ    PIND0   = 0     ; Port D Input Pins bit 0
.equ    PIND1   = 1     ; Port D Input Pins bit 1
.equ    PIND2   = 2     ; Port D Input Pins bit 2
.equ    PIND3   = 3     ; Port D Input Pins bit 3
```

```
.equ    PIND4    = 4      ; Port D Input Pins bit 4
.equ    PIND5    = 5      ; Port D Input Pins bit 5
.equ    PIND6    = 6      ; Port D Input Pins bit 6
.equ    PIND7    = 7      ; Port D Input Pins bit 7

; ***** WATCHDOG ********************
; WDTCR - Watchdog Timer Control Register
.equ    WDP0     = 0      ; Watch Dog Timer Prescaler bit 0
.equ    WDP1     = 1      ; Watch Dog Timer Prescaler bit 1
.equ    WDP2     = 2      ; Watch Dog Timer Prescaler bit 2
.equ    WDE      = 3      ; Watch Dog Enable
.equ    WDTOE    = 4      ; RW
.equ    WDDE     = WDTOE  ; For compatibility

; ***** LOCKSBITS ********************************************************
.equ    LB1      = 0      ; Lock bit
.equ    LB2      = 1      ; Lock bit
.equ    BLB01    = 2      ; Boot Lock bit
.equ    BLB02    = 3      ; Boot Lock bit
.equ    BLB11    = 4      ; Boot lock bit
.equ    BLB12    = 5      ; Boot lock bit

; ***** FUSES ************************************************************
; LOW fuse bits
.equ    CKSEL0   = 0      ; Select Clock Source
.equ    CKSEL1   = 1      ; Select Clock Source
.equ    CKSEL2   = 2      ; Select Clock Source
.equ    CKSEL3   = 3      ; Select Clock Source
.equ    SUT0     = 4      ; Select start-up time
.equ    SUT1     = 5      ; Select start-up time
.equ    BODEN    = 6      ; Brown out detector enable
.equ    BODLEVEL         = 7      ; Brown out detector trigger level

; HIGH fuse bits
.equ    BOOTRST  = 0      ; Select Reset Vector
.equ    BOOTSZ0  = 1      ; Select Boot Size
.equ    BOOTSZ1  = 2      ; Select Boot Size
.equ    EESAVE   = 3      ; EEPROM memory is preserved through chip erase
.equ    CKOPT    = 4      ; Oscillator Options
.equ    SPIEN    = 5      ; Enable Serial programming and Data Downloading
.equ    JTAGEN   = 6      ; Enable JTAG
.equ    OCDEN    = 7      ; Enable OCD

; ***** CPU REGISTER DEFINITIONS *****************************************
.def    XH       = r27
.def    XL       = r26
.def    YH       = r29
.def    YL       = r28
.def    ZH       = r31
.def    ZL       = r30

; ***** DATA MEMORY DECLARATIONS *****************************************
.equ    FLASHEND         = 0x1fff         ; Note: Word address
.equ    IOEND    = 0x003f
.equ    SRAM_START       = 0x0060
.equ    SRAM_SIZE        = 1024
.equ    RAMEND   = 0x045f
.equ    XRAMEND  = 0x0000
.equ    E2END    = 0x01ff
.equ    EEPROMEND        = 0x01ff
.equ    EEADRBITS        = 9
#pragma AVRPART MEMORY PROG_FLASH 16384
#pragma AVRPART MEMORY EEPROM 512
#pragma AVRPART MEMORY INT_SRAM SIZE 1024
#pragma AVRPART MEMORY INT_SRAM START_ADDR 0x60

; ***** BOOTLOADER DECLARATIONS ******************************************
```

```
.equ    NRWW_START_ADDR = 0x1c00
.equ    NRWW_STOP_ADDR  = 0x1fff
.equ    RWW_START_ADDR  = 0x0
.equ    RWW_STOP_ADDR   = 0x1bff
.equ    PAGESIZE        = 64
.equ    FIRSTBOOTSTART  = 0x1f80
.equ    SECONDBOOTSTART = 0x1f00
.equ    THIRDBOOTSTART  = 0x1e00
.equ    FOURTHBOOTSTART = 0x1c00
.equ    SMALLBOOTSTART  = FIRSTBOOTSTART
.equ    LARGEBOOTSTART  = FOURTHBOOTSTART

; ***** INTERRUPT VECTORS ************************************************
.equ    INT0addr        = 0x0002        ; External Interrupt Request 0
.equ    INT1addr        = 0x0004        ; External Interrupt Request 1
.equ    OC2addr = 0x0006        ; Timer/Counter2 Compare Match
.equ    OVF2addr        = 0x0008        ; Timer/Counter2 Overflow
.equ    ICP1addr        = 0x000a        ; Timer/Counter1 Capture Event
.equ    OC1Aaddr        = 0x000c        ; Timer/Counter1 Compare Match A
.equ    OC1Baddr        = 0x000e        ; Timer/Counter1 Compare Match B
.equ    OVF1addr        = 0x0010        ; Timer/Counter1 Overflow
.equ    OVF0addr        = 0x0012        ; Timer/Counter0 Overflow
.equ    SPIaddr = 0x0014        ; Serial Transfer Complete
.equ    URXCaddr        = 0x0016        ; USART, Rx Complete
.equ    UDREaddr        = 0x0018        ; USART Data Register Empty
.equ    UTXCaddr        = 0x001a        ; USART, Tx Complete
.equ    ADCCaddr        = 0x001c        ; ADC Conversion Complete
.equ    ERDYaddr        = 0x001e        ; EEPROM Ready
.equ    ACIaddr = 0x0020        ; Analog Comparator
.equ    TWIaddr = 0x0022        ; 2-wire Serial Interface
.equ    INT2addr        = 0x0024        ; External Interrupt Request 2
.equ    OC0addr = 0x0026        ; Timer/Counter0 Compare Match
.equ    SPMRaddr        = 0x0028        ; Store Program Memory Ready

.equ    INT_VECTORS_SIZE        = 42    ; size in words

#endif  /* _M16DEF_INC_ */

; ***** END OF FILE ******************************************************
```