RHODES UNIVERSITY
Where leaders learn

Department of Physics & Electronics

Lecture Notes

Physics Honours

Computer Interfacing 1

An Introduction to Assembler and the ATMega16

Anthony Sullivan

Ver: 2022-01-05

Preface

Lecture Times: 8:00 to 9:00 (Mon,Thur,Fri for the first 2 weeks, Mon,Tue,Wed,Thur,Fri after the first 2 weeks.)

Lecture Venue: Room 17 Physics Department (above the Fountain Lab).

Assessment: As we progress through the material you will be expected to look at the tasks assigned and alter the given code to do things "properly" and extend functionality. These will not be marked or assessed, they are for you to make sure you know what is going on in the course. The course mark is made up from the final Theory exam in June and an end of course assignment. At the end of the course you will receive a specification for code that you have to write as well as a functioning binary release. You will have about 4 days to replicate this. You will be expected to hand back your working hardware along with an electronic submission of your code (binary and source). You will then write a short 45 minute test that ensures you know how your code works.

Amount of work: Everything contained within these notes is relevant (as well as the corresponding section in the datasheet), you can expect to be spending about 2-3 hours per day reading and working on problems.

Common Pitfalls: Often people assume that they can "just catch up" right at the end of the course, by just going through the last few tasks. This has lead to people getting "unexpected" marks. Everything done near the end of the course builds on everything else.

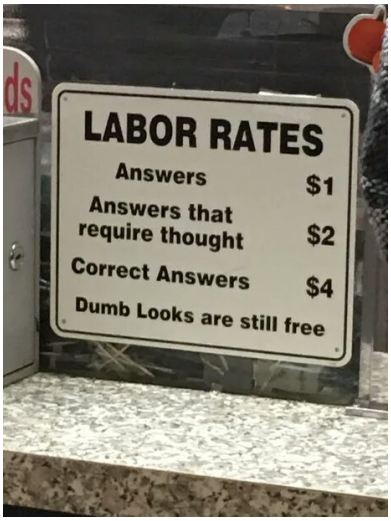
There are intentional mistakes in some of the tasks (not the examples - I am not that much of a bastard) always READ code before you try something.

Problems: If you are having trouble with something (after reading through the notes and datasheets and then watching the RUConnected videos ask questions. (A.Sullivan@ru.ac.za). Also use the forums on RUConnected to chat to other members of the class - they might have already experienced the problem you are having.

Contents

CHAPTER

1 Introduction



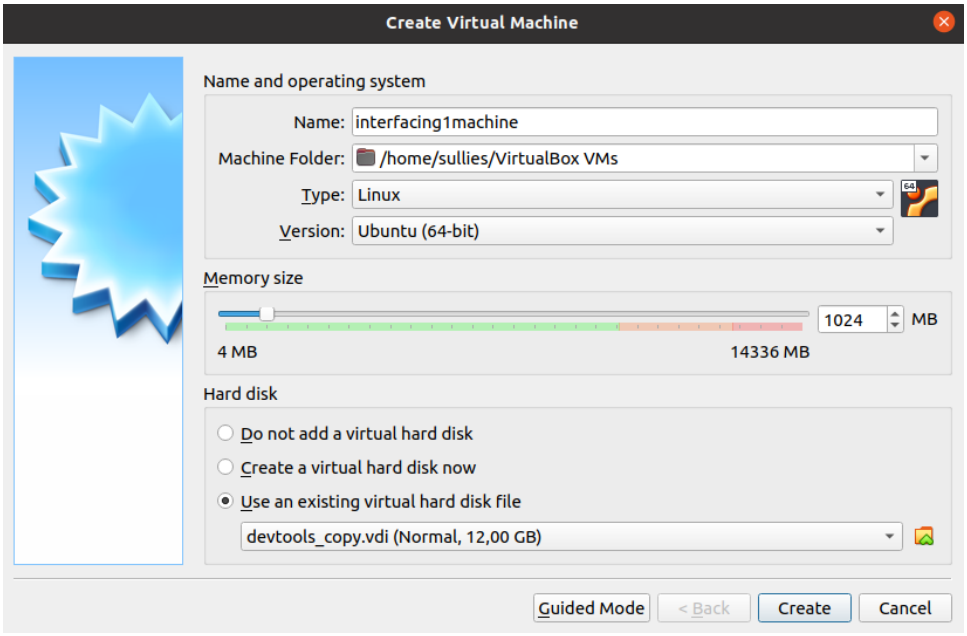
1.1 Installing the Development Environment

This is the first step, without it you will not be able to do the course. Fortunately for you I have provided a virtual machine with all the tools pre-installed and set up. All you have to do is install Virtual-Box on your host machine and point it to the drive.

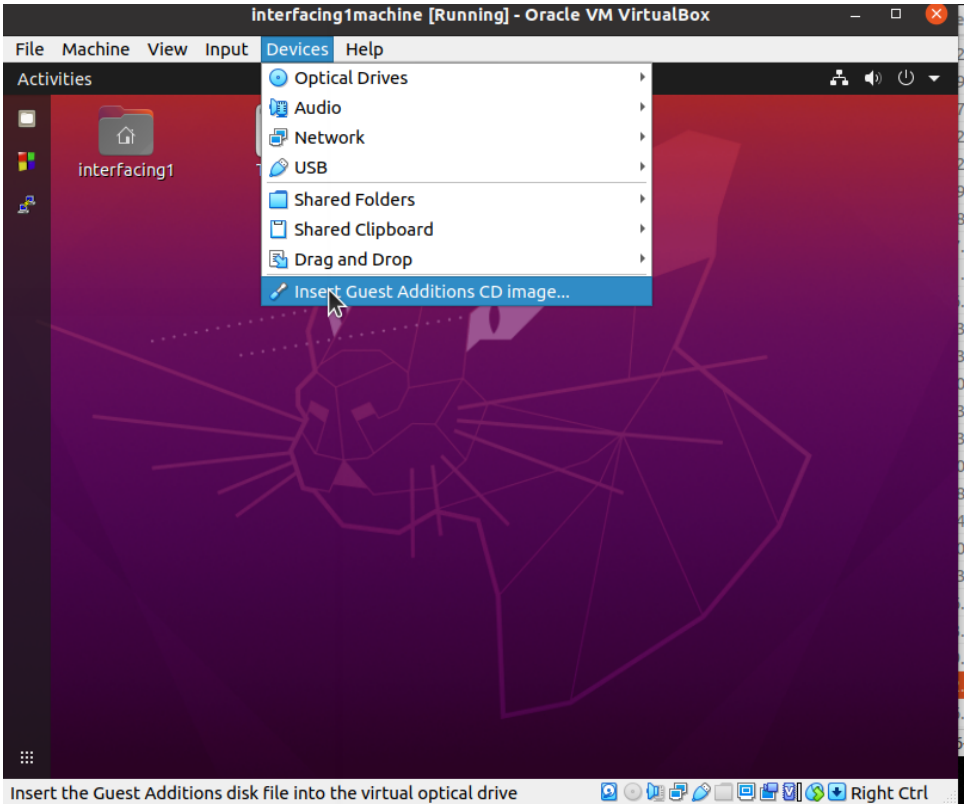
The username of the account on the machine is interfacing1, the password for it is interfacing1.

Getting the Virtual Disk Image Running

- Self enrol to the Interfacing I course on RUConnected (located under *Physics & Electronics*).
- There is a link on the RUConnected page to a file shared via Google Drive (±6 GB)
- When you try and download it (if you are not logged-in to google using a .ru.ac.za account you will have to wait for me to grant you access to the file.
- Install Virtualbox (I recommend ver 6.1). (Downloads for various operating systems can be found at: https://www.virtualbox.org/wiki/Downloads)
- If you are installing on a Linux host add your user to the vboxusers group. Ask google for instructions for your particular flavour of Linux. (If you are using a Linux host you will also have to add your user to the group with access to serial terminals - on Ubuntu this group is dialout.
- Reboot, trust me - reboot.
- Create a virtual machine. See below.



- Start the machine.
- Log in to the virtual machine.
- Install “Guest Additions”.

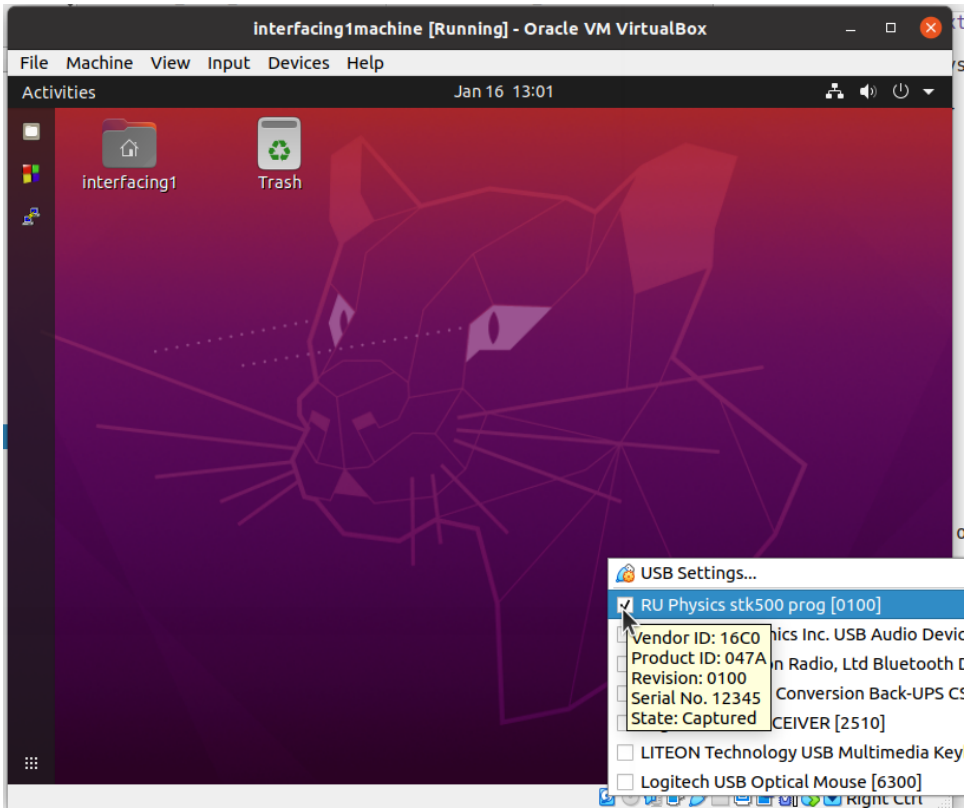


Connecting the Development board to the Virtual Machine

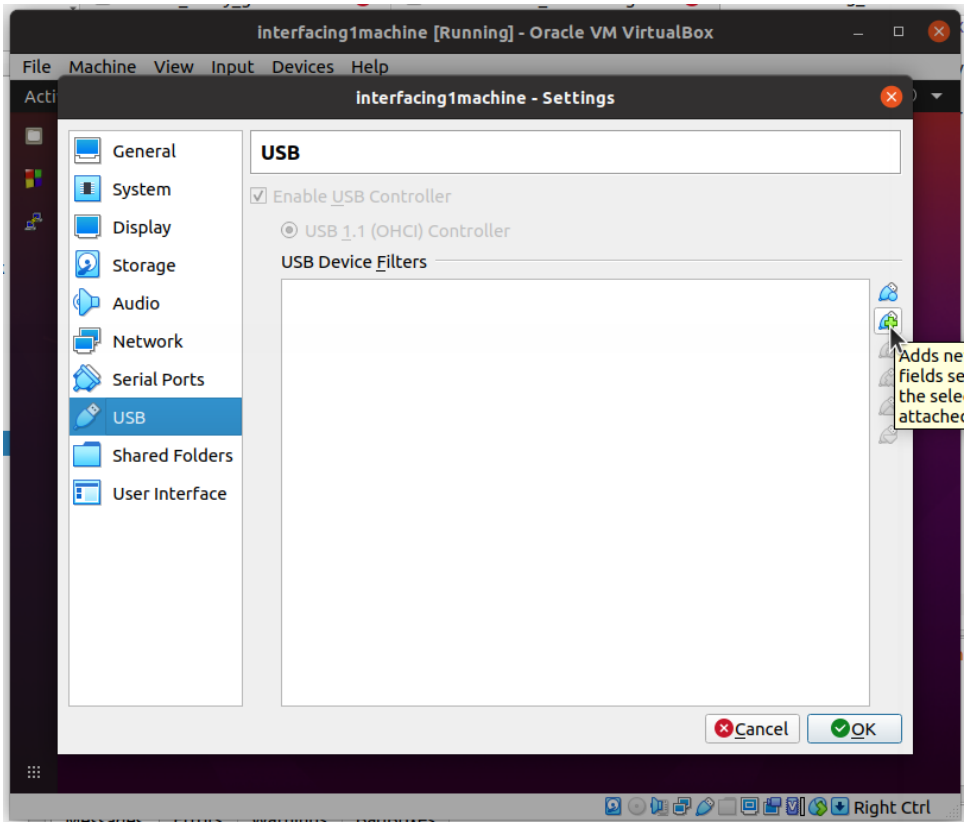
Now we are ready to finally connect out development board.

- Make sure the mode selector on the programmer is on “serial mode” (jumper on 2/3) - see page 174.
- Plug USB cable into Host and development environment.
- Right-click the USB icon on the bottom of the Virtual Machine Window.

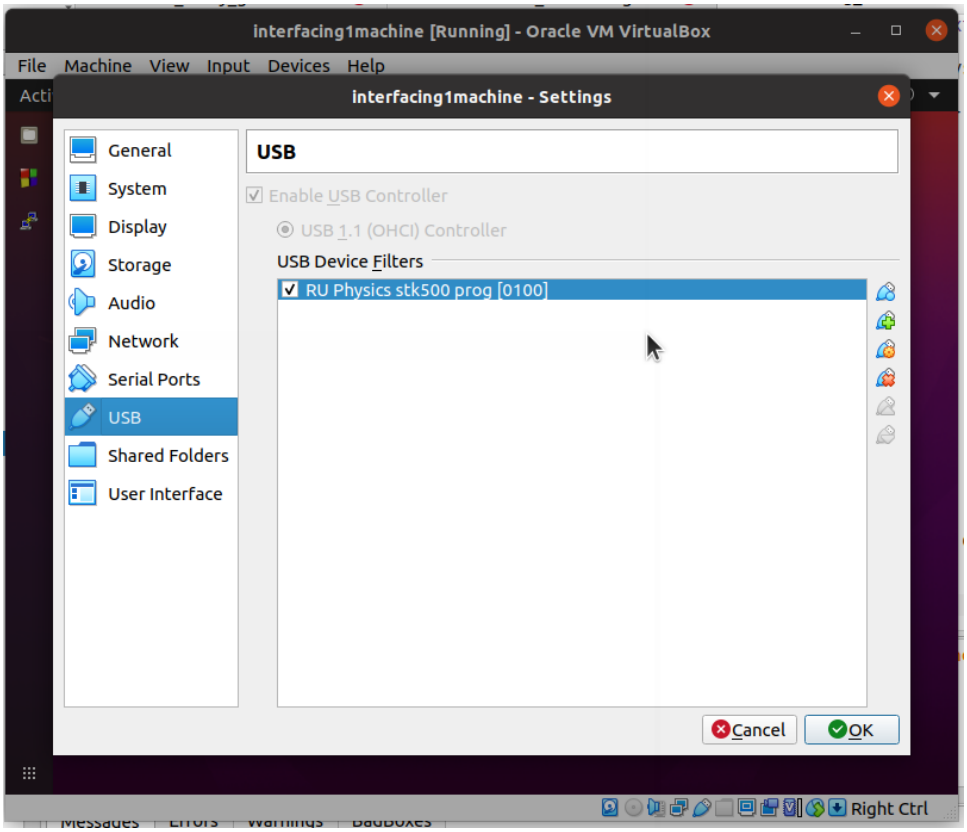
- Select the Programmer as shown.



- Right-click again and select USB settings.
- Click the indicated filter



- Select the programmer again.



- Now whenever the virtual machine is running and you plug in the development hardware it will be automatically be switched through to the Virtual Machine.

See Video0-InstallingVirtualMachine.

1.2 What is Embedded Programming?

Embedded programming is the term for the computer programming that lives in and operates the great many computer-controlled devices that surround us in our homes, cars, workplaces and communities. To be clear, all microcontroller programming is embedded programming, but not all embedded programming is microcontroller programming. A little more will be said about this further along. Sometimes the terms will be used interchangeably, but the focus of this course is always on microcontrollers.

For every desktop or notebook or tablet computer you have, you may have a dozen or more (perhaps a great deal more) microcontrollers quietly doing their embedded duty, and with these devices many people don't even realize they involve a tiny computer running a program. But there is, and it is, and those programs had to be written, and that's why the world needs embedded programming. Microcontrollers add intelligence to countless devices and systems, enabling those devices and systems to operate better, faster, more safely, more efficiently, more conveniently, more usefully, and in many cases allowing the very existence of devices and systems that could not be built otherwise. Spend some time looking around you and trying to recognize where μCs are working, and you will begin to get a sense of how ubiquitous they have become since their invention some 40+ years ago.

In ways very different from most desktop or mainframe programming, embedded programs make stuff do stuff, and to an embedded programmer, stuff doing stuff is what makes us happy.

1.3 What is an Embedded System?

There's no perfect answer to that question, since every answer will have some exceptions. However, for our purposes let us declare that an embedded system is one that uses one or more microcomputers (that is, small to very, very small computers), running custom dedicated programs and connected to specialized hardware, to perform a dedicated set of functions. This can be contrasted with a general-purpose computer such as the familiar desktop or notebook, which are not designed to run only one dedicated program with one specialized set of hardware. It is not a perfect definition but it is a start.

Some examples of embedded systems are:

- Alarm / security system
- Automobile cruise control
- Heating / air conditioning thermostat
- Microwave oven
- Anti-skid braking controller
- Traffic light controller
- Vending machine
- Petrol pump
- Irrigation system controller
- Multicopter
- Oscilloscope
- Mars Rover

For the most part I have listed example embedded applications on the less-complex end of the spectrum, since this is after all a beginning tutorial. By the end of this course you should have a good general idea how most of these applications would be programmed, and in rough terms what kinds of IO, timing, interrupt and communications hardware and functionality they would require.

There are a few things worth noticing about the above list. While many embedded systems use fairly traditional user input-output devices (keypads, displays), many others do not. Also, many embedded systems interact directly with human beings, but others do not (and we're still waiting to see if the Mars Rover will interact directly with any Martians).

1.4 What is different about embedded programming?

Embedded programs must work closely with the specialized components and custom circuitry that makes up the hardware. Unlike programming on top of a full-function operating system, where the hardware details are removed as much as possible from the programmer's notice and control, most embedded programming acts directly with and on the hardware. This includes not only the hardware of the CPU, but also the hardware which makes up all the peripherals (both on-chip and off-chip) of the system. Thus an embedded programmer must have a good knowledge of hardware, at least as it pertains to writing software that correctly interfaces with and manipulates that hardware. This knowledge will often extend to specifying key components of the hardware (microcontroller, memory devices, I/O devices, etc), and in smaller organizations will sometimes go as far as designing and laying out (as a printed circuit board) the hardware. An embedded programmer will also need to have a good understanding of debugging equipment such as multimeters, oscilloscopes, logic analysers and the like.

Another difference from general purpose computers is that most (but not all) embedded systems are quite limited as compared to the former. The microcomputers used in embedded systems may have program memory sizes of a few thousand to a few hundred thousand bytes rather than the gigabytes in the desktop machine, and will typically have even less data (RAM) memory than program memory. Further, the CPU will often be smaller 8 and 16 bit devices as opposed to the 32 bit and larger devices found in a desktop. A smaller CPU word size means, among other things, that a program will require more instructions (and thus more clock cycles) than an equivalent program running on a CPU with a larger word size. And finally, the speed at which smaller microcontrollers run is much less than the speed at which a PC runs. Typical smaller microcontroller clock rates are between 1 and 200 MHz, not the GHz rates of PCs.

1.5 What are the differences between Microcontrollers, Microprocessors and Microcomputers?

A microprocessor is usually understood to be a single-chip central processing unit (CPU), with the CPU being the "brains" of a computer - the part of the computer that executes program instructions. A microcomputer is any computer built around a microprocessor, along with program and data memory, and I/O devices and other peripherals as needed. A microcontroller (often shortened to μC or MCU in this course) is a single chip device which has built onto the chip not only a microprocessor but also on the same chip, nonvolatile program (ROM) and volatile data (RAM) memory, along with useful peripherals such as general-purpose I/O (GPIO), timers and serial communications channels. Thus it follows that all microcontrollers are microcomputers, but not all microcomputers use microcontrollers.

In smaller embedded systems it is most common to use microcontrollers rather than microprocessor-based designs since microcontrollers give the most compact design and the lowest hardware cost. Larger embedded systems, on the other hand, may use one or more microprocessors if a microcontroller of suitable speed and functionality cannot be found. This can extend to the use of industrial PCs and even more powerful hardware. It is also possible to include both microprocessors and microcontrollers in a complex embedded system. The only real rules are, use whatever device(s) fit the task, given the constraints on budget, availability, time, tools, etc.

It should also be pointed out that with most microcontrollers it is possible to add external memory and peripherals, should the on-board mix not take care of all the system needs. When it makes sense to add such external devices, as opposed to choosing a larger microcontroller with the needed resources on-board, is a choice that needs to be made on an individual design basis.

1.6 What is an N-bit Microcontroller?

There is some discussion about what it means to call a device an N-bit processor, but it's fairly obvious in most cases. If the device can perform most of its data manipulation instructions on data words up to N bits in size, the device is an N-bit processor. By way of example, a device may have a full set of instructions that can operate on 8 bit data, along with a few instructions that operate on 16 bit data. That device should be considered an 8-bit design, even if the marketing department says otherwise and calls it a 16-bit chip.

By volume, 8-bit microcontrollers are the biggest segment of the embedded market. Many applications simply don't need any more power, and never will. 16-bit devices are more powerful, but they are squeezed between the 8-bit devices on the low end and the 32-bit devices on the high end. 32-bit devices are at the high end of the embedded spectrum for all but the most complex or high-performance designs, but they are moving ever downward in price.

1.7 So What exactly are we going to be going?

We are going to be using an Atmel ATMega16 on a custom development board to investigate and understand the AVR 8-bit RISC architecture. We will also look at programming techniques and methods needed in Embedded programming using the Atmel assembly language.

Part I How to procede from here...

The Lessons presented here follow the lectures. There are also videos highlighting concepts etc on RUConnected.

COVID-Alert

Should you currently find yourself in 202x and face to face lectures are not happening, then this section is basically your lesson plan. If you have questions that are not answered in the RUConnected videos then email your questions to me... (A.Sullivan@ru.ac.za).

PS: When using a new peripheral or feature it is also advisable to read over the corresponding sections in the actual datasheet.

Chapter 2

The Virtual Machine is up and running so what now?

Before we start programming we need to do a bit of reading/watching videos, because just throwing code around without knowing what you are doing never helped anyone.

Stuff to read:

- Chapter 19 on page 92.
- Chapter 20 on page 95.

Videos (on RUConnected) to watch:

- Video1-ArchitectureIntroduction
- Video2-DevBoardLayout
- Video3-AssemblerDirectives-A
- Video4-InstructionSet-A

CHAPTER

3 Looking at IO control

So now that we have covered a bit of material we can look at our first bit of coding. We are going to be looking at Task 1 on page **??** as a place to start.

Stuff to read:

- Chapter 26 on page 113 about IO Ports and using them.
- Read over the instruction set summary for all instructions used in Task 1, reviewing the assembler directives might also help.

Videos (on RUConnected) to watch:

- Video5-IOPorts
- Task1-Overview
- Task1-Solution (if you are getting stuck)

3.1 Task 1 - Simple IO - A rather expensive switch

What to read through beforehand

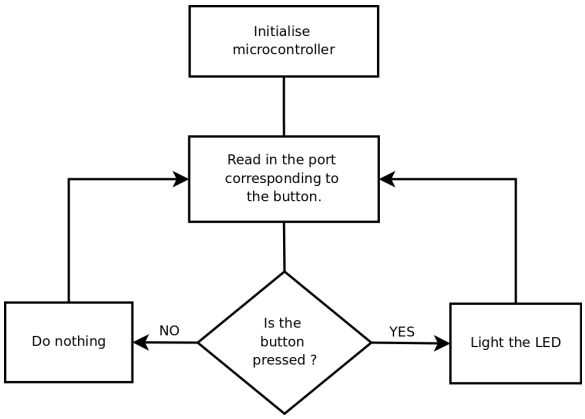
- Familiarise yourself with the instruction set, particularly the branch instructions.
- Familiarise yourself with the IO port section in the datasheet (pages 50-67) and notes.
- Read through and understand the layout of the switches/LEDs in the "Mega16 Devboard" part.

The problem

We would like an led to light up whenever a switch is held down.

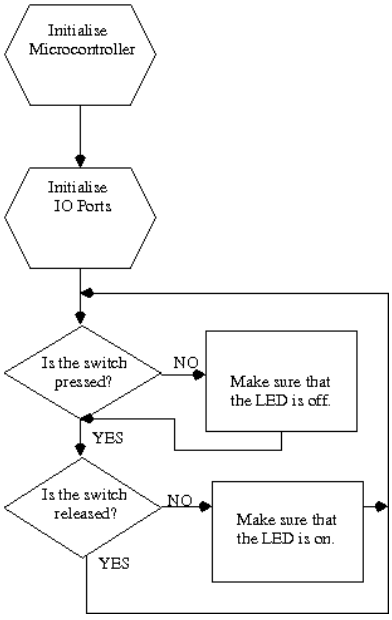
The solution

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.

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



The Code

```
;LED-SWITCH - Task1
; This program toggles portB pin 0
; Remember to patch PortB0 to an LED
; and PORTD0 to Switch0
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST

.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
```

Easter Egg

Look closely at the code - something is wrong. It works but there is a problem . . .

To Do

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. (You can do this in one of two ways: using logic operations or by using the structure of your code)

Chapter

4 IO and Bitmasks

Stuff to read:

- Chapter 26 on page 113 about IO Ports and using them.
- Chapter 37 on page 164 about bitmasks and using them.

Videos (on RUConnected) to watch:

- Video6-Bitmasks
- Task2-Overview
- Task2-Solution
- Task3-Overview
- Task3-Solution

4.1 Task 2 - Simpler Expensive switches

What to read through beforehand

- Familiarise yourself with the IN and OUT instructions.
- Familiarise your self with the bitwise logical operators.

The problem

We would like to use all four switches to control 4 of the LEDs.

The solution

There has to be an easier was than extending the previous task, there is. We are going to read in the value of the switches (all at the same time) and simply output them to the LEDs.

The Code

```
;LED-SWITCH - Task2
; Remember to patch PortB0-3 to an LEDs 0-3
; and PORTD0-3 to Switch0-3
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST

.def TMP1=R16
.def ZERO=R0
.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:
     IN tmp1, pinb
     out portd, tmp1
      RJMP MAIN_LOOP

INITIALISE:
     EOR R0, R0 ; R0 now has the value 0x00
     LDI TMP1, 0X0F ; make the lower 4 bits of portd output
     OUT DDRD, TMP1
     out portd, r0  ; make sure pd0-3 are lo
     out ddrb, r0 ; make portb inputs
     out portb, r0
      RET                        ;Return from subroutine
```

To Do

- As the code stands, the values on PORTD(4-7),(which might be used by other procedures in a larger project) are changed by our output routine. Change the code (still using the IN and OUT instructions) so that the values of PORTD(4-7) would not be changed.

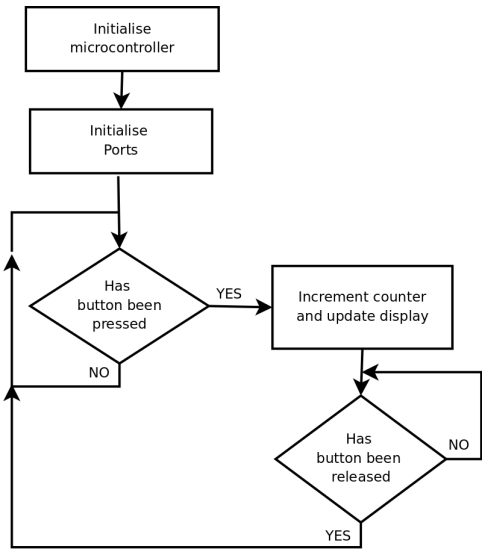
4.2 Task 3 - Counting Button Presses

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

```
; Connect sw0-1 to pd0-1
; Connect PB0-7 to led0-7
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST

.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;
```

```
33        OUT PORTB, COUNT
34  SAMPLE_UP:
35        SBIS PIND, 0 ; skip if button released
36        RJMP SAMPLE_UP
37        RJMP SAMPLE_DOWN
38
39  INITIALISE:
40        ; Setup port b as output for LEDs
41        ; and inital state to all off and set counter to zero
42        CLR COUNT
43        OUT DDRB, TMP1
44        OUT PORTB, TMP1 ; 0x00 = all off
45        OUT DDRD, COUNT ; make d inputs (count is a handy clear register)
46        RET
```

To Do

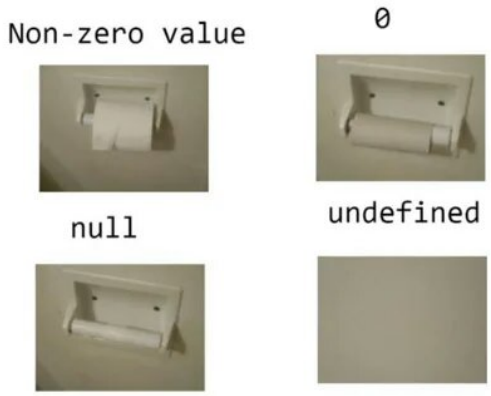
1. Modify (or rewrite) the code so that switch 0 increments the counter and switch 1 decrements the counter. Only register a button press if the previous button has been released.
2. Modify your code so that instead of displaying the count, pressing a button moves a lit LED up or down the display. (Start with LED0 lit).

Easter Egg

As soon as you wire up all the LEDs to PORTB the device refuses to program. Why does this happen? (HINT: Look at the section explaining the Development Board.)

Chapter

5 (External) Interrupts



Stuff to read:

- Chapter 21 on page 99 about Interrupts.
- The section on External Interrupts on page 68 of the Mega16 Datasheet.

Videos (on RUConnected) to watch:

- Video7-InterruptsinGeneral
- Video8-ExternalInterrupts
- Task4-Overview
- Task4-Solution

5.1 Task 4 - Counting Button Presses using Interrupts

What to read through beforehand

- The section on External Interrupts

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 Code

```
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST

.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 portd is an input
      SBI PORTD,2          ; enable pull-ups on int0
      SBI PORTD,3          ; enable pull-ups on int1
      OUT PORTB, TMP1      ; port B all low
     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
```

```
52      RET
53
54 INT0_ROUTINE:
55 ;int 0 is to increment the counter
56      PUSH TMP1       ;save tmp1
57      IN TMP1, SREG
58      PUSH TMP1       ; save sreg
59      DEC COUNT       ; decrement the counter
60      OUT PORTB, count     ; output
61      POP TMP1
62      OUT SREG, TMP1 ;resotre sreg
63      POP TMP1        ;resotr tmp1
64      RETI
65
66 INT1_ROUTINE:
67 ;int 1 is to increment the counter
68      PUSH TMP1
69      IN TMP1, SREG
70      PUSH TMP1
71      INC COUNT
72      OUT PORTB, count
73      POP TMP1
74      OUT SREG, TMP1
75      POP TMP1
76      RETI
```

To Do

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.

CHAPTER

6 Introduction to Timers & Counters

Stuff to read:

- Read the timer/counter chapter on page 115.

Videos (on RUConnected) to watch:

- Video9-Timers1
- Video10-Timers2
- Video11-16bitRegisters
- Task5-Overview
- Task5-Solution
- Task6-Overview

6.1 Timers / Counters

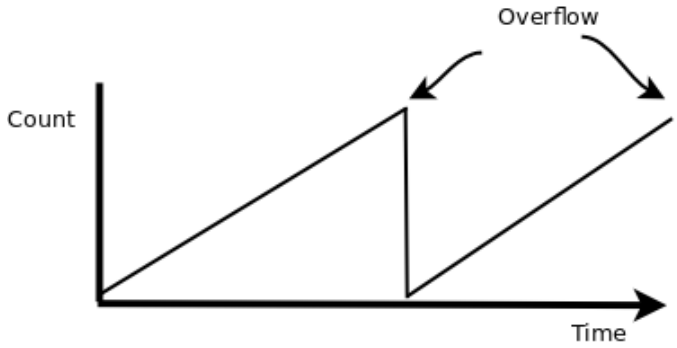
6.1.1 Overview

What is the difference between a timer and a counter?

The main difference in terminology is that it is refered to as a timer when the clocksource driving it is a source with a well-defined periodicity.

6.1.2 General behaviour

Generally a counter starts at some value (usually zero) and counts up as shown below.



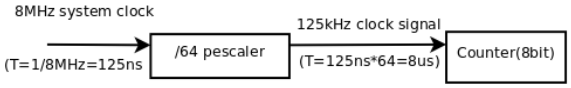
6.1.3 General Timer Control

There are two GPIO registers that govern timers overall:

- TIMSK: This is the interrupts mask register for all available timer interrupts. (A 1 in a bit position enables the corresponding interrupt. (Explained on pages 85,115,133 of the datasheet.)
- TIFR: This is the flag register for the corresponding timer interrupts. They work the same as the flags for the external interrupts. (Explained on pages 85,115,133 of the datasheet.)
- Besides the obvious size difference of timer 1, the interrupts of the timers vary greatly in their priority - this is important when designing a system and assigning timers to tasks that vary from 'must be done' to 'must be done now'.
- Timer1 also has *input capture* interrupts. This is usefull for timestamping events accurately.

6.1.4 Prescaling

The prescaling of the timers helps us adjust the speed at which they count.



- The timer in the example above will increment its value every 8 μs when prescaled, rather than every 125 ns when not prescaled.
- If the counter started from 0, then it would take $T_{overflow}$ to overflow where:

$$T_{overflow} = 125 \times 10^{-9} * 64_{prescalevalue} * 255 = 2.04 \text{ ms}$$

6.1.5 Modes

- **Normal Mode:**The timer starts counting and overflows, generating configured interrupts when condidtions are met.

- **CTC:** The timer will clear itself and restart from 0 on a compare-match, interrupts will be generated if configured.
- **Fast PWM:** The timer will start at 0 counting up, it will generate interrupts if configured, and over flow as per normal. It can be configured so that the timer hardware can take control of a specific IO pin and alter its state according to configuration.
- **Phase Correct PWM:** The timer will start at zero, count up to the maximum value and then count back down to zero again, generating configured interrupts along the way. If configured the timer hardware can take control of a specific hardware pin.

6.1.6 Code Examples

- **Counter_Example1:** This goes through setting up a timer to have a particular prescale value and generate interrupts as well.
- **Counter_Example2:** This sets up a timer to do PWM[1]with a hardware pin and in software.
- **Counter_Example3:** This sets up a timer to debounce a switch on an interrupt pin.

[1]Pulse Width Modulation is a method for altering the average power delivered to a load by rapidly switching the load on and off. This is used for light dimming, speed control of motors . . .

Counter_Example1

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;Counter Example 1
2  ;
3  ; Remember to patch PortA0 to an LED
4  ; For you to do:
5  ;    - Setup T0 to generate an Ouput compare interrupt 0.01s after starting (use an output compare
       interrupt)
6  ;    - in your isr do not forget to stop, zero and restart your timer
7  ;    - increment a register, when it gets to 100 toggle the LED, the LED should now be on for 1sec
       and off for 1sec
8
9  .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
10
11 .NOLIST
12 .INCLUDE "m16def.inc"
13 .LIST
14
15 .def zero=R0          ; reserve R0 as our ZERO register
16 .def TMP1=R16         ;defines serve the same purpose as in C,
17 .def TMP2=R17         ;before assembly, defined values are substituted
18 .def TMP3=R18
19
20 .cseg                 ;Tell the assembler that everything below this is in the code segment
21 .org $000         ;locate code at address $000
22 jmp START         ; Jump to the START Label
23 .org OVF0addr     ; locate next instruction at Timer0's overflow interrupt vector
24 jmp T0_ISR        ; Jump to the interrupt service routine
25 .org $02A         ;locate code past the interupt vectors
26 START:
27      ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
28       out SPL, TMP1
29       ldi TMP1, HIGH(RAMEND)
30       out SPH, TMP1
31       RCALL INITIALISE_IO ; Call the subroutine INITIALISE
32       RCALL INIT_TIMER0
33      SEI ; enable global interrupts
34 MAIN_LOOP:
35       NOP
36       NOP
37       RJMP MAIN_LOOP
38
39
40 INITIALISE_IO:
41       SBI DDRA, 0               ; Make pin0 of portA output
42       CBI PORTB, 0         ; Set the initial state of pin to low=LED-off
43       RET                       ;Return from subroutine
44
45 ; We are going to start with something simple, we will initialise timer0 with the highest possible
       prescale value
46 ; which is /1024.  This means that the effective count rate will be ~8000 counts per second, which
       means that the
47 ; counter will overflow about 31 times per second, too fast to see, but bear with me.
48 ; So before we start we need to know what 'arcane values' to put in the gpio registers that control
        timer0
49 ; TIMSK=0b00000001 (this will enable the overflow interrupt for T0)
50 ; TCCR0=0b00000101 (all other functionality ignored, just setting the prescaler to /1024)
51 INIT_TIMER0:
52      ldi tmp1, 0b00000001
53      out TIMSK, tmp1 ;Timer0 overflow interrupt is now enabled
54      ldi tmp1, 0b00000101
55      out TCCR0, tmp1 ;Timer0 is now counting up with a prescaler of /1024
56      RET
57
58 ; To be pedantically correct we should be preseving the sreg, tmp1 and tmp2 on the stack, but I am
       tired of typing,
59 ; It will not affect us for now . . . .
60 T0_ISR:
61      in tmp1, porta ; read in the values that are currently being output in porta
62                     ; we do not want to read the pin values since then we would have to be carefull
63                     ; with masking values as we may enable/disable pullups
64      ldi tmp2, 0b00000001
65      eor tmp1, tmp2 ; toggle bit 0
66      out porta, tmp1 ; output back to porta
67      RETI
```

Code Explanation

- Note the positioning of the jumps to START and T0_ISR at the corresponding reset/interrupt vector locations.
- Note that in INITIALISE_IO there is an error. PORTB should be PORTA. Reading code properly is INCREDIBLY important.
- In line 52/3 we enable the overflow interrupt of Timer0 by setting TOIE0[a].
- In lines 54/55 we set the prescaler to a value of /1024[b]. At this point the timer is running.
- In the T0_OVF interrupt, PORTA0 is toggled by reading in the port values, toggling the relevant bit with a bitwise XOR and then outoutted back to PORTA.
- Can predict the frequency/period of the flashing of the LED as follows:

$$\begin{aligned} T_{counter} &= T_{CPU} * \text{PrescaleValue} \\ &= = 0.125\mu s \times 1024 \\ &= 128\mu s \end{aligned}$$

  So each count value takes 128 μs.
- The counter counts from 0 to 255, a total of 256 time periods (each taking 128 μs), giving a total time to over flow of:

$$T_{overflow} = 128\mu s \times 256 = 32ms$$

- So the light would be on for 32ms and then off for 32 ms, giving a peiod of 64 ms or 15 Hz, which should be visible as a flicker to the human eye.

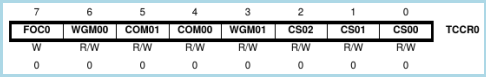
[a] page 85 of datasheet
[b] page 84/5 of datasheet - table 42

Things for you to try

- Setup T0 to generate an Ouput compare interrupt 0.01s after starting (use an output compare interrupt).
- In your ISR do not forget to stop, zero and restart your timer
- Increment a register, when it gets to 100 toggle the LED, the LED should now be on for 1 second and off for 1 second.

```
;Counter Example 2
; We are going to set up two LEDs to be pulse width modulated, one using the hardware
;   attached to the timer and one in software
; Remember to patch PortA0 to an LED
; Patch PB3 (OC0) to an LED as well
; For you to do: After you have seen and understood the effects of changing the OCR value try writing code that will
;                increase the brightness of one led on each press of a button. (If you get this done then you have
;                essentially made a brightness control for a piece of display electronics.)

.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
.NOLIST
.INCLUDE "m16def.inc"
.LIST

.def zero=R0          ; reserve R0 as our ZERO register
.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
jmp START        ; Jump to the START Label
.org OVF0addr     ; locate next instruction at Timer0's overflow interrupt vector
jmp T0ovf_ISR       ; Jump to the interrupt service routine
.org OC0addr    ; locate next instruction at Timer0's overflow interrupt vector
jmp T0oc_ISR       ; Jump to the interrupt service 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_IO ; Call the subroutine INITIALISE
      RCALL INIT_TIMER0
      ; we have to set a value in OCR0
      ldi tmp1, 0x10  ;Assemble with different values in here and make sure you can see the effect and explain it.
                      ; try 0x10, 0x40, 0x80, 0xb0, 0xd0
      out OCR0, tmp1
     SEI ; enable global interrupts
MAIN_LOOP:
      NOP
      NOP
      RJMP MAIN_LOOP

INITIALISE_IO:
      SBI DDRA, 0               ; Make pin0 of portA output
     SBI DDRB, 3          ; set pb3 to be output
      CBI PORTA, 0         ; Set the initial state of pin to low=LED-off
      CBI PORTB, 3         ; Set the initial state of pin to low=LED-off
      RET                       ;Return from subroutine

; We are going to start with something simple, we will initialise timer0 with a prescale value
; of /64.
; TIMSK=0b00000011 (this will enable the overflow interrupt and output compare interrupt for T0)
; TCCR0=0b00000101 (all other functionality ignored, just setting the prescaler to /64)
INIT_TIMER0:
     ldi tmp1, 0b00000011
     out TIMSK, tmp1 ;Timer0 overflow interrupt is now enabled
     ldi tmp1, 0b01111011 ; FAST PWM inverting - read the datasheet and your notes
     out TCCR0, tmp1 ;Timer0 is now counting up with a prescaler of /64
     RET

T0oc_ISR:
  cbi PORTA, 0
RETI

; To be pedantically correct we should be preseving the sreg, tmp1 and tmp2 on the stack, but I am tired of typing,
; It will not affect us for now . . . .
T0ovf_ISR:
   sbi PORTA,0
     RETI
```

Code Explanation

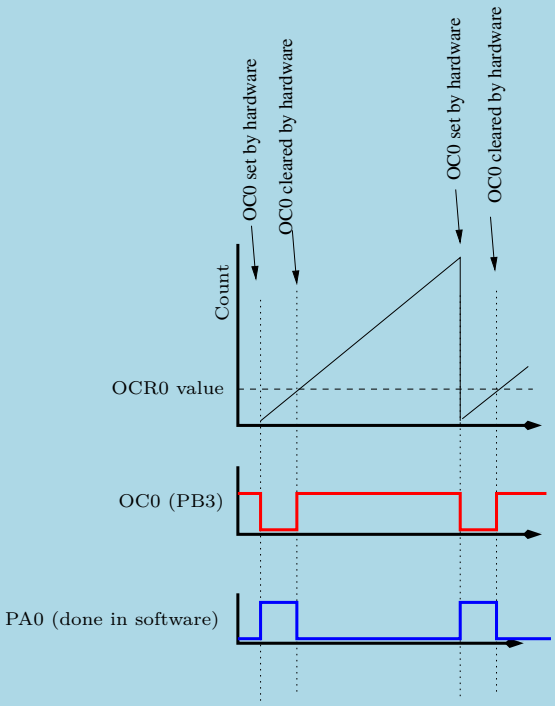
- In our initialising of IO (lines 49-54) we are setting both PA0 (the output for our *software* PWM) and PB3 (The hardware controlled pin).
- In the initialisation of timer0:
  - In TIMSK we enable both the overflow and output compare interrupts - this is so that the interrupts will be generated and we can manipulate PA0 in those interrupts to our hearts content.
  - 

7	6	5	4	3	2	1	0	
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

    We set TCCR0 to 0b01111011, WGM01 and WGM00 set the timer to Fast PWM mode, COM01 and COM00 (given the WGM01:0 settings) mean that the timer hardware takes control of the OC0 pin (PB3) and clears it (logical 0 outputted) on a compare match, and it is set (logical 1) at the bottom of the count. (From page 83-84 of datasheet.)
  - The prescaler is set to /64 - which actually starts the timer right then.
  - Our OCR0 value is set in the START: section directly after the timer initialisation returns.
- If we stop and do a quick calculation:

$$\begin{aligned} \text{Time for single count} &= \text{period system clock} \times \text{prescaler value} \\ &= 125\text{ ns} \times 64 \\ &= 8\mu\text{s} \end{aligned}$$

- So if we look at the diagram below and we assume that OCR0 has the value 0x10 (16) in it, OC0 will be low for 128 μs and high for 1912μs, i.e. it is on for most of the time.
- PA0, is the exact opposite.



So all we have to do is:

- Alter this code to setup INT0 to be triggered by a button (Don't forget to connect them).
- In the INT0 interrupts we need to:
  - Read the value of OCR0 into a register.

- Increment that register by a set value (I would recommend doing a simple multiply by two first to check - easy multiply by two is just a lsl. This will break when the one falls of the end, but it is OK for initial testing - you can do proper addition later. You could always just have 16 INC operations on the register. Remember don't try to be fancy all at once - after you know things are working properly you can make it look elegant and decent enough to show to other people).
- Output this register back to OCR0.

Things for you to try

- Change your code and hardware (i.e. use a second button on INT1) so that pressing one button makes an LED brighter and the other one makes the LED dimmer.
- Try this using both software PWM and the hardware OC0 pin for the timer.

Counter Example3

```
;Counter Example 3 - debouncing a switch
; For you to do * Make sure you understand what is going on.

.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place

.NOLIST
.INCLUDE "m16def.inc"
.LIST

.def zero=R0        ; reserve R0 as our ZERO register
.def count=R1
.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
jmp START       ; Jump to the START Label
.org INT0addr
jmp int0isr
.org OVF0addr    ; locate next instruction at Timer0's overflow interrupt vector
jmp T0ovf_ISR       ; Jump to the interrupt service 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_IO ; Call the subroutine INITIALISE
     ; we have to set a value in OCR0
    SEI ; enable global interrupts
MAIN_LOOP:
     NOP
     NOP
     RJMP MAIN_LOOP

INITIALISE_IO:
ldi tmp1, 0x0F;
out DDRA, tmp1;
clr count
sbi portd, 2 ; pullup on pd2 pin for int 0
ldi tmp1, 0b01000000
out GICR, tmp1 ; int0 enabled
ldi tmp1, 0b00000010
out mcucr, tmp1 ;falling edge triggered.
     RET                        ;Return from subroutine

; On overflow of timer we reinit int0, after clearing the flags, stop the timer.
T0ovf_ISR:
 out TCCR0, zero ; stop counter
 out TCNT0, zero ; zero counter
 ldi tmp1, 0b01000000
 out GIFR, tmp1 ; clear int0 flag
                ; Remove this line and watch the switch bounce "come back" because the flag is set
                    when
               ; the bounce occurs and if we don't clear it, when we re-enable int0 the flag will
                   cause an interrupt
ldi tmp1, 0b01000000
out GICR, tmp1 ; int0 re-enabled
    RETI

int0isr:
inc count ; increment out variable
out porta, count
;uncomment the code section below to enable our switch debouncing
;out gicr, zero ; diable external interrupts
;ldi tmp1, 0x01
;out timsk, tmp1 ; enable t0 overflow
;ldi tmp1, 0b00000101
;out tccr0, tmp1 ; start timer0 with prescaler set to /1024
;---end of debouncing
   RETI
```

Code Explanation

Things for you to try

- Make sure you can see the effects of switch bounce.
- Try using different times for debouncing to see if you can work out how long switch bounce carries on for...

6.2 Using 16bit registers

IMPORTANT

- Many of the peripherals on the mega16 make use of 16-bit IO registers, the concatenation of 2 8-bit IO registers.
- Precautions are neccesary when using the 16-bit IO registers. See chapter 36 on page 163 of the notes. This warning is also repeated in the Mega16 datasheet when peripherals use 16bit IO registers.
- Ignore this at your own peril:
  - Writing: The High byte must be written first and then the low byte.
  - Reading: The Low byte must be read first and then the high byte.
  - Read the datasheet / notes to find out why. #pastexamhint

6.3 Task 5 - Using timers to time events

What to read through beforehand

- The section on Timer/counters

The Code

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.

```
; wire switches 0-1 to portd0-1
; wire leds0-1 to porta0-1
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST
.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

```

```
63 INITIALISE_EXTERNAL_INTERRUPTS:
64       ;enable  int0
65       ldi tmp1, 0x40
66       out GICR, tmp1
67       ; interrupt on falling edge.
68       ldi tmp1, 0x0a
69       out MCUCR, tmp1
70       ret
71
72 INT0_ISR:
73       ;wait a while and then light the test led
74 ;disable int 0 and enable int1
75       ldi tmp1, 0x80
76       out GICR, tmp1
77       sbi porta,0
78      ldi tmp1, 0x60
79 loop1: ser tmp2
80 loop2: ser tmp3
81 loop3: ;
82       dec tmp3
83       cpi tmp3, 0
84       BRNE loop3
85       dec tmp2
86       BRNE loop2
87       dec tmp1
88       BRNE loop1
89       ;we have now finished a bit of a delay.
90       sbi porta, 1 ;turn led on
91       cbi porta,0
92       ;start timer going
93       ldi tmp1, 0x05
94       out tccr1b, tmp1
95       reti
96
97 INT1_ISR:
98 ; stop timer1
99       ldi tmp1, 0x00
100      out tccr1b, tmp1
101 ;disable int 1 and enable int0
102      ldi tmp1, 0x40
103      out GICR, tmp1
104 ;read both timer bytes and display (we will only display the high byte)
105     IN tmp2, TCNT1L
106      IN tmp1, TCNT1H
107      out PORTA,tmp1
108      reti
109
110 TIMER1_OVF_ISR:
111 ; too slow
112 ldi tmp1, 0x00
113      out tccr1b, tmp1
114      out PORTA, tmp1
115      reti
```

To Do

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.

6.4 Task 6 - Generating a 'Random' Number

What to read through beforehand

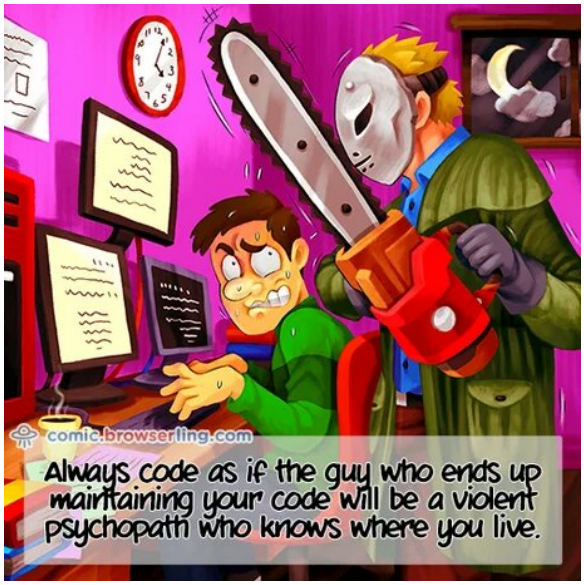
- The section on Timer/counters

To Do

1. Write code that will display a pseudo-random number on the LEDs when a button is pressed. (HINT: Consider the value in a counter that is running very fast.)

CHAPTER

7 Introduction to the Stepper Motor



Stuff to read:

- Make sure you have read all the preceding sections and relevant sections in the Mega16 datasheet.

Videos (on RUConnected) to watch:

- Video12-StepperMotors

7.1 Stepper Motor

7.1.1 So what is a stepper motor?

- Well you can read https://en.wikipedia.org/wiki/Stepper_motor
- What this means for us is we need to be able to control a stepper motor and this is going to help introduce some other concepts - mainly because flashing lights get a bit old, but now we can control a physical object.
- The steppers that we use all came from old $5\frac{1}{4}$ inch floppy drives, but today you would find them mostly in 3D printers. As a matter of fact alot of the low-end 3D printers use microcontrollers from the Atmel Mega series. (In theory if you had a chassis, by the end of the course you could code the firmware for a printer)

7.1.2 Driving Hardware

We drive the windings of our stepper motor using a MOSFET (n-channel enhancement mode) since the current requirements of the stepper push the limits of what the microcontroller can supply. See page 177 of your notes for the schematic.

7.1.3 Moving the stepper

- The motion of the stepper is controlled by the sequence in which you energise the windings. (The sequence controls the direction).
- The speed of the stepper depends on how often you change the pattern that you are driving with.
- If the DRV0 .. DRV3 are driven with the following bit pattern:

  . . .
  0001
  0010
  0100
  1000
  0001
  0010
  . . .

- We would sequentially pull the stator to the next winding thereby causing the motor to step.
- If we stepped through the sequence in the opposite order the motor would turn in the other direction.

7.1.4 Code Examples

- **Stepper_Example1:** Single stepping the stepper on a button push.
- **Stepper_Example2:** Automatic stepping using a timer.

Stepper_Example1

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;Stepper Example 1
2  ; Hardware connections.
3  ; pd4->drv0
4  ; pd5->drv1
5  ; pd6->drv2
6  ; pd7->drv3
7  ; pd3 -> button0
8  ; pd2 - button1
9  ; We are going to start with a modified example of our debounced external interrupt
10 ; code from the previos example.
11 ; For you to do
12 ;  * Make sure you understand what is going on
13 ;  * Alter the code to back the stepper in the other direction. - maybe making 2 subroutines,
       turnleft and turnright.
14 ;  * make int0 turn the motor one way and int1 turn the motor the other way.
15 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
16 .NOLIST
17 .INCLUDE "m16def.inc"
18 .LIST
19
20 .def zero=R0          ; reserve R0 as our ZERO register
21 .def step=R1
22 .def TMP1=R16         ;defines serve the same purpose as in C,
23 .def TMP2=R17         ;before assembly, defined values are substituted
24 .def TMP3=R18
25
26 .cseg                 ;Tell the assembler that everything below this is in the code segment
27 .org $000        ;locate code at address $000
28 jmp START        ; Jump to the START Label
29 .org INT0addr
30 jmp int0isr
31 .org OVF0addr    ; locate next instruction at Timer0's overflow interrupt vector
32 jmp T0ovf_ISR       ; Jump to the interrupt service routine
33
34 .org $02A        ;locate code past the interupt vectors
35 START:
36     ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
37      out SPL, TMP1
38      ldi TMP1, HIGH(RAMEND)
39      out SPH, TMP1
40      RCALL INITIALISE_IO ; Call the subroutine INITIALISE
41      ; we have to set a value in OCR0
42     SEI ; enable global interrupts
43 MAIN_LOOP:
44      NOP
45      NOP
46      RJMP MAIN_LOOP
47
48 INITIALISE_IO:
49 ldi tmp1, 0xF0;
50 out DDRD, tmp1;
51 ldi tmp1, 0x10
52 mov step, tmp1
53 sbi portd, 2 ; pullup on pd2 pin for int 0
54 ldi tmp1, 0b01000000
55 out GICR, tmp1 ; int0 enabled
56 ldi tmp1, 0b00000010
57 out mcucr, tmp1 ;falling edge triggered.
58      RET                        ;Return from subroutine
59
60 ; On overflow of timer we reinit int0, after clearing the flags, stop the timer.
61 T0ovf_ISR:
62  out TCCR0, zero ; stop counter
63  out TCNT0, zero ; zero counter
64  ldi tmp1, 0b01000000
65  out GIFR, tmp1 ; clear int0 flag
66                  ; Remove this line and watch the switch bounce "come back" because the flag is set
                        when
67                 ; the bounce occurs and if we don't clear it, when we re-enable int0 the flag will
                       cause an interrupt
68 ldi tmp1, 0b01000000
69 out GICR, tmp1 ; int0 re-enabled
70     RETI
71
```

```
72 int0isr:
73 call nextstep
74 out gicr, zero ; diable external interrupts
75 ldi tmp1, 0x01
76 out timsk, tmp1 ; enable t0 overflow
77 ldi tmp1, 0b00000101
78 out tccr0, tmp1 ; start timer0 with prescaler set to /1024
79    RETI
80
81 NEXTSTEP:
82 IN tmp1, PORTD
83 andi tmp1, 0x0F ;tmp now contains the masked off values of portD
84 lsl step ; increment out variable
85 brne outstep
86 ldi tmp2, 0x10
87 mov step, tmp2
88 outstep:
89   or tmp1, step
90   out portd, tmp1
91   RET
```

Code Explanation

- Initialise_IO:
  - We start off bysetting the upper 4 bits of PORTD to OUTPUT and then setting our 'STEP' register (general purpose working register as defined in line 21).[a]
  - We then enable the pull-up resistor on PD2 for interrupt 0.
  - Lines 54-55 enable the INT0 interrupt.[b]
  - Lines 56-57 set INT0 to be falling triggered.[c]
- T0ovf_ISR:
  - Lines 62-4 The timer is stopped and reset.
  - Lines 64-5 INT0 flag is cleared.
  - Lines 68-9 INT0 is re-enabled.
  - Following same approach as debouncing buttons when looking at debouncing switches earlier.
- INT0_ISR:
  - Line 73 - The next step is output (see below).
  - Line 74 - External Interrupt is disabled.
  - Line 75-8 - Timer overflow interrupt is setup and timer started.
- NEXTSTEP:
  - Lines 82-3 - Current state of PORTD is read in and masked.
  - Lines 84-7 - Mask is moved and reset if needed.
  - Lines 89-90 - Mask is ORed with PORD's original state and then output.

[a] IO Ports datasheet page 50
[b] GICR - datasheet page 69
[c] MCUCR - datasheet page 68/9 table 35

Things for you to try

- Alter the code to make the stepper turn in the other direction. - maybe making 2 subroutines, turnleft and turnright.
- Now make INT0 turn the motor one way and INT1 turn it in the other direction.

Stepper_Example2

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;Stepper Example 2
2  ; Hardware connections.
3  ; pd4->drv0
4  ; pd5->drv1
5  ; pd6->drv2
6  ; pd7->drv3
7  ; pd3 -> button0
8  ; pd2 - button1
9  ; We are going to start with a modified example of our previous stepper motor code
10 ; For you to do
11 ;  * Make sure you understand what is going on
12 ;  * Alter the code to have int0 start or stop the motor.
13 ;  * Alter the code so that int1 will add 0x10 to the oc0 value to step through speeds
14 ;  *  (and wrap around to a small value again)
15 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
16
17 .NOLIST
18 .INCLUDE "m16def.inc"
19 .LIST
20
21 .def zero=R0        ; reserve R0 as our ZERO register
22 .def step=R1
23 .def TMP1=R16       ;defines serve the same purpose as in C,
24 .def TMP2=R17       ;before assembly, defined values are substituted
25 .def TMP3=R18
26
27 .cseg               ;Tell the assembler that everything below this is in the code segment
28 .org $000       ;locate code at address $000
29 jmp START       ; Jump to the START Label
30 .org INT0addr
31 jmp int0isr
32 .org OVF0addr    ; locate next instruction at Timer0's overflow interrupt vector
33 jmp T0ovf_ISR      ; Jump to the interrupt service routine
34 .org OC0addr
35 jmp T0OC_ISR
36
37 .org $02A       ;locate code past the interupt vectors
38 START:
39     ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
40      out SPL, TMP1
41      ldi TMP1, HIGH(RAMEND)
42      out SPH, TMP1
43      CALL INITIALISE_IO ; Call the subroutine INITIALISE
44      call initialise_t0 ; call subroutine to init t0 with output comparevalue
45     SEI ; enable global interrupts
46 MAIN_LOOP:
47      NOP
48      NOP
49      RJMP MAIN_LOOP
50
51 Initialise_t0:
52     ldi tmp1, 0x60 ; try a range of values here and observe the motor speed. As an exercise
           determine the # of steps per
53                     ; second that the motor is performing for each value
54     out ocr0, tmp1 ; setup oc0
55     ldi tmp1, 0b00000010
56     out timsk, tmp1 ; enable t0 ocint - we use the output compare intterupt in this case
57                     ; because we will set the timer to CTC mode
58     ldi tmp1, 0b00001101; ctc mode, no OC pin use, /1024 prescaler
59     out TCCR0, tmp1
60     ret
61
62 INITIALISE_IO:
63     ldi tmp1, 0xF0;
64     out DDRD, tmp1;
65     ldi tmp1, 0x10
66     mov step, tmp1
67     sbi portd, 2 ; pullup on pd2 pin for int 0
68     ldi tmp1, 0b01000000
69     out GICR, tmp1 ; int0 enabled
70     ldi tmp1, 0b00000010
71     out mcucr, tmp1 ;falling edge triggered.
72     RET                        ;Return from subroutine
73
```

```
74 ; On overflow of timer we reinit int0, after clearing the flags, stop the timer.
75 T0ovf_ISR:
76 ;call Nextstep
77     RETI
78 T0OC_ISR:
79 call Nextstep
80     RETI
81
82 int0isr:
83 call nextstep
84    RETI
85
86 NEXTSTEP:
87 IN tmp1, PORTD
88 andi tmp1, 0x0F ;tmp now contains the masked off values of portD
89 lsl step ; increment out variable
90 brne outstep
91 ldi tmp2, 0x10
92 mov step, tmp2
93 outstep:
94   or tmp1, step
95   out portd, tmp1
96   RET
```

Code Explanation

- Initialise_t0:
  - Lines 52-4 : The output compare register is initialised to a non-zero value.
  - Lines 55-6 : The output compare interrupt on TC0 is enabled.
  - Lines 58-9 : TC0 is started with a prescaler or /1024 in CTC mode (Timer clears on compare match).
- INITIALISE_IO:
  - As in the previous example the IO on PORTD is set up.
  - INT0 is also enabled (for no particular reason).
- T0OC_ISR:
  - The Next step is called on each output compare.
- The code is set up in this manner so that you can easily check the behaviour if the over flow interrupt was used instead.

Things for you to try

- Alter the code to have int0 start or stop the motor.
- Alter the code so that int1 will add 0x10 to the oc0 value to step through speeds (and wrap around to a small value again)

CHAPTER

8 The ADC

Stuff to read:

- Read over the secions on the Analog Comparator (page 140) and the Analog to Digital Converter (page 132)

Videos (on RUConnected) to watch:

- Video13-ADC

8.1 Analog to Digital Converter

8.1.1 Overview

- The ADC allows us to convert real-world analog voltages to a digital representation. These analog voltages can represent temperature, air pressure, light or any other real-world quantity that can be detected by suitable electronics.
- The ADC in the Mega16 (and most other microcontrollers) is a 10-bit successive approximation ADC. https://en.wikipedia.org/wiki/Successive_approximation_ADC

- The internals are not of much interest, but we do need to know how to use it.

8.1.2 ADC Operation

- First the ADC must be enabled, note the overheads of enabling the ADC (see data sheet).
- The ADC has multiple sources. (p63 of notes)
- The ADC can run off of a prescaled clock. A lower clock frequency = more accurate conversion. (See page 33 for more accurate methods).
- We can either manually trigger a conversion, have the conversion triggered by an event (p64/65 in notes) or have it in free-running mode (also p64/65 in notes).
- When a conversion is completed we can have an interrupt triggered.

8.1.3 Code Examples

- **ADC_Example1:** Single ADC Conversion.
- **ADC_Example2:** Free-running ADC Conversion.
- **ADC_Example3:** ADC Controlled Stepper.

ADC_Example1

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;ADC Example 1
2  ; Hardware connections.
3  ; Some LEDS to show the results of our conversion
4  ;PC0-led0
5  ;PC1-led1
6  ;PC2-led2
7  ;PC3-led3
8  ;PC4-led4
9  ;PC5-led5
10 ;PC6-led6
11 ;PC7-led7
12 ;To trigger our conversion
13 ;PD2-sw0
14 ;PA0 to VAR , the output of our potentiometer so we can vary the voltage to digitise
15
16 ;-------------------------------------------------------------------------------
17 ; todo:  Turn the pot and trigger conversions, note the binary output of the conversion.
18 ; * Change the code so that ADLAR is not used.  Read in both ADL & ADH and format the output
19 ; so that it is the same as before.
20 ; * Change your code again so that the lower 8 bits are now displayed.  Note that sometimes (when
       the potentiometer
21 ;   is not moved you will get different results - the AD conversion is not always exact.
22 ;   Change the prescaler settings to make the conversion more inaccurate.)
23
24 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
25
26 .NOLIST
27 .INCLUDE "m16def.inc"
28 .LIST
29
30 .def zero=R0          ; reserve R0 as our ZERO register
31 .def TMP1=R16         ;defines serve the same purpose as in C,
32 .def TMP2=R17         ;before assembly, defined values are substituted
33
34 .cseg                 ;Tell the assembler that everything below this is in the code segment
35 .org $000       ;locate code at address $000
36 jmp START       ; Jump to the START Label
37 .org INT0addr
38 jmp int0isr
39 .org ADCCaddr    ; locate next instruction at Timer0's overflow interrupt vector
40 jmp ADC_ISR       ; Jump to the interrupt service routine
41
42 .org $02A       ;locate code past the interupt vectors
43 START:
44      ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
45       out SPL, TMP1
46       ldi TMP1, HIGH(RAMEND)
47       out SPH, TMP1
48       RCALL INITIALISE_IO ; Call the subroutine INITIALISE
49      SEI ; enable global interrupts
50 MAIN_LOOP:
51       NOP
52       NOP
53       RJMP MAIN_LOOP
54
55
56 INITIALISE_IO:
57      ldi tmp1, 0xFF; Set porta as output to show the results of our conversion
58      out DDRC, tmp1;
59      sbi portd, 2 ; pullup on pd2 pin for int 0
60      ldi tmp1, 0b01000000
61      out GICR, tmp1 ; int0 enabled
62      ldi tmp1, 0b00000010
63      out mcucr, tmp1 ;falling edge triggered.
64 RET                        ;Return from subroutine
65
66 ADC_ISR:
67  ;output conversion result
68  in tmp1, ADCH
69  out portc, tmp1
70  ;Clear int0 flags
71  ldi tmp1, 0b01000000
72  out GIFR, tmp1 ; int0 enabled
73  ;re-enable int0
```

```
74  out GICR, tmp1 ; int0 enabled
75     RETI
76
77 int0isr:
78 ;SBI portc, 0
79 call Start1conversion
80 out gicr, zero ; diable external interrupts
81    RETI
82
83 Start1conversion:
84     ldi tmp1, 0b01100000 ; AVCC selected as reference (limitation of wiring on board),
85                          ; ADLAR set so that the most significant 8 bits are in ADCH read datasheet
                                p207-225 and make sure
86                          ; you are familiar with the ADC) and ADMUX set so that ADC0 is the input to
                                the ADC
87     OUT ADMUX, tmp1
88     ldi tmp1, 0b11001111      ; ADC enabled, conversion started, no auto trigger, interrupt enabled
          and prescaler set to /128
89     OUT ADCSRA, tmp1
90      ; so the clock to the adc is set to 8MHz/128=62.5kHz
91      ;  The first conversion will take 25 clock cycles (see page 211 of data sheet) so the
          conversion will take 0.4ms
92      RET
```

Code Explanation

- INITIALISE_IO:
  - PORTC is initialised as output and INT0 is enabled on a falling edge (as in previous examples).
- ADC_ISR:
  - Lines 68-9 : The ADCH register is read in and then output to PORTC for display on the LEDS.
  - Lines 70-4 : INT0 flags are cleared and then INT0 is re-enabled.
- int0isr:
  - A conversion is initiated and then INT0 is disabled.
  - You may have noticed that if we incorporated the actions in Start1Conversion into the ISR for INT0 we would be saving some code space as well as cutting out the overhead of the call instruction and the associated RET and you would be perfectly correct (if you spotted this then well done you are thinking how embedded programmers often have to make things *un-pretty* to meet execution deadlines, but at this stage (and this example) that would be counter-productive.
- Start1conversion:
  - Lines 84-7 : The reference voltage for the ADC is selected as well as selecting which pin of PORTA is used as the input.[a]
  - Lines 88-91 : The ADC is enabled, single conversion started with prescaler set to /128.[b]

[a] ADMUX - datasheet page 217, table 83& 84
[b] ADCSRA - datasheet page 219-220, table 85

Things for you to try

- Change the code so that ADLAR is not used. Read in both ADL & ADH and format the output so that it is the same as before. (Yes - I know it is a fair bit of code that can be done much more efficiently as given, trust me on this.)
- Change your code again so that the lower 8 bits are now displayed. Note that sometimes (when the potentiometer is not moved you will get different results) - the AD conversion is not always exact.
- Change the prescaler settings to make the conversion more inaccurate.Read over the section on *Noise Cancelling* in the ADC section of the datasheet. #pastexamhint

ADC_Example2

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;ADC Example 2
2  ; Hardware connections.
3  ; Some LEDS to show the results of our conversion
4  ;PC0-led0
5  ;PC1-led1
6  ;PC2-led2
7  ;PC3-led3
8  ;PC4-led4
9  ;PC5-led5
10 ;PC6-led6
11 ;PC7-led7
12 ;
13 ;PA0 to VAR , the output of our potentiometer so we can vary the voltage to digitise
14
15 ;-------------------------------------------------------------------------------
16 ; todo:  Make the free running conversion trigger from Timer1's compare match interrupt and
17 ;        get the compare match interrupt to trigger every 0.5s - you might want to start by getting
18 ;        T1 to generate this interrupt correctly and text it by flashing an LED to make sure things
       are
19 ;        working as desired.  - Remember is is FAR easier to test little things by themselves
      before you
20 ;        complicate your code.
21
22 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
23
24 .NOLIST
25 .INCLUDE "m16def.inc"
26 .LIST
27
28 .def zero=R0          ; reserve R0 as our ZERO register
29 .def TMP1=R16         ;defines serve the same purpose as in C,
30 .def TMP2=R17         ;before assembly, defined values are substituted
31
32 .cseg                 ;Tell the assembler that everything below this is in the code segment
33 .org $000         ;locate code at address $000
34 jmp START         ; Jump to the START Label
35 .org INT0addr
36 jmp int0isr
37 .org ADCCaddr    ; locate next instruction at Timer0's overflow interrupt vector
38 jmp ADC_ISR       ; Jump to the interrupt service routine
39
40 .org $02A         ;locate code past the interupt vectors
41 START:
42      ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
43       out SPL, TMP1
44       ldi TMP1, HIGH(RAMEND)
45       out SPH, TMP1
46       RCALL INITIALISE_IO ; Call the subroutine INITIALISE
47      call Startfreerunningconversion
48      SEI ; enable global interrupts
49 MAIN_LOOP:
50       NOP
51       NOP
52       RJMP MAIN_LOOP
53
54
55 INITIALISE_IO:
56      ldi tmp1, 0xFF; Set porta as output to show the results of our conversion
57      out DDRC, tmp1;
58      sbi portd, 2 ; pullup on pd2 pin for int 0
59      ldi tmp1, 0b01000000
60      out GICR, tmp1 ; int0 enabled
61      ldi tmp1, 0b00000010
62      out mcucr, tmp1 ;falling edge triggered.
63 RET                        ;Return from subroutine
64
65
66
67 ; On overflow of timer we reinit int0, after clearing the flags, stop the timer.
68 ADC_ISR:
69  ;output conversion result
70  in tmp1, ADCH
71  out portc, tmp1
72     RETI
```

```
73
74 int0isr:
75    RETI
76
77 Startfreerunningconversion:
78     ldi tmp1, 0b01100000 ; AVCC selected as reference (limitation of wiring on board),
79                          ; ADLAR set so that the most significant 8 bits are in ADCH read datasheet
                                p207-225 and make sure
80                          ; you are familiar with the ADC) and ADMUX set so that ADC0 is the input to
                                the ADC
81     OUT ADMUX, tmp1
82     ldi tmp1, 0b11101111      ; ADC enabled, conversion started, no auto trigger, interrupt enabled
            and prescaler set to /128
83     OUT ADCSRA, tmp1
84      ; so the clock to the adc is set to 8MHz/128=62.5kHz
85      ;  The first conversion will take 25 clock cycles (see page 211 of data sheet) so the
            conversion will take 0.4ms
86      RET
```

Code Explanation

- This code is basically the same as the first example except that a free running conversion is started and then the ADC interrupt just contiuously updates the output after each conversion.

Things for you to try

- Change the code so that conversions are triggered from Timer1's Compare Match Interrupt. Make this trigger every 0.5 seconds. (You might want to start by getting Timer1 to generate this interrupt correctly and test it by flashing an LED
- HINT: To make your life easier look at page 221 of the datasheet.

ADC_Example3

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;ADC Example 3
2  ; Hardware connections.
3  ; Some LEDS to show the results of our conversion for debugging
4  ;PC0-led0
5  ;PC1-led1
6  ;PC2-led2
7  ;PC3-led3
8  ;PC4-led4
9  ;PC5-led5
10 ;PC6-led6
11 ;PC7-led7
12
13 ;PD7-DRV3
14 ;PD6-DRV2
15 ;PD5-DRV1
16 ;PD4-DRV0
17
18
19 ;PA0 to VAR , the output of our potentiometer so we can vary the voltage to digitise
20
21 ;-------------------------------------------------------------------------------
22 ; todo:  NOTE if you try to drive the motor too fast, you will just get a high-pitched whine from
       it
23 ;        * Can you alter the code so that the speed of the morot only updates when int0 is pressed
24 ;          (there is an easy way and an easier way)
25
26 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
27
28 .NOLIST
29 .INCLUDE "m16def.inc"
30 .LIST
31
32 .def zero=R0         ; reserve R0 as our ZERO register
33 .def step=R1
34 .def TMP1=R16        ;defines serve the same purpose as in C,
35 .def TMP2=R17        ;before assembly, defined values are substituted
36
37 .cseg                ;Tell the assembler that everything below this is in the code segment
38 .org $000       ;locate code at address $000
39 jmp START       ; Jump to the START Label
40 .org OC0addr
41 jmp OC0_isr
42 .org ADCCaddr    ; locate next instruction at Timer0's overflow interrupt vector
43 jmp ADC_ISR       ; Jump to the interrupt service routine
44
45 .org $02A       ;locate code past the interupt vectors
46 START:
47     ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
48      out SPL, TMP1
49      ldi TMP1, HIGH(RAMEND)
50      out SPH, TMP1
51      RCALL INITIALISE_IO ; Call the subroutine INITIALISE
52      call initialise_t0 ; call subroutine to init t0 with output comparevalue
53     call Startfreerunningconversion
54     SEI ; enable global interrupts
55 MAIN_LOOP:
56      NOP
57      NOP
58      RJMP MAIN_LOOP
59
60
61 INITIALISE_IO:
62     ldi tmp1, 0xF0; Set porta as output to show the results of our conversion
63     out DDRD, tmp1;
64     sbi portd, 2 ; pullup on pd2 pin for int 0
65     ldi tmp1, 0x10
66     mov step, tmp1
67     ldi tmp1, 0b01000000
68     out GICR, tmp1 ; int0 enabled
69     ldi tmp1, 0b00000010
70     out mcucr, tmp1 ;falling edge triggered.
71 RET                      ;Return from subroutine
72
73 Initialise_t0:
```

```
74      ldi tmp1, 0xff ; try a range of values here and observe the motor speed. As an exercise
            determine the # of steps per
75                     ; second that the motor is performing for each value
76      out ocr0, tmp1 ; setup oc0
77      ldi tmp1, 0b00000010
78      out timsk, tmp1 ; enable t0 ocint - we use the output compare intterupt in this case
79                      ; because we will set the timer to CTC mode
80      ldi tmp1, 0b00001101; ctc mode, no OC pin use, /1024 prescaler
81      out TCCR0, tmp1
82      ret
83
84  ; On overflow of timer we reinit int0, after clearing the flags, stop the timer.
85  ADC_ISR:
86  ;our conversion result goes into OC0
87   in tmp1, adch
88   out ocr0, tmp1
89      RETI
90
91  OC0_isr:
92     CALL NEXTSTEP
93     RETI
94
95  Startfreerunningconversion:
96      ldi tmp1, 0b01100000 ; AVCC selected as reference (limitation of wiring on board),
97                           ; ADLAR set so that the most significant 8 bits are in ADCH read datasheet
                                 p207-225 and make sure
98                           ; you are familiar with the ADC) and ADMUX set so that ADC0 is the input to
                                 the ADC
99      OUT ADMUX, tmp1
100     ldi tmp1, 0b01100000 ;free running mode to trigger from T0 compare match
101     out SFIOR, tmp1
102     ldi tmp1, 0b11101111       ; ADC enabled, conversion started, no auto trigger, interrupt enabled
            and prescaler set to /128
103     OUT ADCSRA, tmp1
104      RET
105 NEXTSTEP:
106 IN tmp1, PORTD
107 andi tmp1, 0x0F ;tmp now contains the masked off values of portD
108
109 lsl step ; increment out variable
110 brne outstep
111 ldi tmp2, 0x10
112 mov step, tmp2
113
114 outstep:
115   or tmp1, step
116   out portd, tmp1
117   RET
```

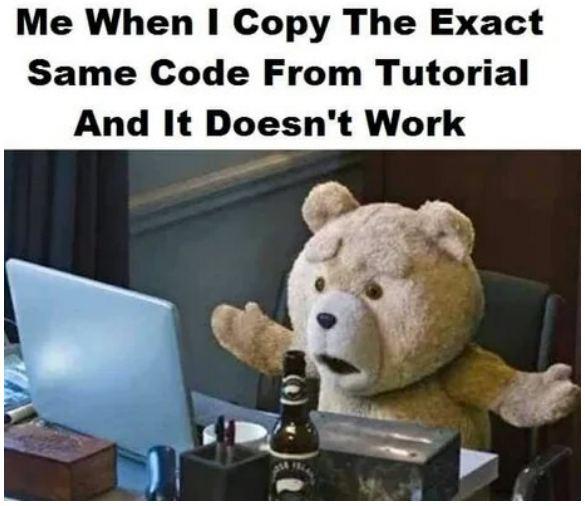
Code Explanation

Things for you to try

- Using the free running ADC conversions, (maybe incorporating your stepper routines from previous work) make the variable voltage control the speed of the stepper motor.

CHAPTER

9 Using the LCD



Stuff to read:

Videos (on RUConnected) to watch:

- Video16-LCDIntro
- Video17-LCDDriver

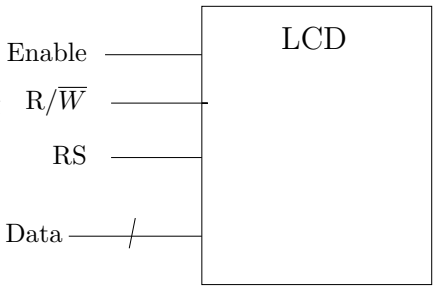
9.1 Liquid Crystal Display

9.1.1 Overview

- Refer to LCD.asm in Appendix B (task 10 has the .asm file as well as the examples)
- Refer to the abridged datasheet for the LCD in Appendix B.
- This is a common interface for parrallel (as opposed to I^2C and other serial interfaces).
- The LCD has an 8-bit wide data interface with several control lines.

9.1.2 LCD Operation

- The Enable line is rising edge active, i.e. the LCD responds to instructions/data and control lines on the rising edge (see page 198 of notes).
- The R/S line is a signal line indicating whether data is being sent or an instruction is being sent.
- The R/$\overline{\text{W}}$ line indicates whether data is being read from the LCD or written to it.
- the 8bit parallel data lines allow for transfer of data to and from the LCD.



9.1.3 Drivers in LCD.asm

Write_instruc

This procedure writes the instruction passed to it in R16 to the LCD.

- R17 is preserved on the stack - we don't bother preserving the SREG because it is highly unlikely that you would be writing to the LCD while performing an arithmetic operation that you wanted to use in a branch or loop.
- Checks to see if the LCD is free - this is a blocking IO operation - it olny returns when free.
- The R/$\overline{\text{W}}$ is taken low, signalling a write to the LCD.
- The RS line is taken low - signalling the data to be sent is an instruction.
- PORTC (the port we use for the data transmission) is set as an output.
- The data byte is written to PORTC.
- The ENABLE line is pulse high, causing the LCD to read the instruction on the data lines.
- PORTC is made input again.
- R17 is restored from the stack.

Write_char

This procedure writes byte passed to it in R16 to the LCD as data.

- R17 is preserved on the stack - we don't bother preserving the SREG because it is highly unlikely that you would be writing to the LCD while performing an arithmetic operation that you wanted to use in a branch or loop.
- Checks to see if the LCD is free - this is a blocking IO operation - it olny returns when free.
- The R/$\overline{\text{W}}$ is taken low, signalling a write to the LCD.
- The RS line is taken high - signalling the data to be sent is a character.
- PORTC (the port we use for the data transmission) is set as an output.
- The data byte is written to PORTC.
- The ENABLE line is pulse high, causing the LCD to read the instruction on the data lines.
- PORTC is made input again.
- R17 is restored from the stack.

Init_LCD

This procedure configures the LCD - no options are passed to it so all configuration changes must be done in the file or manually.

- R16 is preserved on the stack - we don't bother preserving the SREG because it is highly unlikely that you would be configuring the LCD while performing an arithmetic operation that you wanted to use in a branch or loop.
- PORTC is set to input.
- The lines attached to RS, R/W and enable are made outputs
- The following instructions are written to the LCD:
  - 0x38: 0b00111000 in binary. Using the datasheet extract on page 154 of the notes, this corresponds to a *Function Set* instruction where DL=1, sets the data length to 8-bits (4-bits can be used if you are short of IO lines but we are going to use the LCD exclusively in 8-bit mode), N=1 sets the LCD to 2-line mode and F=0 sets the font to 5x7 dots.
  - 0x0C: 0b00001100 in binary. This corresponds to a *Display Control* instruction where D=1 sets the Display to be On, C=0, turns the cursor off, B=0 sets the blinking of the cursor off (if the cursor was being shown).
  - 0x06: 0b00000110 in binary. This correxponds to an *Entry Mode Set* instruction where I/D=1 will increment the cursor position for each sequential character and S=0 means that the cursor will move rather than the display shifting.
  - 0x01: 0b00000001 in binary. This instruction clears the display and sets the Cursor to the home position (address 0x00 - see diagram at bottom of page 154 for character address layout).

Connections

The drivers assume the following connections for the LCD:

- PC0 → D0 on the LCD connector
- PC1 → D1 on the LCD connector
- PC2 → D2 on the LCD connector
- PC3 → D3 on the LCD connector
- PC4 → D4 on the LCD connector
- PC5 → D5 on the LCD connector
- PC6 → D6 on the LCD connector
- PC7 → D7 on the LCD connector
- Please note the white label on PORTC - PORTC connections are in the exact opposite order to the other ports.
- PB0 → E on the LCD connector
- PB1 → R/W on the LCD connector
- PB2 → RS on the LCD connector
- CON(trast) on the LCD connector should not be connected.

9.1.4 Code Examples

- **LCD_Example1:** Simple Text to LCD.

LCD_Example1

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ; LCD_Example1
2  ;Portc0-7 connected to LCD_data0-7
3  ;PB0 - E
4  ;PB1 - rw
5  ;PB2 - rs
6  ; TODO: Alter the address location that 'Bye' starts at so that it is present on the second line.
7  ;       *Make sure you are happy with this, after learning about storing strings in
8  ;        eeprom and program memory we will re-visit this.
9  ;       * Go on . . . do it . . . make it display 'Hello World'
10 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
11 ;.DEVICE ATmega16 -- done in m16def.inc
12 .NOLIST
13 .INCLUDE "m16def.inc"
14 .LIST
15
16 .def TMP1=R16        ;
17 .def TMP2=R17        ;
18
19 .cseg
20 .org $000       ;locate code at address $000
21 rjmp START           ; Jump to the START Label
22
23 .org $02A       ;locate code past the interupt vectors
24 START:
25       ldi TMP1, LOW(RAMEND)     ;initialise the stack pointer
26       out SPL, TMP1
27       ldi TMP1, HIGH(RAMEND)
28       out SPH, TMP1
29       call Init_LCD
30       ldi r16, 0x48
31       call Write_char
32       ldi r16, 0x65
33       call Write_char
34       ldi r16, 0x6c
35       call Write_char
36       call Write_char
37       ldi r16, 0x6f
38       call Write_char
39
40       ldi r16, 0x8d
41       call Write_instruc
42       rcall delay
43       ldi r16, 'B'
44       call Write_char
45       ldi r16, 'y'
46       call Write_char
47       ldi r16, 'e'
48       call Write_char
49
50 MAIN_LOOP:
51       NOP
52       NOP
53       RJMP MAIN_LOOP
54
55 .include "LCD.asm"
```

Code Explanation

- Init_LCD:
  - Code can be found in the appendices on page 196.
  - Lines 96-103 set the directions of the data bus and control lines.
  - The 0x38 written as a command, is a *function set* command (see LCD datasheet in appendix), where the data-width is set to 8-bits, 2-lines and a font size of 5x7 dots is selected.
  - The 0x0c written as a command is a command, where the display is turned on, cursor is off and not blinking.
  - The *0x06* is a *entry mode* command, incrementing address on each character and moving the cursor.

  - The *0x01* is a is s *clear display* command.
- Several ASCII characters are then written: 0x48, 0x65, 0x6c, 0c6c, 0x6f. (Hello)
- A command, 0x8d is written, which sets the *DDRAM address* of the LCD to 13 (D), which moves the cursor to the 14th character position.
- More ASCII characters are written. These are done using character notation to represent the character rather than the ASCII value.

Things for you to try

- Try various setups for the LCD to investigate how they work.

Chapter

10 Lookuptables

Videos (on RUConnected) to watch:

- Vid14-Lookuptables

10.1 Lookup tables

(I am lazy so I am just going to copy-paste from the notes)

Applications for lookup tables

Lookup tables can be used whenever there is a need for associating a particular value with an "index". This can be used when there is no logical or mathematical way of determining the value or when doing so would take too long.

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.

Choosing a location for lookup tables

The location of a lookup table depends on the availability of resuorces and the required speed of operation.

EEPROM

EEPROM provides 512 bytes of space that will not be required by code, it is however rather slow.

Program Memory

Storing lookup tables in the flash based program memory provides the largest storage location (provided it is not needed by code) and is faster then EEPROM, but still slower than SRAM.

SRAM

Lookup tables in SRAM are the fastest, but they can use up the valuable resource rather quickly, given that there is only 1kB of SRAM. It is also volatile, which means that the lookup table would have to be stored in a non-volatile memory location and then transferred to SRAM on startup.

10.1.1 So what are we going to be using it for?

We are going to be using lookup tables to make using stepper motors easier. Driving a stepper in single steps is reasonably easy, but if we wanted to *half-step*[1] the algorithm becomes cumbersome.

So if we look at our previous stepping sequence:

. . .
0001
0010
0100
1000
0001
0010
. . .

and alter it to include half-steps we get:

. . .
0001
0011
0010
0110
0100
1100
1000
1001
0001
. . .

We can store this sequence in a lookup table, and all we have to do is either increment or decrement an index, retrieve the corresponding bit-pattern and we have the steps. What makes this even more versatile is if the windings were not connected in the correct order we could still drive the stepper (with no change to algorithm or code) by just recalculating the values to be put into the look-up table.

10.1.2 Code Examples

- **Stepper_Example3:** Stepping using a look-up table.

[1] Half stepping is where instead of just energising the 'next' winding to move the stator, bothe the current winding and the next winding are energised as an intermediate 'half-step' before energising the next winding by itself. This method allows finer control of movement (i.e. doubled resolution in 3D printers) and greater torque that can be applied.

Stepper_Example3

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ;Stepper Example 3
2  ; Hardware connections.
3  ; pd4->drv0
4  ; pd5->drv1
5  ; pd6->drv2
6  ; pd7->drv3
7  ; pd3 -> button0
8  ; pd2 - button1
9
10 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
11
12 .NOLIST
13 .INCLUDE "m16def.inc"
14 .LIST
15
16 .def zero=R0          ; reserve R0 as our ZERO register
17 .def step=R1          ; will now contain the index into steptable of the current step
18 .def slow=R19
19 .def TMP1=R16        ;defines serve the same purpose as in C,
20 .def TMP2=R17        ;before assembly, defined values are substituted
21 .def TMP3=R18
22 .dseg
23 STEP_table: .byte 10
24
25 .cseg                ;Tell the assembler that everything below this is in the code segment
26 .org $000        ;locate code at address $000
27 jmp START        ; Jump to the START Label
28 .org INT0addr
29 jmp int0isr
30 .org OVF0addr    ; locate next instruction at Timer0's overflow interrupt vector
31 jmp T0ovf_ISR       ; Jump to the interrupt service routine
32 .org OC0addr
33 jmp T0OC_ISR
34
35 .org $02A        ;locate code past the interupt vectors
36 START:
37      ldi TMP1, LOW(RAMEND)      ;initialise the stack pointer
38       out SPL, TMP1
39       ldi TMP1, HIGH(RAMEND)
40       out SPH, TMP1
41       call read_steptable
42       CALL INITIALISE_IO ; Call the subroutine INITIALISE
43       call initialise_t0 ; call subroutine to init t0 with output comparevalue
44      SEI ; enable global interrupts
45 MAIN_LOOP:
46       NOP
47       NOP
48       RJMP MAIN_LOOP
49
50 READ_STEPTABLE:
51      ;For now I am going to manually populate the steptable in SRAM
52      ; single steps
53      ldi xl, low(STEP_table)
54      ldi xh, high(STEP_table)
55      ldi tmp1, 0x00
56      st x+, tmp1
57      ldi tmp1, 0x10
58      st x+, tmp1
59      ldi tmp1, 0x30
60      st x+, tmp1
61      ldi tmp1, 0x20
62      st x+, tmp1
63      ldi tmp1, 0x60
64      st x+, tmp1
65      ldi tmp1, 0x40
66      st x+, tmp1
67      ldi tmp1, 0xC0
68      st x+, tmp1
69      ldi tmp1, 0x80
70      st x+, tmp1
71      ldi tmp1, 0x90
72      st x+, tmp1
73      ldi tmp1, 0x00
74      st x+, tmp1
```

```
75      ret
76
77  Initialise_t0:
78      ldi tmp1, 0xf8 ;
79      out ocr0, tmp1 ; setup oc0
80      ldi tmp1, 0b00000010
81      out timsk, tmp1 ; enable t0 ocint - we use the output compare intterupt in this case
82                      ; because we will set the timer to CTC mode
83      ldi tmp1, 0b00001101; ctc mode, no OC pin use, /1024 prescaler
84      out TCCR0, tmp1
85      ret
86
87  INITIALISE_IO:
88      ldi tmp1, 0xF0;
89      out DDRD, tmp1;
90      ldi tmp1, 0x01; 1 because we want to be past the null on the boundary -
91      mov step, tmp1
92      sbi portd, 2 ; pullup on pd2 pin for int 0
93      ldi tmp1, 0b01000000
94      out GICR, tmp1 ; int0 enabled
95      ldi tmp1, 0b00000010
96      out mcucr, tmp1 ;falling edge triggered.
97      RET                         ;Return from subroutine
98
99  T0ovf_ISR:
100
101     RETI
102 T0OC_ISR:
103 inc slow
104 cpi slow,0x10
105 brne notnow
106 mov slow, zero
107 call Nextstep
108 notnow:
109     RETI
110
111 int0isr:
112 ;call nextstep
113    RETI
114
115 NEXTSTEP:
116 inc step ; get to the next step
117 ; get correct value from lookuptable
118 getstep:
119 ldi xl, low(STEP_table)
120 ldi xh, high(STEP_table)
121 add xl, step
122 adc xh, zero
123 ld tmp2, X
124 cpi tmp2, 0x00
125 brne outstep
126  ; if we got a null then we need to wrap around
127   ldi tmp2, 0x01 ;using tmp2 since we need the value in tmp1 and tmp2's value is rubbish at this
        moment'
128   mov step, tmp2
129   rjmp getstep
130 outstep:
131   IN tmp1, PORTD
132   andi tmp1, 0x0F ;tmp1 now contains the masked off values of portD
133   or tmp1, tmp2
134   out portd, tmp1
135   RET
```

Code Explanation

- READ_STEPTABLE: We manually populate a lookup table in SRAM.
  - Lines 58-83: the X register is pointed to STEP_table, which was setup in lines 30-31.
  - The steps are bounded by NULLs to delimit and are set up to provide halfsteps.
  - 0x00, 0x01, 0x03, 0x02, 0x06, 0x04, 0x0C, 0x09, 0x00
- Timer0: Timer 0 is used to set up a delay between steps as in previous examples.
- NEXTSTEP:

- This routine adds the offset into the table[a] to the address of the table and the step is retrieved from RAM.
- If a NULL is retrieved a *wrap around* is performed.
- This step is then output.

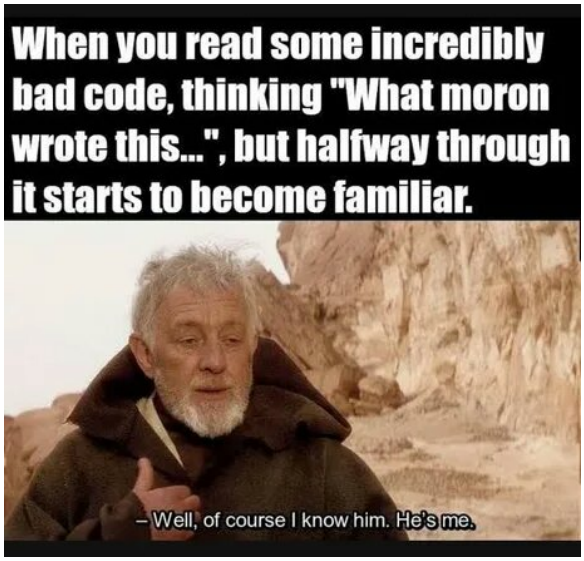
[a] step

Things for you to try

- Can you implement a system that will wrap seamlessly (without changing hardcoded limits) regardless of the size of the lookup table?

CHAPTER

11 Using program memory and EEPROM



Videos (on RUConnected) to watch:

- Video15-Memories

11.1 Using Program Memory and EEPROM

11.1.1 Overview

At some point we are going to want to use our EEPROM for what it was intended (and program memory because there is just so much of it - face it you have noticed how small the code is that you are generating - if not look at the window in Codeblocks when you assemble your code and take note of the size).

11.1.2 Reserving Space for Data in SRAM

Firstly we need to see how we can reserve space in SRAM for variables. Please note, we cannot actually assign values to the variables - SRAM is volatile - any initialisation of variables should be done in your code.

```
1 .DSEG
2 Variable1: .BYTE 8
3 Variable2: .BYTE 1
```

Code Explanation

The .DSEG tells that this is in the Data (SRAM) segment, and space needs to be allocated to a label Variable1 that is 8 bytes long as well as a label Variable2 that is 2 bytes long

We will worry about accessing these locations later.

11.1.3 Storing data in Program Memory

To define variables in Program Memory we do the following:

```
1 .CSEG
2 .ORG 0xf00
3 Variable1: .BYTE 1
4 Variable2: .DB 15
5 Variable3: .DB "This is a string",0x00
```

Code Explanation

The .CSEG tells the assembler that this is in the code segment. The .ORG is locating these variables at some point.[1] Variable1 has one byte located for it, buy no value attached. Variable2 is one byte long and contains the value 15. Variable3 Contains a null terminated[2] string of text and is 17 bytes long. This data is automatically added to the .hex file generated by the assembler.

11.1.4 Storing data in EEPROM

To define variables in EEPROM we do the following:

```
1 .ESEG
2 .ORG 0x00
3 Variable1: .BYTE 1
4 Variable2: .DB 15
5 Variable3: .DB "This is a string",0x00
```

Code Explanation

The .ESEG tells the assembler that this is in the EEPROM segment. The .ORG is locating these variables at the beginning od EEPROM Variable1 has one byte located for it, buy no value attached. Variable2 is one byte long and contains the value 15. Variable3 Contains a null terminated string of text and is 17 bytes long. This data is automatically added to the .eep file generated by the assembler. So when we are using EEPROM, we need to use the tool to program the .eep file as well or EEPROM will be empty - actually each location will have the value 0xff

11.1.5 A quick Primer on the X, Y and Z registers

Strictly speaking the X, Y and Z registers have nothing to do with EEPROM or program memory, but we need them to access data in SRAM and program memory, because the instructions used to store and retrieve data use them. (See section 5.3 in your notes or find it in the datasheet.)

There registers are just other names (or think of them as aliases) for the registers R26 to R31 as shown below:

[1] Normally you would leave out the .org and just place the rest of the instructions at the end of your code and the assembler will locate them itself. The assembler will gladly overwrite things if told to do so. - You have been warned - see footnotes are relevant.

[2] ALWAYS NULL terminate your strings.

R26	XL (The low byte of the X register)
R27	XH (The high byte of the X register)
R28	YL (The low byte of the Y register)
R29	YH (The high byte of the Y register)
R30	ZL (The low byte of the Z register)
R31	ZH (The high byte of the Z register)

The Indirect Address Registers are used as pointers for the *Data Transfer Instructions* that need them. See your Instruction set summary.

We are going to need these alot later, so it is a good idea not to use R26-R31 for other purposes in your code.

11.1.6 Reading Data from EEPROM and Program Memory to SRAM

We are now going to work through a simplified example of what is done in Chapter 18 of your notes.

```
1  .INCLUDEPATH "/usr/share/avra"
2  .NOLIST
3  .INCLUDE "m16def.inc"
4  .LIST
5  .def TMP1=R16
6  .def MESSAGE_offset=r19
7  .eseg
8  .org 0x000
9  EEPMESSAGE:      .db "Hello",0x00
10 .dseg
11 MESSAGE: .byte 10 ; reserve 10bytes for the
       message
12 .cseg
13 .org $000        ;locate code at address $000
14 rjmp START            ; Jump to the START Label
15 .org $02A        ;locate code past the interupt
       vectors
16 START:
17       ldi TMP1, LOW(RAMEND)     ;initialise the
             stack pointer
18       out SPL, TMP1
19       ldi TMP1, HIGH(RAMEND)
20       out SPH, TMP1
21       call READ_MESSAGES_FROM_EEPROM
22 MAIN_LOOP:
23     NOP
24       NOP
25       RJMP MAIN_LOOP
26
27 READ_MESSAGES_FROM_EEPROM:
28       LDI XL, low(EEPMESSAGE)
29       LDI XH, hight(EEPMESSAGE)
30       LDI YL, low(MESSAGE)
31       LDI YH,     high(MESSAGE)
32       call Read_Bytes
33       RET
34
35 READ_BYTES:
36       SBIC EECR, EEWE     ; wait to make sure
             there is no active write
37       RJMP READ_BYTES
38       out EEARH, XH
39       OUT EEARL, XL
40       SBI EECR, EERE
41       in tmp2, EEDR
42       st y+, tmp2
43       adiw xl, 1
44       cpi tmp2, 0x00 ; test for null
45       BRNE READ_BYTES
46       ret
```

Code Explanation

- EEPMESSAGE is located at the beginning of EEPROM and contains the null-terminated string "Hello".
- 10 bytes are reserved for the variable Message in SRAM.
- In READ_MESSAGES_FROM_EEPROM the X register is set up to contain the address of the first byte of EEPMESSAGE in EEPROM and the Y register is set up to contain the address of the first byte of MESSAGE in SRAM.
- In READ_BYTES the first thing done is to make sure that an EEPROM write is not in progress. (Because we are **never going to write** to EEPROM in our code this can be left out.) The code stays in a loop until any pending write has completed.
- The address that we wish to read from EEPROM is put into the Eeprom Address Register. (It is a 16-bit register so remember the rule.)
- A EEPROM read is triggered by setting the EERE bit in EECR.
- (Read through the EEPROM section in your notes as well as the datasheet to ensure you understand what is going on in the background - Consider it an exam ***HINT***.)
- The byte that the magical gnomes fetched from EEPROM is now in EEDR (the EEPROM data register), this is read into tmp2.
- The value is then stored to the SRAM location pointed to by the Y register (Note - a post-increment is used, i.e. the Y register is automatically incremented after the storing is done.)
- We than add 1 to the X register to get it to point to the next address that is to be read in.
- If tmp2 (the last value read in) was a null, then we exit, otherwise we read in the next byte.
- PLEASE NOTE: EEPROM is accessed as a peripheral, we do not access the data stored in it like we to the other two memory segments. We have to manually manipulate the peripheral to get the data out.

```
1   .INCLUDEPATH "/usr/share/avra" ;set the inlude
        path to the correct place
2  .NOLIST
3  .INCLUDE "m16def.inc"
4  .LIST
5  .def TMP1=R16        ;
6  .def TMP2=R17        ;
7  .def TMP3=R18
8  .dseg
9  SRAM_VAR_1:     .BYTE 4 ;reserve 4 bytes of RAM
       for this variable
10 .cseg
11 .org 0x000           ;locate code at address $000
12 rjmp START           ; Jump to the START Label
13
14 .org $02A       ;locate code past the interupt
       vectors
15
16 START:
17       ldi TMP1, LOW(RAMEND)     ;initialise the
             stack pointer
18       out SPL, TMP1
19       ldi TMP1, HIGH(RAMEND)
20       out SPH, TMP1
21       RCALL READ_IN_CONSTANTS
22 MAIN_LOOP:
23       NOP
24       NOP
25       NOP
26       RJMP MAIN_LOOP
27
28 READ_IN_CONSTANTS:
29       ;Setup pointer to .cseg locations
30       LDI ZL, LOW(2*constants)
31       LDI ZH, high(2*constants)
32       LDI YL, low(SRAM_VAR_1)
33       LDI YH, high(SRAM_VAR_1)
34
35       ;transfer first byte
36       LPM TMP1, Z+
37       ST Y+, TMP1
38       ;transfer 2nd byte
39       LPM TMP1, Z+
40       ST Y+, TMP1
41       ;transfer 3rd byte
42       LPM TMP1, Z+
43       ST Y+, TMP1
44       RET
45
46 constants: .DB 0xDE, 0x73, 0b01010101
```

Code Explanation

- SRAM_VAR_1 has 4 bytes reserved for it in SRAM.
- constants has its contents defined right at the end of the code.
- The Z register is set to the address of the first byte of constants. (Note: we use 2*constants because constants is the address in **words** of its location in program memory and we need the byte address for retrieving data from program memory.)
- The Y register is set tthe address of the first byte of SRAM_VAR_1.
- LPM is used to (L)oad (P)rogram (M)emory from the location pointed to by the Z register into TMP1. A post increment is used so that the Z register is pointing to the next location for the next read.
- The Z register is the only register that can be used to access program memory - see intruction set summary.
- TMP1 is then stored to the SRAM location pointed to by the Y register - with post increment.

11.1.7 Code Examples

- **LCD_Example2:** Writing strings from program memory and EEPROM to the LCD.

LCD_Example2

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ; LCD_Example2
2  ;Portc0-7 connected to LCD_data0-7
3  ;PB0 - E
4  ;PB1 - rw
5  ;PB2 - rs
6  .INCLUDEPATH "/usr/share/avra"
7  .NOLIST
8  .INCLUDE "m16def.inc"
9  .LIST
10
11 .def TMP1=R16        ;
12 .def TMP2=R17        ;
13
14 .eseg
15 EEMSG: .DB "EEP-hello world",0x00
16
17 .dseg
18 message1: .BYTE 20
19 message2: .BYTE 20
20 .cseg
21 .org $000
22 rjmp START
23
24 .org $02A
25 START:
26       ldi TMP1, LOW(RAMEND)
27       out SPL, TMP1
28       ldi TMP1, HIGH(RAMEND)
29       out SPH, TMP1
30       call Init_LCD
31      ldi XL, low(Message1)
32      ldi XH, high(Message1)
33      ldi YL, low(EEMSG)
34      ldi YH, high(EEMSG)
35      call READEE
36      ldi XL, low(Message2)
37      ldi XH, high(Message2)
38      ldi ZL, low(2*PGMMSG)
39      ldi ZH, high(2*PGMMSG)
40      call READPGM
41      ldi XL, low(Message1)
42      ldi XH, high(Message1)
43      call Message2LCD
44
45 MAIN_LOOP:
46       NOP
47       NOP
48       RJMP MAIN_LOOP
49
50 Message2LCD:
51      ld tmp1, x+
52      cpi tmp1, 0x00
53      breq m2ldone
54      call Write_char
55      rjmp Message2LCD
56 m2ldone:
57      RET
58
59 READEE:
60      OUT EEARH, YH
61      OUT EEARL, YL
62      SBI EECR, EERE
63      IN TMP1, EEDR
64      st X+, tmp1
65      adiw YL, 1
66      cpi TMP1, 0x00
67      BRNE READEE
68      RET
69
70 READPGM:
71      LPM tmp1, Z+
72      st X+, tmp1
73      cpi TMP1, 0x00
74      BRNE READPGM
```

```
75      RET
76
77  .include "LCD.asm"
78 PGMMSG:  .db "PM-hello world",0x00
```

Things for you to try

- Change from sending message1 to LCD to message2
- Make the LCD wrap text to the next line if it is longer than 16 characters
- Use timer 1 to generate a 2second interrupt, use this to alternate messages displayed on the LCD

Stepper_Example3 - revisited

Things for you to try

Can you take the original code for Stepper_Example3 and modify it to do the following:

- Store the values for the lookup table in EEPROM and read them in to SRAM during startup.
- Reverse the values in the lookup table and verify that the direction of rotation is reversed.
- Rewrite next_step as stepleft and stepright that step in oppisite directions and test them.
- Wire button 0 to PA0, when the button is held down, step left. When the button is not held down step right.
- Alter you code so that the lookuptable is stored in program memory and used directly from program memory without using SRAM.

CHAPTER

12 Using the U(S)ART

Stuff to read:

-

Videos (on RUConnected) to watch:

- Video18 - UART Overview

12.1 Universal Asynchronous Receiver / Transmitter

The U(S)ART[1] is a peripheral allowing communications in a variety of standards, most notably RS232[2] (what serial mice and printers used back in the day of dial-up modems - which also used RS232 to communicate.

See page 145 of the notes and pages 144 to 171 of the datasheet.

For an overview of RS232 see https://www.best-microcontroller-projects.com/how-rs232-works.html, note we are only using logic level signals, not ±12 V

[1] The 'S' stands for synchronous - i.e. when a clock is distributed with a signal - we are only going to be looking at this peripheral in Asynchronous mode.

[2] The different RS standards differ mainly in their electrical characteristics.

12.1.1 Settings

The main settings we need to worry about are:

- Data Length:
  - We are exclusively going to be using 8 bits of data.
- Stop Bits:
  - One or 2 stop bits is fine. More stop bits means we have more time to process a byte before the next is received. One stop bit (104μs) means 833 instruction cyles - quite a bit of time at 9600 baud.
- Parity:
  - We are not doing any parity checking.
- Flow Control:
  - We are doing no flow control. (Make sure your Putty setup[6] reflects this or you will not get anything to work.)
- Baud Rate:
  - Fortunately for us the programmer/USB-serial converter will only do 9600 baud (I was lazy when I wrote the code for it and now I can't be bother to find the source code to alter it - so you win.
  - the baud rate is the number of bits that are transmitted per second, so the duration of 1 bit is 0.104 ms.

12.1.2 Sending and Receiving Data

This is outlined starting on page 145 of your notes.

12.1.3 Working with buffers

Sometimes we will want to send an entire buffer of data. This is generally done by filling the buffer (a variable in SRAM) with the data, setting up a pointer to the next character to be sent, sending the first character of the buffer and then when a TXC or UDRE interrupt occurs we just send the next character in the buffer (and update the pointer to the next byte to be sent). See UART_Example2.

Receiving works similarly, received characters are stored in a buffer sequentially as they arrive and then when a terminating control character is received the buffer is processed. See UART_Example3.

12.1.4 Code Examples

- **UART_Example1:** Sending and Receiving a character.
- **UART_Example2:** Sending a string.
- **UART_Example3:** Receiving a string.

[6] Under the Serial settings right at the bottom of the window on the startup of putty.

UART_Example1

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ; USART
2  ; Connect RX and TX from usb-serial converter to TX and RX on board
3  ; The current code just sets up the UART to RX and TX at 9600,
4  ; when a character is received, it is "incremented" and then sent back - mainly to avoid
5  ; any problems with "local echo" settings that may or may not be set in putty.
6  ; Remember after programming the device, change the mode jumper - unplug usb - replug usb and then
       open putty
7  .INCLUDEPATH "/usr/share/avra"
8  .NOLIST
9  .INCLUDE "m16def.inc"
10 .LIST
11
12 .def TMP1=R16        ;
13 .def TMP2=R17        ;
14 .dseg
15
16 .eseg
17
18 .cseg
19 .org $000       ;locate code at address $000
20 rjmp START           ; Jump to the START Label
21 .org URXCaddr
22 rjmp URXC_ISR
23 .org UDREaddr
24 rjmp UDRE_ISR
25 .org UTXCaddr
26 rjmp UTXC_ISR
27
28 .org $02A       ;locate code past the interupt vectors
29
30 START:
31       ldi TMP1, LOW(RAMEND)     ;initialise the stack pointer
32       out SPL, TMP1
33       ldi TMP1, HIGH(RAMEND)
34       out SPH, TMP1
35       call Init_UART
36       ldi tmp1, 0x55 ; Send a 'U' on startup - to see this connect with putty and press the reset
             button.
37       out UDR, tmp1
38       SEI
39 MAIN_LOOP:
40       NOP
41       NOP
42       RJMP MAIN_LOOP
43
44 Init_UART:
45 ;set baud rate (9600,8,n,2)
46       ldi Tmp1, 51
47       ldi Tmp2, 0x00
48       out UBRRH, Tmp2
49       out  UBRRL, Tmp1
50 ;set rx and tx enable
51       sbi UCSRB, RXEN
52       sbi UCSRB, TXEN
53 ; enable uart interrupts
54       sbi UCSRB, RXCIE
55       RET
56
57 ; Interrupt code for when UDR empty
58 UDRE_ISR:
59       RETI
60 ; Code for TX complete
61 UTXC_ISR:
62       RETI
63 ; Code for RX complete
64 URXC_ISR:
65      in tmp1, UDR
66      inc tmp1
67      out UDR, tmp1
68      RETI
```

UART_Example2

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ; USART
2  ; Connect RX and TX from usb-serial converter to TX and RX on board
3
4  .INCLUDEPATH "/usr/share/avra"
5  .NOLIST
6  .INCLUDE "m16def.inc"
7  .LIST
8
9  .def TMP1=R16        ;
10 .def TMP2=R17        ;
11 .dseg
12 BUFFER: .byte 10
13 .eseg
14 MSG: .db "Hello",0x00
15 .cseg
16 .org $000       ;locate code at address $000
17 rjmp START           ; Jump to the START Label
18 .org URXCaddr
19 rjmp URXC_ISR
20 .org UDREaddr
21 rjmp UDRE_ISR
22 .org UTXCaddr
23 rjmp UTXC_ISR
24
25 .org $02A       ;locate code past the interupt vectors
26
27 START:
28       ldi TMP1, LOW(RAMEND)     ;initialise the stack pointer
29       out SPL, TMP1
30       ldi TMP1, HIGH(RAMEND)
31       out SPH, TMP1
32       ; Read a message from eeprom into the buffer
33       ldi XL, low(Buffer)
34      ldi XH, high(Buffer)
35      ldi YL, low(MSG)
36      ldi YH, high(MSG)
37      call READEE
38      ; init uart
39       call Init_UART
40       SEI
41 MAIN_LOOP:
42       NOP
43       NOP
44       RJMP MAIN_LOOP
45
46 READEE:
47      OUT EEARH, YH
48      OUT EEARL, YL
49      SBI EECR, EERE
50      IN TMP1, EEDR
51      st X+, tmp1
52      adiw YL, 1
53      cpi TMP1, 0x00
54      BRNE READEE
55      RET
56
57 Init_UART:
58 ;set baud rate (9600,8,n,2)
59       ldi Tmp1, 51
60       ldi Tmp2, 0x00
61       out UBRRH, Tmp2
62       out  UBRRL, Tmp1
63 ;set rx and tx enable
64       sbi UCSRB, RXEN
65       sbi UCSRB, TXEN
66 ; enable uart interrupts
67       sbi UCSRB, RXCIE
68       sbi UCSRB, TXCIE
69       RET
70
71 ; Interrupt code for when UDR empty
72 UDRE_ISR:
73       RETI
74 ; Code for TX complete
```

```
75 UTXC_ISR:
76     ;get the next byte and post increment X
77     ld tmp1, x+
78     cpi tmp1, 0x00
79     breq gotnull ;if the nex char to send is a null don't send it
80     ; Send it out into the world
81     out udr, tmp1
82 donetx:
83      RETI
84
85 gotnull:
86
87     RETI
88 ; Code for RX complete
89 URXC_ISR:
90     in tmp1, udr; This looks pointless doesn't it?
91     ; Comment it out and see if you can explain the behaviour - read through the reception section
           in the datasheet
92     ; when we get a received char say hello - it is only polite
93     call send_buffer
94     RETI
95
96 Send_buffer:
97     ; point X to beginning of buffer
98     ldi XL, low(Buffer)
99     ldi XH, high(Buffer)
100    ;get the first byte and post increment X
101    ld tmp1, x+
102    ; Send it out into the world
103    out udr, tmp1
104        RET
```

Code Explanation

- The code reads in a message from EEPROM to RAM.
- UART is initialised.
- When a character is received the Ram buffer containing the message is transmitted.

Things for you to try

- As mentioned in the code find out what happens when UDR is not read on reception.
- Alter the code so that the message is not sent unless a specific character is received.
- Change your code so that there are two different messages in EEPROM and a different one is sent depending on the received

UART_Example3

Firstly here is the code with line numbers to facilitate easy referencing:

```
1  ; USART
2  ; Connect RX and TX from usb-serial converter to TX and RX on board
3
4  .INCLUDEPATH "/usr/share/avra"
5  .NOLIST
6  .INCLUDE "m16def.inc"
7  .LIST
8  .def zero=r0
9  .def TMP1=R16        ;
10 .def TMP2=R17        ;
11 .def rxcnt=r18
12 .dseg
13 BUFFER: .byte 20
14 .cseg
15 .org $000       ;locate code at address $000
16 rjmp START           ; Jump to the START Label
17 .org URXCaddr
18 rjmp URXC_ISR
19
20 .org $02A       ;locate code past the interupt vectors
21
22 START:
23       ldi TMP1, LOW(RAMEND)     ;initialise the stack pointer
24       out SPL, TMP1
25       ldi TMP1, HIGH(RAMEND)
26       out SPH, TMP1
27      ; init uart
28      ldi tmp1, 0x00
29      mov zero, tmp1
30      mov rxcnt, tmp1
31       call Init_UART
32      call Init_LCD
33
34       SEI
35 MAIN_LOOP:
36       NOP
37       NOP
38       RJMP MAIN_LOOP
39
40
41 Init_UART:
42 ;set baud rate (9600,8,n,2)
43       ldi Tmp1, 51
44       ldi Tmp2, 0x00
45       out UBRRH, Tmp2
46       out  UBRRL, Tmp1
47 ;set rx and tx enable
48       sbi UCSRB, RXEN
49       sbi portd, 1 ; enable pullup on unused transmitter pin - see what happens when you remove this
              line
50 ; enable uart interrupts
51       sbi UCSRB, RXCIE
52       RET
53
54 ; Code for RX complete
55 URXC_ISR:
56      in tmp1, udr
57      cpi tmp1, '*'
58      breq Processbuffer
59      cpi rxcnt, 0x0F  ; cpi is rd-K
60      BREQ overrun
61      ; if we get here we have not got a buffer overrun
62      inc rxcnt
63      out porta, rxcnt
64      ldi xl, low(BUFFER)
65      ldi xh, high(BUFFER)
66      ;add offset
67      add xl, rxcnt
68      adc xh, zero
69      st x, tmp1
70      ld tmp1, x
71      call Write_char
72          RETI
73
```

```
74 Processbuffer:
75     sbi ddra, 7
76     sbi porta, 7
77     mov rxcnt, zero
78     RETI
79
80 overrun:
81     sbi ddra, 6
82     sbi porta, 6
83     mov rxcnt, zero
84     reti
85 .include "LCD.asm"
```

Code Explanation

Things for you to try

- This example does not use the transmitter.
- Received characters are stored in a RAM buffer (BUFFER) until a '*' is received.
- Change the code so that the characters received are printed to the LCD.
- Alter the code further so that if the backspace ASCII character is the first character in the buffer then the LCD is cleared.
- Change your code further so that is the TAB character is the first in the buffer then the remaining text is written to the other line of the LCD.

CHAPTER

13 Things we have not covered yet.

Stuff to read:

-

Videos (on RUConnected) to watch:

- Video19-Sleepmodes
- Video20-Watchdog timer

13.1 Sleep Modes

Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power. The Mega16 has a few different modes. I do not expect you to know the details of each mode. The modes differ in what is shut down and what will wake the MCU from its sleep state.

See Chapter 24 of your notes.

13.2 The Watchdog Timer

A watchdog timer (WDT) is a hardware timer that automatically generates a system reset if the main program neglects to periodically service it. It is often used to automatically reset an embedded device that hangs because of a software or hardware fault.

The main program typically has a loop that it constantly goes through performing various functions. The watchdog timer is loaded with an initial value greater than the worst case time delay through the main program loop. Each time it goes through the main loop the code resets the watchdog timer (sometimes called "kicking" or "feeding" the dog). If a fault occurs and the main program does not get back to reset the timer before it counts down, an interrupt is generated to reset the processor. Used in this way, the watchdog timer can detect a fault on an unattended embedded device and attempt corrective action with a reset. Typically after reset, a register can also be read to determine if the watchdog timer generated the reset or if it was a normal reset.

See chapter 30 of your notes for details of the Watchdog timer on the mega16.

CHAPTER

14 Further Challenges

In preparation for the final assesment you might want to consider trying the following:

- Sending strings directly from program memory via the UART.
- Stepping the stepper motor at a well-defined rate.
- Stepping the stepper a precise number of steps.
- Parsing received buffers from the UART.

CHAPTER 15

Questions you should be able to answer

Here is a list of questions that you should be able to answer by going through the datasheet/notes/tasks/examples (consider this exam prep):

- What is a RISC processor?
- How does the ATmega16 interact with the external world?
- What are the different methods of applying a clock source to the ATmega16? List the inherent advantages/disadvantages of each type.
- Describe the three registers associated with each port.
- With a specific port, can some port pins be declared as output pins while others as input pins, if so how?
- What is the best clock speed to operate the ATmega16 at for a specific application?
- If an ATmega16 microcontroller is operating at 12 MHz, what is the maximum transmission rate for the USART?
- If 12 bits are used to quantize a sampled signal, what is the number of available quantized levels? What will be the resolution of such a system if the input range of the ADC is 10 V?
- If an analog signal is converted by an ADC to a binary representation and then back to an analog voltage using a DAC, will the original analog input voltage be the same as the resulting analog output voltage? Explain.
- Describe the flow of events when an interrupt occurs.
- What is interrupt priority? How is it determined?
- Given an 8-bit free-running counter and a system clock rate of 24 MHz, find the time required for the counter to count from 0 to its maximum value.
- Describe how you can compute the period of an incoming digital signal with varying duty cycles.
-

PART II

The AVR Assembly Language

CHAPTER

16 Why learn assembly language?

Why should you learn another language, if you already learned other programming languages? The best argument: while you live in France you are able to get through by speaking English, but you will never feel at home and life remains complicated. You can get through with this, but it is rather inappropriate. If things need a hurry, you should use the country's language.

Assembler commands translate one by one to executed machine instructions. The processor needs only to execute what you want it to do and what is necessary to perform the task. No extra loops and unnecessary features blow up the generated code. If your program storage is short and limited and you have to optimize your program to fit into memory, assembler is the best choice. Shorter programs are easier to debug, every step makes sense.

Because only necessary code steps are executed assembly programs are as fast as possible. The duration of every step is known. Time critical applications, like time measurements without a hardware timer.

It is not true that assmbly language is more complicated or not as easy to understand than other languages. Learning assembly language for whatever hardware type brings you to understand the basic concepts of any assembly language. Adding other dialects later is easy. It also means that you must know the architecture and peripherals of the processor that you are using. At first assembly code does not look very attractive.

The Assembler works on source files containing instruction mnemonics, labels and directives. The instruction mnemonics and the directives often take operands. Code lines should be limited to 120 characters. Every input line can be preceded by a label, which is an alphanumeric string terminated by a colon. Labels are used as targets for jump and branch instructions and as variable names in Program memory and RAM. An input line may take one of the four following forms:

1. [label:] directive [operands] [Comment]
2. [label:] instruction [operands] [Comment]
3. Comment
4. Empty line

A comment has the following form:
; [Text]
Items placed in braces are optional. The text between the comment-delimiter (;) and the end of line (EOL) is ignored by the Assembler. Labels, instructions and directives are described in more detail later. Examples:

```
label:    .EQU var1=100     ; Set var1 to 100 (Directive)
.EQU var2=200     ; Set var2 to 200
test:     rjmp  test        ; Infinite loop (Instruction)
; Pure comment line
; Another comment line
```

CHAPTER 17

Instruction Set

The Assembler is not case sensitive.

Status Register

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

These flags are set/cleared by certain instructions under certain conditions. They can also be directly manipulated.

The operands have the following forms:
Rd: R0-R31 or R16-R31 (depending on instruction)
Rr: R0-R31
b: Constant (0-7), can be a constant expression
s: Constant (0-7), can be a constant expression
P: Constant (0-31/63), can be a constant expression
K: Constant (0-255), can be a constant expression
k: Constant, value range depending on instruction. Can be a constant expression.
q: Constant (0-63), can be a constant expression

We are going to look at the different types of instructions available to us.

17.1 Arithmetic and Logic Instructions

As advertised these instructions perform arithmetic or logic operations.

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rd, K	Add Immediate to Word	Rd+1:Rd ← Rd+1:Rd + K	Z,C,N,V	2
SUB	Rd, Rr	Subtract without Carry	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract Immediate with Carry	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rd, K	Subtract Immediate from Word	Rd+1:Rd ← Rd+1:Rd - K	Z,C,N,V	2
AND	Rd, Rr	Logical AND	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND with Immediate	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR with Immediate	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← $FF - Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← $00 - Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • ($FFh - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd - 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← $FF	None	1
MUL	Rd,Rr	Multiply Unsigned	R1, R0 ← Rd × Rr	C	2 (1)

Note: 1. Not available in base-line microcontrollers

17.2 Branch Instructions

These instructions allow changes in flow of the code.

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP	k	Jump	PC ← k	None	3
RCALL	k	Relative Call Subroutine	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL	k	Call Subroutine	PC ← k	None	4
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd - Rr	Z,C,N,V,H	1
CPC	Rd,Rr	Compare with Carry	Rd - Rr - C	Z,C,N,V,H	1
CPI	Rd,K	Compare with Immediate	Rd - K	Z,C,N,V,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if(I/O(P,b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register Set	if(I/O(P,b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC ← PC+k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC ← PC+k + 1	None	1 / 2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC+ k + 1	None	1 / 2
BRLT	k	Branch if Less Than, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1 / 2

17.3 Data Transfer Instructions

These Instructions govern data transfer operations available.

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Copy Register	Rd ← Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	3
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Increment	Rd ← (X), X ← X + 1	None	2
LD	Rd, -X	Load Indirect and Pre-Decrement	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Increment	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, -Y	Load Indirect and Pre-Decrement	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Increment	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Decrement	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	3
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Increment	(X) ← Rr, X ← X + 1	None	2
ST	-X, Rr	Store Indirect and Pre-Decrement	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Increment	(Y) ← Rr, Y ← Y + 1	None	2
ST	-Y, Rr	Store Indirect and Pre-Decrement	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Increment	(Z)← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Decrement	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q)← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

17.4 Bit Test and Set Instructions

These instructions allow testing of certain bits as well as manipulation thereof.

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
BIT AND BIT-TEST INSTRUCTIONS					
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n),Rd(0) ← 0,C ← Rd(7)	Z,C,N,V,H	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1),Rd(7) ← 0,C ← Rd(0)	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0) ← C,Rd(n+1) ← Rd(n),C ← Rd(7)	Z,C,N,V,H	1
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C,Rd(n) ← Rd(n+1),C ← Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0) ↔ Rd(7..4)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
SBI	P, b	Set Bit in I/O Register	I/O(P, b) ← 1	None	2
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) ← 0	None	2
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Two's Complement Overflow	V ← 1	V	1
CLV		Clear Two's Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ←1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1

17.5 MCU Control Instructions

NOP		No Operation		None	1
SLEEP		Sleep		None	1
WDR		Watchdog Reset		None	1

CHAPTER 18

Assembler Directives

The Assembler supports a number of directives. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on. An overview of the directives is given in the following table.

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

Note: All directives must be preceded by a period.

18.1 Directives That Make Things Easier to Read

18.1.1 DEF - Set a symbolic name on a register

The DEF directive allows the registers to be referred to through symbols. A defined sym- bol can be used in the rest of the program to refer to the register it is assigned to. A reg- ister can have several symbolic names attached to it. A symbol can be redefined later in the program.
Syntax:
.DEF Symbol=Register
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
```

18.1.2 ENDMACRO - End macro

The ENDMACRO directive defines the end of a Macro definition. The directive does not take any parameters. See the MACRO directive for more information on defining Macros.
Syntax:
.ENDMACRO
Example:

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

18.1.3 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.
Syntax:
.EQU label = expression
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
```

18.1.4 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.
Syntax:
.INCLUDE “filename”
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
```

18.1.5 (NO)LIST - Turn thelistfile generation on/off

The LIST/NOLIST directive tells the Assembler to turn listfile generation on/off. The Assembler generates a listfile which is a combination of assembly source code, addresses and opcodes. Listfile generation is turned on by default. The directive can also be used together with the NOLIST directive in order to only generate listfile of selected parts of an assembly source file.
Syntax:
.LIST
.NOLIST
Example:

```
.NOLIST ; Disable listfile generation
.INCLUDE “macro.”inc ; The included files will not
.INCLUDE “const.”def ; be shown in the listfile
.LIST ; Reenable listfile generation
```

18.1.6 MACRO - Begin macro

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 Assem- bler. 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.
Syntax:
.MACRO macroname
Example:

```
.MACRO SUBI16 ; Start macro definition
subi @1,low(@0) ; Subtract low byte
sbci @2,high(@0) ; Subtract high byte
.ENDMACRO ; End macro definition
.CSEG ; Start code segment

SUBI16 0x1234,r16,r17; Sub.0x1234 from r17:r16
```

18.1.7 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 expres- sions. A label assigned to a value by the SET directive can be changed later in the pro- gram.
Syntax:
.SET label = expression
Example:

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

18.2 Directives that control structure/location

18.2.1 BYTE - Reserve bytes to a variable

The BYTE directive reserves memory resources in the SRAM. 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 only be used within a Data Segment (see directives CSEG, DSEG and ESEG). Note that a parameter must be given. The allocated bytes are not initialized. Syntax:
LABEL: .BYTE expression
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
```

18.2.2 CSEG - Code Segment

The CSEG directive defines the start of a Code Segment. An Assembler file can consist of several Code Segments, which are concatenated into one Code Segment when assembled. The BYTE directive can not be used within a Code Segment. The default segment type is Code. The Code Segments have their own location counter which is a word counter. The ORG directive can be used to place code and constants at specific locations in the Program memory. The directive does not take any parameters.

Syntax:
.CSEG
Example:

```
.DSEG
; Start data segment
vartab: .BYTE 4 ; Reserve 4 bytes in SRAM
.CSEG ; Start code segment
const: .DW 2 ; Write 0x0002 in prog.mem.
mov r1,r0 ; Do something
```

18.2.3 DB-Define constant byte(s) in program memory or E^{2}PROM memory

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 expres- sion must evaluate to a number between -128 and 255. If the expression evaluates to a negative number, the 8 bits two's complement of the number will be placed in the pro- gram memory or EEPROM memory location. If the DB directive is used in a Code Segment and the expressionlist contains more than one expression, the expressions are packed so that two bytes are placed in each pro- gram 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 assembly code contains a DB directive.
Syntax:
LABEL: .DB expressionlist
Example:

```
.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
eeconst:.DB 0xff
```

18.2.4 DEVICE - Define which device to assemble for

The DEVICE directive allows the user to tell the Assembler which device the code is to be executed on. If this directive is used, a warning is issued if an instruction not supported by the specified device occurs in the code. If the size of the Code Segment or EEPROM Segment is larger than supported by the specified device, a warning is issued. If the DEVICE directive is not used, it is assumed that all instructions are supported and that there are no restrictions on memory sizes.
Syntax:
.DEVICE AT90S1200 | AT90S2313 | AT90S4414 | AT90S8515
Example:

```
.DEVICE AT90S1200
; Use the AT90S1200
.CSEG
push r30 ; This statement will generate
        ; a warning since the
        ; specified device does not
        ;have this instruction
```

18.2.5 DSEG - Data Segment

The DSEG directive defines the start of a Data Segment. An Assembler file can consist of several Data Segments, which are concatenated into one Data Segment when assembled. A Data Segment will normally only consist of BYTE directives (and labels). The Data Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place the variables at specific locations in the SRAM. The directive does not take any parameters.
Syntax:
.DSEG

Example:

```
.DSEG ; Start data segment
var1:.BYTE 1 ; reserve 1 byte to var1
table:.BYTE tab_size ; reserve tab_size bytes.
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
```

18.2.6 DW-Define constant word(s) in program memory or E^{2}PROM memory

The DW directive reserves memory resources in the program memory or 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 location.
Syntax:
LABEL: .DW expressionlist
Example:

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

18.2.7 ESEG - EEPROM Segment

The ESEG directive defines the start of an EEPROM Segment. An Assembler file can consist of several EEPROM Segments, which are concatenated into one EEPROM Segment when assembled. The BYTE directive can not be used within an EEPROM Segment. The EEPROM Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place constants at specific locations in the EEPROM memory. The directive does not take any parameters.
Syntax:
.ESEG
Example:

```
.DSEG ; Start data segment
vartab: .BYTE 4 ; Reserve 4 bytes in SRAM
.ESEG
eevar: .DW 0xff0f ; Initialize one word in
     ; EEPROM

.CSEG ; Start code segment
const: .DW 2 ; Write 0x0002 in prog.mem.
mov r1,r0 ; Do something
```

18.2.8 EXIT - Exit this file

The EXIT directive tells the Assembler to stop assembling the file. Normally, the Assem- bler runs until end of file (EOF). If an EXIT directive appears in an included file, the Assembler continues from the line following the INCLUDE directive in the file containing the INCLUDE directive.
Syntax:
.EXIT
Example:

```
.EXIT ; Exit this file
```

18.2.9 ORG - Set program origin

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, then it is the EEPROM location counter which is set. If the directive is preceded by a label (on the same source code line), the label will be given the value of the parameter. The default values of the Code and EEPROM location counters are zero, whereas the default value of the SRAM location counter is 32 (due to the registers occupying addresses 0-31) when the assembling is started. Note that the EEPROM and SRAM location counters count bytes whereas the Program memory location counter counts words.
Syntax:
.ORG expression
Example:

```
1 .DSEG ; Start data segment
2 .ORG 0x67 ; Set SRAM address to hex 67
3 variable:.BYTE 1 ; Reserve a byte at SRAM adr.67H
4 .ESEG ; Start EEPROM Segment
5 .ORG 0x20 ; Set EEPROM location counter
6 eevar: .DW 0xfeff ; Initialize one word
7 .CSEG
8 .ORG 0x10 ; Set Program Counter to hex 10
9 mov r0,r1 ; Do something
```

Part III The AVR Architecture

CHAPTER

19 Overview

The AVR microcontrollers are based on the advanced RISC architecture and consist of 32 x 8-bit general purpose working registers. Within one single clock cycle, AVR can take inputs from two general purpose registers and present them to ALU for carrying out the requested operation, and transfer back the result to an arbitrary register.

The ALU can perform arithmetic as well as logical operations over the inputs from the register or between the register and a constant. Single register operations like taking a complement can also be executed in ALU.

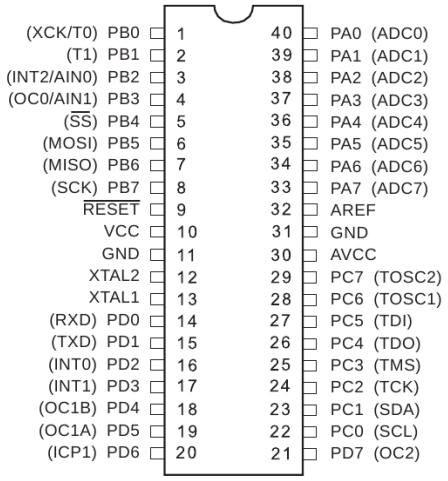
AVR follows Harvard Architecture format in which the processor is equipped with separate memories and buses for Program Memory (flash based) and the Data Memory (SRAM based).

Features of the ATMega16 Include:

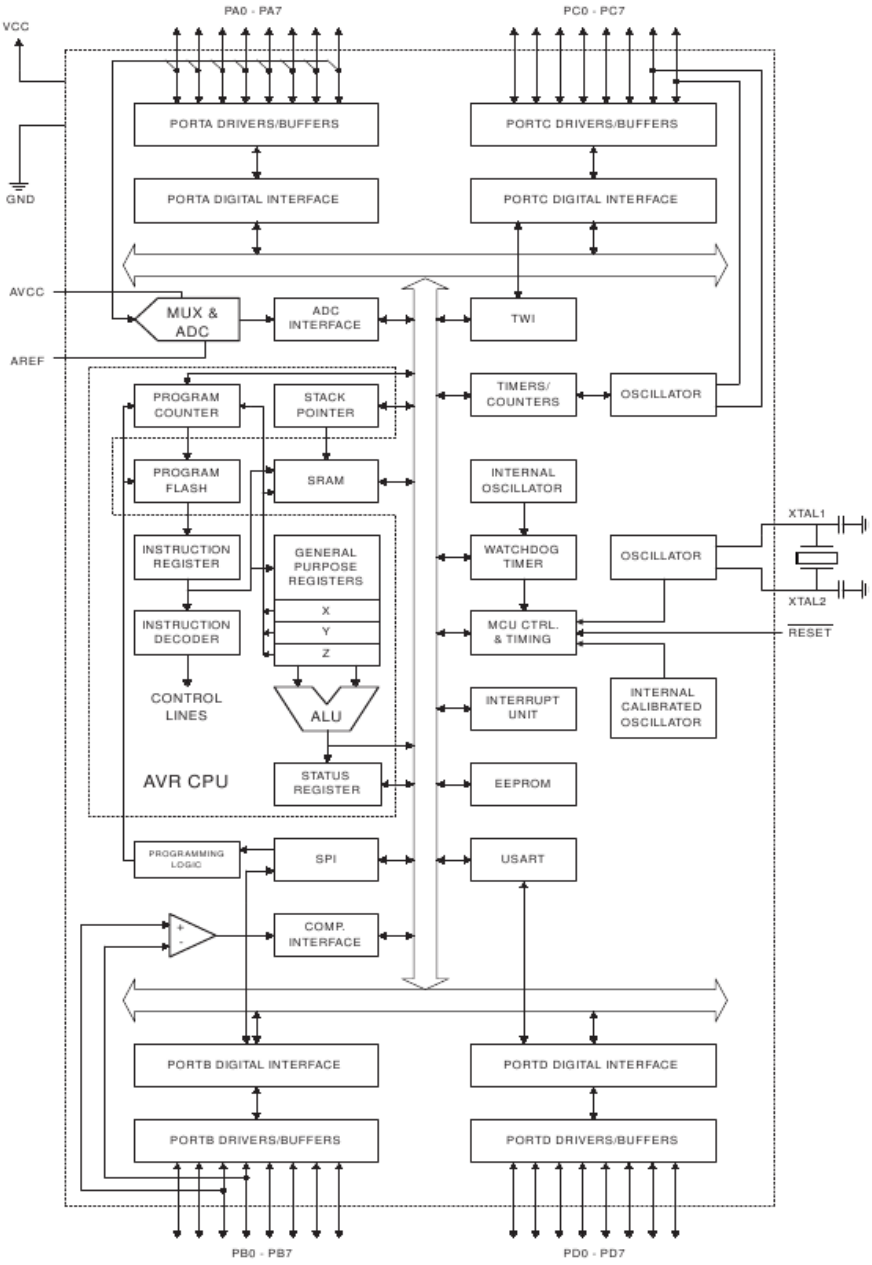
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single-clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 16 MIPS Throughput at 16 MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory segments
  - 16K Bytes of In-System Self-programmable Flash program memory
  - 512 Bytes EEPROM
  - 1K Byte Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C
  - Optional Boot Code Section with Independent Lock Bits
  - In-System Programming by On-chip Boot Program
  - True Read-While-Write Operation
  - Programming Lock for Software Security
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Four PWM Channels
  - 8-channel, 10-bit ADC

  - Byte-oriented Two-wire Serial Interface
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated RC Oscillator
  - External and Internal Interrupt Sources
  - Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby and Extended Standby

Pinout



Block Diagram



CHAPTER 20

The CPU

20.1 Introduction

This section discusses the AVR core architecture in general. The main function of the CPU core is to ensure correct program execution. The CPU must therefore be able to access memories, perform calculations, control peripherals, and handle interrupts.

20.2 Architectural Overview

In order to maximize performance and parallelism, the AVR uses a Harvard architecture – with separate memories and buses for program and data. Instructions in the program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. The program memory is In-System Reprogrammable Flash memory. The fast-access Register File contains 32 x 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle Arithmetic Logic Unit (ALU) operation. In a typical ALU operation, two operands are output from the Register File, the operation is executed, and the result is stored back in the Register File – in one clock cycle. Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing – enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in Flash Program memory. These added function registers are the 16-bit X-, Y-, and Z-register, described later in this section. The ALU supports arithmetic and logic operations between registers or between a constant and a register. Single register operations can also be executed in the ALU. After an arithmetic operation, the Status Register is updated to reflect information about the result of the operation. Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction.

Program Flash memory space is divided in two sections, the Boot program section and the Application Program section. Both sections have dedicated Lock bits for write and read/write protection. The SPM instruction that writes into the Application Flash memory section must reside in the Boot Program section. During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the Stack. The Stack is effectively allocated in the general data SRAM, and consequently the Stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine (before subroutines or interrupts are executed). The Stack Pointer SP is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes supported in the AVR architecture. The memory spaces in the AVR architecture are all linear and regular memory maps. A flexible interrupt module has its control registers in the I/O space with an additional global interrupt enable bit in the Status Register. All interrupts have a separate interrupt vector in the interrupt vector table. The interrupts have priority in accordance with their interrupt vector position. The lower the interrupt vector address, the higher the priority.

20.3 The Arithmetic Logic Unit

The high-performance AVR ALU operates in direct connection with all the 32 general purpose working registers. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and

an immediate are executed. The ALU operations are divided into three main categories – arithmetic, logical, and bit-functions. Some implementations of the architecture also provide a powerful multiplier supporting both signed/unsigned multiplication and fractional format. See the "Instruction Set" section for a detailed description.

20.3.1 The Status Register

The Status Register contains information about the result of the most recently executed arithmetic instruction. This information can be used for altering program flow in order to perform conditional operations. Note that the Status Register is updated after all ALU operations, as specified in the Instruction Set Reference. This will in many cases remove the need for using the dedicated compare instructions, resulting in faster and more compact code. The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.

The Status Register - SREG - is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**
  The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. 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. See the "Instruction Set Description" for detailed information.

- **Bit 4 – S: Sign Bit, S = N⊕V**
  The S-bit is always an exclusive or between the Negative Flag N and the Two's Complement Overflow Flag V. See the "Instruction Set Description" for detailed information.

- **Bit 3 – V: Two's Complement Overflow Flag**
  The Two's Complement Overflow Flag V supports two's complement arithmetics. See the "Instruction Set Description" for detailed information.

- **Bit 2 – N: Negative Flag**
  The Negative Flag N indicates a negative result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 1 – Z: Zero Flag**
  The Zero Flag Z indicates a zero result in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

- **Bit 0 – C: Carry Flag**
  The Carry Flag C indicates a carry in an arithmetic or logic operation. See the "Instruction Set Description" for detailed information.

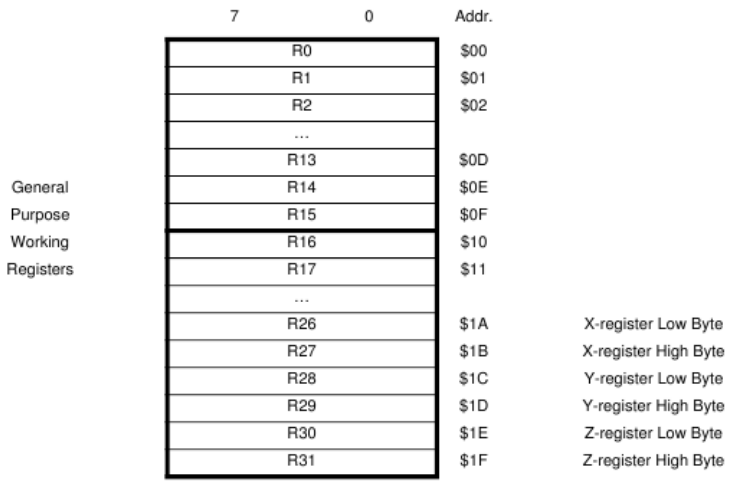
20.3.2 General Purpose Registers

The General Purpose Registers are optimized for the AVR Enhanced RISC instruction set. In order to achievethe required performance and flexibility, the following input/output schemes are supported by the Registers:

- One 8-bit output operand and one 8-bit result input

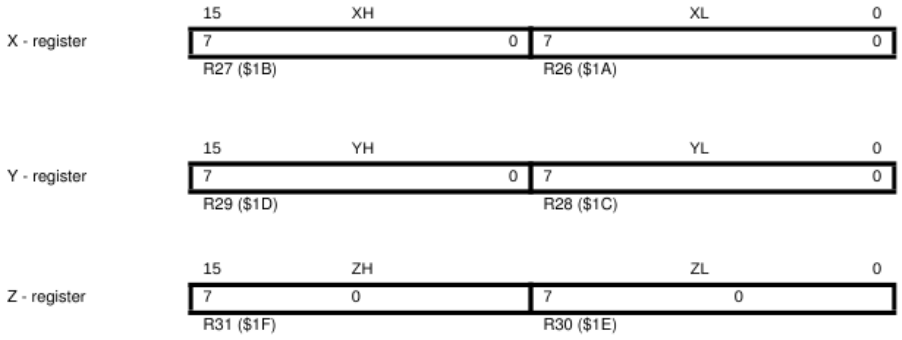
- Two 8-bit output operands and one 8-bit result input
- Two 8-bit output operands and one 16-bit result input
- One 16-bit output operand and one 16-bit result input

The structure of these registers is shown below:



20.3.3 The X, Y and Z Registers

The registers R26..R31 have some added functions to their general purpose usage. These registers are 16-bit address pointers for indirect addressing of the Data Space. The three indirect address registers X, Y, and Z are defined as described below:



20.3.4 Stack Pointer

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. The Stack Pointer Register always points to the top of the Stack. Note that the Stack is implemented as growing from higher memory locations to lower memory locations. This implies that a Stack PUSH command decreases the Stack Pointer. If software reads the Program Counter from the Stack after a call or an interrupt, unused bits (15:13) should be masked out. The Stack Pointer points to the data SRAM Stack area where the Subroutine and Interrupt Stacks are located. This Stack space in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. The Stack Pointer must be set to point above 0x60. The Stack Pointer is decremented by one when data is pushed onto the Stack with the PUSH instruction, and it is decremented by two when the return address is pushed onto the Stack with subroutine call or interrupt. The Stack Pointer is incremented by one when data is popped from the Stack with the POP instruction, and it is incremented by two when data is popped from the Stack with return from subroutine RET or return from interrupt RETI. The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space. The number of bits actually used is implementation dependent. Note that the data space in some implementations of the AVR architecture is so small that only SPL is needed. In this case, the SPH Register will not be present.

Bit	15	14	13	12	11	10	9	8	
	SP15	**SP14**	**SP13**	**SP12**	**SP11**	**SP10**	**SP9**	**SP8**	**SPH**
	SP7	**SP6**	**SP5**	**SP4**	**SP3**	**SP2**	**SP1**	**SP0**	**SPL**
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

CHAPTER 21

Interrupts

21.1 Reset and Interrupt Handling

The AVR provides several different interrupt sources. These interrupts and the separate reset vector each have a separate program vector in the program memory space. All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt.

The lowest addresses in the program memory space are by default defined as the Reset and Interrupt Vectors. The complete list of vectors is shown later. The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0 etc.

When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a Return from Interrupt instruction – RETI – is executed.

There are basically two types of interrupts. The first type is triggered by an event that sets the Interrupt Flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vec- tor in order to execute the interrupt handling routine, and hardware clears the corresponding Interrupt Flag. Interrupt Flags can also be cleared by writing a logic one to the flag bit position(s) to be cleared. If an interrupt condition occurs while the corresponding interrupt enable bit is cleared, the Interrupt Flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software. Similarly, if one or more interrupt conditions occur while the Global Interrupt Enable bit is cleared, the corresponding Interrupt Flag(s) will be set and remembered until the global interrupt enable bit is set, and will then be executed by order of priority.

The second type of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have Interrupt Flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered.

When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served. 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. When using the CLI instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction. The following example shows how this can be used to avoid interrupts during the timed EEPROM write sequence.

```
1  in r16, SREG ; store SREG value
2  cli ; disable interrupts during timed sequence
3  sbi EECR, EEMWE ; start EEPROM write
4  sbi EECR, EEWE
5  out SREG, r16 ; restore SREG value (I-bit)
```

When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, as shown in this example.

```
1  sei ; set global interrupt enable
2  sleep ; enter sleep, waiting for interrupt
3     ; note: will enter sleep before any pending interrupt(s)
```

21.2 Interrupt response Time

The interrupt execution response for all the enabled AVR interrupts is four clock cycles minimum. After four clock cycles the program vector address for the actual interrupt handling routine is executed. During this four clock cycle period, the Program Counter is pushed onto the Stack. The vector is normally a jump to the interrupt routine, and this jump takes three clock cycles. If an interrupt occurs during execution of a multi-cycle instruction, this instruction is completed before the interrupt is served. If an interrupt occurs when the MCU is in sleep mode, the interrupt execution response time is increased by four clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.

A return from an interrupt handling routine takes four clock cycles. During these four clock cycles, the Program Counter (two bytes) is popped back from the Stack, the Stack Pointer is incremented by two, and the I-bit in SREG is set.

CHAPTER 22

Memory

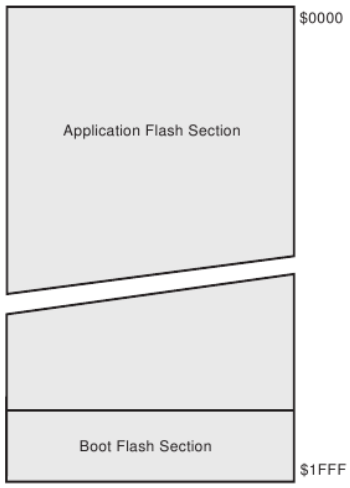
This section describes the different memories in the ATmega16. The AVR architecture has two main memory spaces, the Data Memory and the Program Memory space. In addition, the ATmega16 features an EEPROM Memory for data storage.

22.1 Program Memory

The ATmega16 contains 16K bytes On-chip In-System Reprogrammable Flash memory for pro- gram storage. Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 8K x 16. For software security, the Flash Program memory space is divided into two sections, Boot Program section and Application Program section.

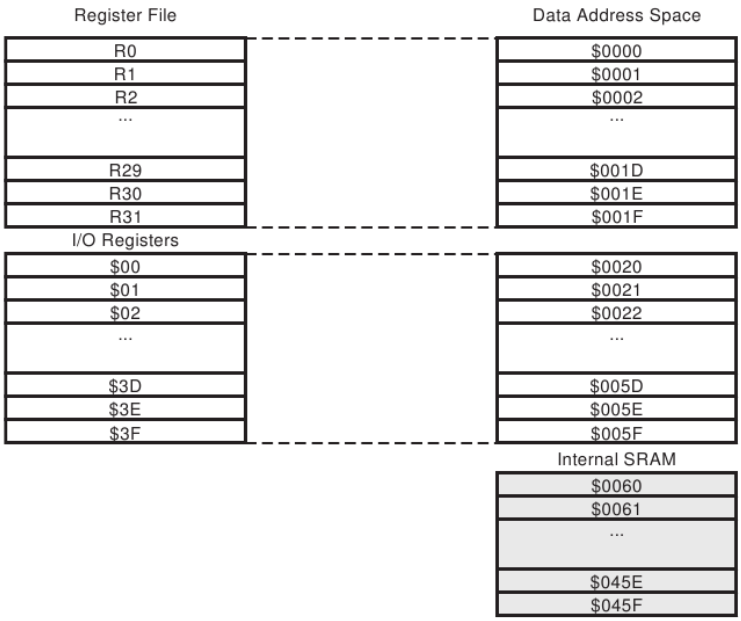
The Flash memory has an endurance of at least 10,000 write/erase cycles. The ATmega16 Program Counter (PC) is 13 bits wide, thus addressing the 8K program memory locations. The operation of Boot Program section and associated Boot Lock bits for software protection are described in detail in “Boot Loader Support – Read-While-Write Self-Programming” on page 246 of the datasheet.

The memory map is shown below.



22.2 SRAM

The diagram below shows the organisation of the SRAM.



The lower 1120 Data Memory locations address the General Purpose Registers (R0-R31), the I/O Memory, and the internal data SRAM. The first 96 locations address the General Purpose Registers (R0-R31) and I/O Memory, and the next 1024 locations address the internal data SRAM.

The five different addressing modes for the data memory cover: Direct, Indirect with Displace- ment, Indirect, Indirect with Pre-decrement, and Indirect with Post-increment. In the General Purpose Registers (R0-R31), registers R26 to R31 feature the indirect addressing pointer registers. The direct addressing reaches the entire data space. The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y- or Z-register. When using register indirect addressing modes with automatic pre-decrement and post-incre- ment, the address registers X, Y, and Z are decremented or incremented.

The 32 general purpose working registers, 64 I/O Registers, and the 1024 bytes of internal data SRAM in the ATmega16 are all accessible through all these addressing modes.

22.3 EEPROM

The ATmega16 contains 512 bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The CPU acesses the EEPROM data as a peripheral using the EEPROM registers in the IO space.

22.3.1 EEPROM Read/Write Access

The EEPROM Access Registers are accessible in the I/O space.

In order to prevent unintentional EEPROM writes, a specific write procedure must be followed. Refer to the description of the EEPROM Control Register for details on this. When the EEPROM is read, the CPU is halted for four clock cycles before the next instruction is executed. When the EEPROM is written, the CPU is halted for two clock cycles before the next instruction is executed.

22.3.2 EEPROM Address Register

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	–	EEAR8	EEARH
	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	X	
	X	X	X	X	X	X	X	X	

- **Bits 15..9 – Res: Reserved Bits**
  These bits are reserved bits in the ATmega16 and will always read as zero.
- **Bits 8..0 – EEAR8..0: EEPROM Address**
  The EEPROM Address Registers – EEARH and EEARL – specify the EEPROM address in the 512 bytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 511. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

22.3.3 EEPROM Data Register

Bit	7	6	5	4	3	2	1	0	
	MSB							LSB	EEDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..0 – EEDR7.0: EEPROM Data**
  For the EEPROM write operation, the EEDR Register contains the data to be written to the EEPROM in the address given by the EEAR Register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

22.3.4 EEPROM Control Register

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	EERIE	EEMWE	EEWE	EERE	EECR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	X	0	

- **Bits 7..4 – Res: Reserved Bits**
  These bits are reserved bits in the ATmega16 and will always read as zero.
- **Bit 3 – EERIE: EEPROM Ready Interrupt Enable**
  Writing EERIE to one enables the EEPROM Ready Interrupt if the I bit in SREG is set. Writing EERIE to zero disables the interrupt. The EEPROM Ready interrupt generates a constant inter- rupt when EEWE is cleared.
- **Bit 2 – EEMWE: EEPROM Master Write Enable** The EEMWE bit determines whether setting EEWE to one causes the EEPROM to be written. When EEMWE is set, setting EEWE within four clock cycles will write data to the EEPROM at the selected address If EEMWE is zero, setting EEWE will have no effect. When EEMWE has been written to one by software, hardware clears the bit to zero after four clock cycles. See the description of the EEWE bit for an EEPROM write procedure.
- **Bit 1 – EEWE: EEPROM Write Enable**
  The EEPROM Write Enable Signal EEWE is the write strobe to the EEPROM. When address and data are correctly set up, the EEWE bit must be written to one to write the value into the EEPROM. The EEMWE bit must be written to one before a logical one is written to EEWE, oth- erwise no EEPROM write takes place. The following procedure should be followed when writing the EEPROM (the order of steps 3 and 4 is not essential):

  1. Wait until EEWE becomes zero.
  2. Wait until SPMEN in SPMCR becomes zero.
  3. Write new EEPROM address to EEAR (optional).
  4. Write new EEPROM data to EEDR (optional).
  5. Write a logical one to the EEMWE bit while writing a zero to EEWE in EECR.
  6. Within four clock cycles after setting EEMWE, write a logical one to EEWE.

  The EEPROM can not be programmed during a CPU write to the Flash memory. The software must check that the Flash programming is completed before initiating a new EEPROM write. Step 2 is only relevant if the software contains a Boot Loader allowing the CPU to program the Flash. If the Flash is never being updated by the CPU, step 2 can be omitted.

Caution: An interrupt between step 5 and step 6 will make the write cycle fail, since the EEPROM Master Write Enable will time-out. If an interrupt routine accessing the EEPROM is interrupting another EEPROM Access, the EEAR or EEDR reGister will be modified, causing the interrupted EEPROM Access to fail. It is recommended to have the Global Interrupt Flag cleared during all the steps to avoid these problems. When the write access time has elapsed, the EEWE bit is cleared by hardware. The user soft- ware can poll this bit and wait for a zero before writing the next byte. When EEWE has been set, the CPU is halted for two cycles before the next instruction is executed.

- **Bit 0 – EERE: EEPROM Read Enable**
  The EEPROM Read Enable Signal – EERE – is the read strobe to the EEPROM. When the cor- rect address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed. The user should poll the EEWE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM, nor to change the EEAR Register. The calibrated Oscillator is used to time the EEPROM accesses.

The following code examples show an assembly function for writing to / rweading from the EEPROM. The examples assume that interrupts are controlled (for example by disabling interrupts globally) so that no interrupts will occur during execution of these functions.

```
EEPROM_read:
SBIC EERC, EEWE ; Wait for completion of previosbic EECR,EEWE
rjmp EEPROM_read
; Set up address (r18:r17) in address register
out EEARH, r18
out EEARL, r17
; Start eeprom read by writing
sbi EECR,EERE
; Read data from data register
in r16,EEDR
ret
```

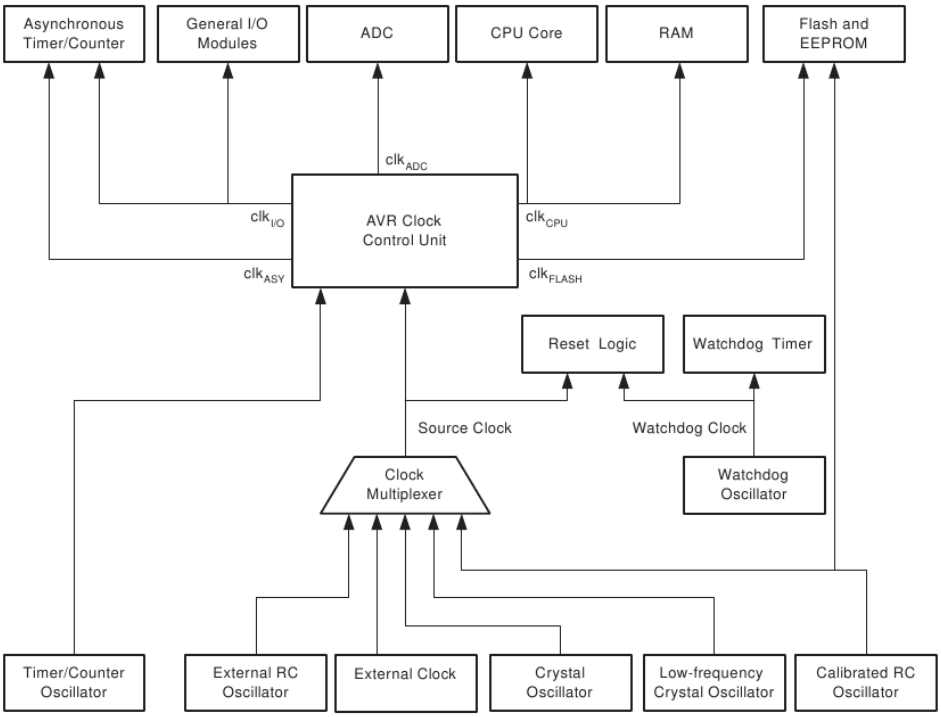
```
EEPROM_write:
sbic EECR,EEWE ; Wait for completion of previous write
rjmp EEPROM_write
; Set up address (r18:r17) in address register
out EEARH, r18
out EEARL, r17
; Write data (r16) to data register
out EEDR,r16
; Write logical one to EEMWE
sbi EECR,EEMWE
; Start eeprom write by setting EEWE
sbi EECR,EEWE
ret
```

CHAPTER 23

System Clock and Clock Options

23.1 Clock systems and their Distribution

The figure below presents the principal clock systems in the AVR and their distribution. All of the clocks need not be active at a given time. In order to reduce power consumption, the clocks to modules not being used can be halted by using different sleep modes, as described in “Power Manage- ment and Sleep Modes”.



- **CPU Clock – clk$_{CPU}$**
  The CPU clock is routed to parts of the system concerned with operation of the AVR core. Examples of such modules are the General Purpose Register File, the Status Register and the data memory holding the Stack Pointer. Halting the CPU clock inhibits the core from performing general operations and calculations.

- **I/O Clock – clk$_{I/O}$**
  The I/O clock is used by the majority of the I/O modules, like Timer/Counters, SPI, and USART. The I/O clock is also used by the External Interrupt module, but note that some external inter- rupts are detected by asynchronous logic, allowing such interrupts to be detected even if the I/O clock is halted. Also note that address recognition in the TWI module is carried out asynchro- nously when clk$_{I/O}$ is halted, enabling TWI address reception in all sleep modes.

- **Flash Clock – clk$_{FLASH}$**
  The Flash clock controls operation of the Flash interface. The Flash clock is usually active simul- taneously with the CPU clock.
- **Asynchronous Timer Clock – clk$_{ASY}$**
  The Asynchronous Timer clock allows the Asynchronous Timer/Counter to be clocked directly from an external 32 kHz clock crystal. The dedicated clock domain allows using this Timer/Counter as a real-time counter even when the device is in sleep mode.

23.2 Clock Sources

The device has the following clock source options, selectable by Flash Fuse bits as shown below. The clock from the selected source is input to the AVR clock generator, and routed to the appropriate modules.

Device Clocking Option	CKSEL3..0
External Crystal/Ceramic Resonator	1111 - 1010
External Low-frequency Crystal	1001
External RC Oscillator	1000 - 0101
Calibrated Internal RC Oscillator	0100 - 0001
External Clock	0000

Note: 1. For all fuses “1” means unprogrammed while “0” means programmed.

On our devboard the internal calibrated 8MHz RC oscillator is used.

23.2.1 Calibrated Internal RC Oscillator

The Calibrated Internal RC Oscillator provides a fixed 1.0, 2.0, 4.0, or 8.0 MHz clock. All frequencies are nominal values at 5V and 25°C. This clock may be selected as the system clock by programming the CKSEL Fuses. If selected, it will operate with no external components. The CKOPT Fuse should always be unpro-grammed when using this clock option. During Reset, hardware loads the calibration byte into the OSCCAL Register and thereby automatically calibrates the RC Oscillator. At 5V, 25°C and 1.0, 2.0, 4.0 or 8.0 MHz Oscillator frequency selected, this calibration gives a frequency within ± 3% of the nominal frequency. Using calibration methods as described in application notes available at www.atmel.com/avr it is possible to achieve ±1% accuracy at any given VCC and Temperature. When this Oscillator is used as the Chip Clock, the Watchdog Oscillator will still be used for the Watchdog Timer and for the reset time-out. For more information on the pre-programmed calibration value, see the section “Calibration Byte” on page 261 of the datasheet.

CHAPTER 24

Power Management and Sleep Modes

Sleep modes enable the application to shut down unused modules in the MCU, thereby saving power. The AVR provides various sleep modes allowing the user to tailor the power consumption to the application's requirements.

To enter any of the six sleep modes, the SE bit in MCUCR must be written to logic one and a SLEEP instruction must be executed. The SM2, SM1, and SM0 bits in the MCUCR Register select which sleep mode (Idle, ADC Noise Reduction, Power-down, Power-save, Standby, or Extended Standby) will be activated by the SLEEP instruction. See the table below for a summary. If an enabled interrupt occurs while the MCU is in a sleep mode, the MCU wakes up. The MCU is then halted for four cycles in addition to the start-up time, it executes the interrupt routine, and resumes execution from the instruction following SLEEP. The contents of the General Purpose Registers and SRAM are unaltered when the device wakes up from sleep. If a Reset occurs during sleep mode, the MCU wakes up and executes from the Reset Vector.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

MCU Control Register– MCUCR

The MCU Control Register contains control bits for power management.

- **Bits 7, 5, 4 – SM2..0: Sleep Mode Select Bits 2, 1, and 0**
  These bits select between the six available sleep modes as shown in Table 13.

Bit	7	6	5	4	3	2	1	0	
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 6 – SE: Sleep Enable**
  The SE bit must be written to logic one to make the MCU enter the sleep mode when the SLEEP instruction is executed. To avoid the MCU entering the sleep mode unless it is the programmers purpose, it is recommended to write the Sleep Enable (SE) bit to one just before the execution of the SLEEP instruction and to clear it immediately after waking up.

24.1 Idle Mode

When the SM2..0 bits are written to 000, the SLEEP instruction makes the MCU enter Idle mode, stopping the CPU but allowing SPI, USART, Analog Comparator, ADC, Two-wire Serial Interface, Timer/Counters, Watchdog, and the interrupt system to continue operating. This sleep mode basically halts clk_{CPU} and clk_{FLASH}, while allowing the other clocks to run. Idle mode enables the MCU to wake up from external triggered interrupts as well as internal ones like the Timer Overflow and USART Transmit Complete interrupts. If wake-up from

the Analog Comparator interrupt is not required, the Analog Comparator can be powered down by setting the ACD bit in the Analog Comparator Control and Status Register – ACSR. This will reduce power consumption in Idle mode. If the ADC is enabled, a conversion starts automatically when this mode is entered.

24.2 ADC Noise Reduction Mode

When the SM2..0 bits are written to 001, the SLEEP instruction makes the MCU enter ADC Noise Reduction mode, stopping the CPU but allowing the ADC, the External Interrupts, the Two-wire Serial Interface address watch, Timer/Counter2 and the Watchdog to continue operat- ing (if enabled). This sleep mode basically halts $\text{clk}_{I/O}$, clk_{CPU}, and clk_{FLASH}, while allowing the other clocks to run.

This improves the noise environment for the ADC, enabling higher resolution measurements. If the ADC is enabled, a conversion starts automatically when this mode is entered. Apart form the ADC Conversion Complete interrupt, only an External Reset, a Watchdog Reset, a Brown-out Reset, a Two-wire Serial Interface Address Match Interrupt, a Timer/Counter2 interrupt, an SPM/EEPROM ready interrupt, an External level interrupt on INT0 or INT1, or an external interrupt on INT2 can wake up the MCU from ADC Noise Reduction mode.

24.3 Power Down Mode

When the SM2..0 bits are written to 010, the SLEEP instruction makes the MCU enter Powerdown mode. In this mode, the External Oscillator is stopped, while the External interrupts, the Two-wire Serial Interface address watch, and the Watchdog continue operating (if enabled). Only an External Reset, a Watchdog Reset, a Brown-out Reset, a Two-wire Serial Interface address match interrupt, an External level interrupt on INT0 or INT1, or an External interrupt on INT2 can wake up the MCU. This sleep mode basically halts all generated clocks, allowing operation of asynchronous modules only.

Note that if a level triggered interrupt is used for wake-up from Power-down mode, the changed level must be held for some time to wake up the MCU. Refer to “External Interrupts” section for details.

When waking up from Power-down mode, there is a delay from the wake-up condition occurs until the wake-up becomes effective. This allows the clock to restart and become stable after having been stopped. The wake-up period is defined by the same CKSEL Fuses that define the reset time-out period, as described in “Clock Sources” on page 25.

24.4 Power Save Mode

When the SM2..0 bits are written to 011, the SLEEP instruction makes the MCU enter Powersave mode. This mode is identical to Power-down, with one exception: If Timer/Counter2 is clocked asynchronously, i.e., the AS2 bit in ASSR is set, Timer/Counter2 will run during sleep. The device can wake up from either Timer Overflow or Output Compare event from Timer/Counter2 if the corresponding Timer/Counter2 interrupt enable bits are set in TIMSK, and the Global Interrupt Enable bit in SREG is set.

If the Asynchronous Timer is NOT clocked asynchronously, Power-down mode is recommended instead of Power-save mode because the contents of the registers in the Asynchronous Timer should be considered undefined after wake-up in Power-save mode if AS2 is 0. This sleep mode basically halts all clocks except clk_{ASY}, allowing operation only of asynchronous modules, including Timer/Counter2 if clocked asynchronously.

24.5 Standby Mode

When the SM2..0 bits are 110 and an external crystal/resonator clock option is selected, the SLEEP instruction makes the MCU enter Standby mode. This mode is identical to Power-down with the exception that the Oscillator is kept running. From Standby mode, the device wakes up in six clock cycles.

24.6 Extended Standby Mode

When the SM2..0 bits are 111 and an external crystal/resonator clock option is selected, the SLEEP instruction makes the MCU enter Extended Standby mode. This mode is identical to Power-save mode with the exception that the Oscillator is kept running. From Extended Standby mode, the device wakes up in six clock cycles..

	Active Clock domains					Oscillators		Wake-up Sources					
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Osc. Enabled	INT2 INT1 INT0	TWI Address Match	Timer 2	SPM / EEPROM Ready	ADC	Other I/O
Idle			X	X	X	X	X(2)	X	X	X	X	X	X
ADC Noise Reduction				X	X	X	X(2)	X(3)	X	X	X	X	
Power Down								X(3)	X				
Power Save					X(2)		X(2)	X(3)	X	X(2)			
Standby(1)						X		X(3)	X				
Extended Standby(1)					X(2)	X	X(2)	X(3)	X	X(2)			

Notes: 1. External Crystal or resonator selected as clock source.
2. If AS2 bit in ASSR is set.
3. Only INT2 or level interrupt INT1 and INT0.

24.7 Minimising Power Consumption

There are several issues to consider when trying to minimize the power consumption in an AVR controlled system. In general, sleep modes should be used as much as possible, and the sleep mode should be selected so that as few as possible of the device's functions are operating. All functions not needed should be disabled. In particular, the following modules may need special consideration when trying to achieve the lowest possible power consumption.

24.7.1 ADC

If enabled, the ADC will be enabled in all sleep modes. To save power, the ADC should be disabled before entering any sleep mode. When the ADC is turned off and on again, the next conversion will be an extended conversion. Refer to "Analog to Digital Converter" section for more information.

24.7.2 Analog Comparator

When entering Idle mode, the Analog Comparator should be disabled if not used. When entering ADC Noise Reduction mode, the Analog Comparator should be disabled. In the other sleep modes, the Analog Comparator is automatically disabled. However, if the Analog Comparator is set up to use the Internal Voltage Reference as input, the Analog Comparator should be dis- abled in all sleep modes. Otherwise, the Internal Voltage Reference will be enabled, independent of sleep mode.

24.7.3 Brown Out Detector

If the Brown-out Detector is not needed in the application, this module should be turned off. If the Brown-out Detector is enabled by the BODEN Fuse, it will be enabled in all sleep modes, and hence, always consume power. In the deeper sleep modes, this will contribute significantly to the total current consumption.

24.7.4 Internal Voltage Reference

The Internal Voltage Reference will be enabled when needed by the Brown-out Detector, the Analog Comparator or the ADC. If these modules are disabled as described in the sections above, the internal voltage reference will be disabled and it will not be consuming power. When turned on again, the user must allow the reference to start up before the output is used. If the reference is kept on in sleep mode, the output can be used immediately.

24.7.5 Watchdog Timer

If the Watchdog Timer is not needed in the application, this module should be turned off. If the Watchdog Timer is enabled, it will be enabled in all sleep modes, and hence, always consume power. In the deeper sleep modes, this will contribute significantly to the total current consump- tion.

24.7.6 Port Pins

When entering a sleep mode, all port pins should be configured to use minimum power. The most important thing is then to ensure that no pins drive resistive loads. In sleep modes where the both the I/O clock ($clk_{I/O}$) and the ADC clock (clk_{ADC}) are stopped, the input buffers of the device will be disabled. This ensures that no power is consumed by the input logic when not needed. In some cases, the input logic is needed for detecting wake-up conditions, and it will then be enabled. If the input buffer is enabled and the input signal is left floating or have an analog signal level close to VCC/2, the input buffer will use excessive power.

CHAPTER 25

System Control And Reset

25.1 Resetting the MCU

During Reset, all I/O Registers are set to their initial values, and the program starts execution from the Reset Vector. The instruction placed at the Reset Vector must be a JMP – absolute jump – instruction to the reset handling routine. If the program never enables an interrupt source, the Interrupt Vectors are not used, and regular program code can be placed at these locations. This is also the case if the Reset Vector is in the Application section while the Interrupt Vectors are in the Boot section or vice versa.

The I/O ports of the AVR are immediately reset to their initial state when a reset source goes active. This does not require any clock source to be running. After all reset sources have gone inactive, a delay counter is invoked, stretching the Internal Reset. This allows the power to reach a stable level before normal operation starts. The time-out period of the delay counter is defined by the user through the CKSEL Fuses. The different selec- tions for the delay period are presented in “Clock Sources” on page 25 of the Datasheet.

25.2 Reset Sources

The ATmega16 has five sources of reset:

- Power-on Reset. The MCU is reset when the supply voltage is below the Power-on Reset threshold (VPOT).
- External Reset. The MCU is reset when a low level is present on the RESET pin for longer than the minimum pulse length.
- Watchdog Reset. The MCU is reset when the Watchdog Timer period expires and the Watchdog is enabled.
- Brown-out Reset. The MCU is reset when the supply voltage VCC is below the Brown-out Reset threshold (VBOT) and the Brown-out Detector is enabled.
- JTAG AVR Reset. The MCU is reset as long as there is a logic one in the Reset Register, one of the scan chains of the JTAG system.

25.2.1 Power-On Reset

A Power-on Reset (POR) pulse is generated by an On-chip detection circuit. The POR is activated whenever VCC is below the detection level. The POR circuit can be used to trigger the Start-up Reset, as well as to detect a failure in supply voltage. A Power-on Reset (POR) circuit ensures that the device is reset from Power-on. Reaching the Power-on Reset threshold voltage invokes the delay counter, which determines how long the device is kept in RESET after VCC rise. The RESET signal is activated again, without any delay, when VCC decreases below the detection level.

25.2.2 External Reset

An External Reset is generated by a low level on the RESET pin. Reset pulses longer than the minimum pulse width will generate a reset, even if the clock is not running. Shorter pulses are not guaranteed to generate a

reset. When the applied signal reaches the Reset Threshold Voltage – V_{RST} – on its positive edge, the delay counter starts the MCU after the Time-out period t_{TOUT} has expired.

25.2.3 Brown-Own Reset

ATmega16 has an On-chip Brown-out Detection (BOD) circuit for monitoring the VCC level during operation by comparing it to a fixed trigger level. The trigger level for the BOD can be selected by the fuse BODLEVEL to be 2.7V (BODLEVEL unprogrammed), or 4.0V (BODLEVEL programmed). The trigger level has a hysteresis to ensure spike free Brown-out Detection. The BOD circuit can be enabled/disabled by the fuse BODEN. When the BOD is enabled (BODEN programmed), and VCC decreases to a value below the trigger level, the Brown-out Reset is immediately activated. When VCC increases above the trigger level, the delay counter starts the MCU after the Time-out period t_{TOUT} has expired. The BOD circuit will only detect a drop in VCC if the voltage stays below the trigger level for longer than t_{BOD}.

25.2.4 Watchdog Reset

When the Watchdog times out, it will generate a short reset pulse of one CK cycle duration. On the falling edge of this pulse, the delay timer starts counting the Time-out period t_{TOUT}. Refer to the section on peripherals for details on operation of the Watchdog Timer.

MCU Control and Status Register – MCUCSR

Bit	7	6	5	4	3	2	1	0	
	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	MCUCSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	See Bit Description					

- **Bit 4 – JTRF: JTAG Reset Flag**
  This bit is set if a reset is being caused by a logic one in the JTAG Reset Register selected by the JTAG instruction AVR_RESET. This bit is reset by a Power-on Reset, or by writing a logic zero to the flag.
- **Bit 3 – WDRF: Watchdog Reset Flag**
  This bit is set if a Watchdog Reset occurs. The bit is reset by a Power-on Reset, or by writing a logic zero to the flag.
- **Bit 2 – BORF: Brown-out Reset Flag**
  This bit is set if a Brown-out Reset occurs. The bit is reset by a Power-on Reset, or by writing a logic zero to the flag.
- **Bit 1 – EXTRF: External Reset Flag**
  This bit is set if an External Reset occurs. The bit is reset by a Power-on Reset, or by writing a logic zero to the flag.
- **Bit 0 – PORF: Power-on Reset Flag**
  This bit is set if a Power-on Reset occurs. The bit is reset only by writing a logic zero to the flag.

To make use of the Reset Flags to identify a reset condition, the user should read and then reset the MCUCSR as early as possible in the program. If the register is cleared before another reset occurs, the source of the reset can be found by examining the Reset Flags.

CHAPTER

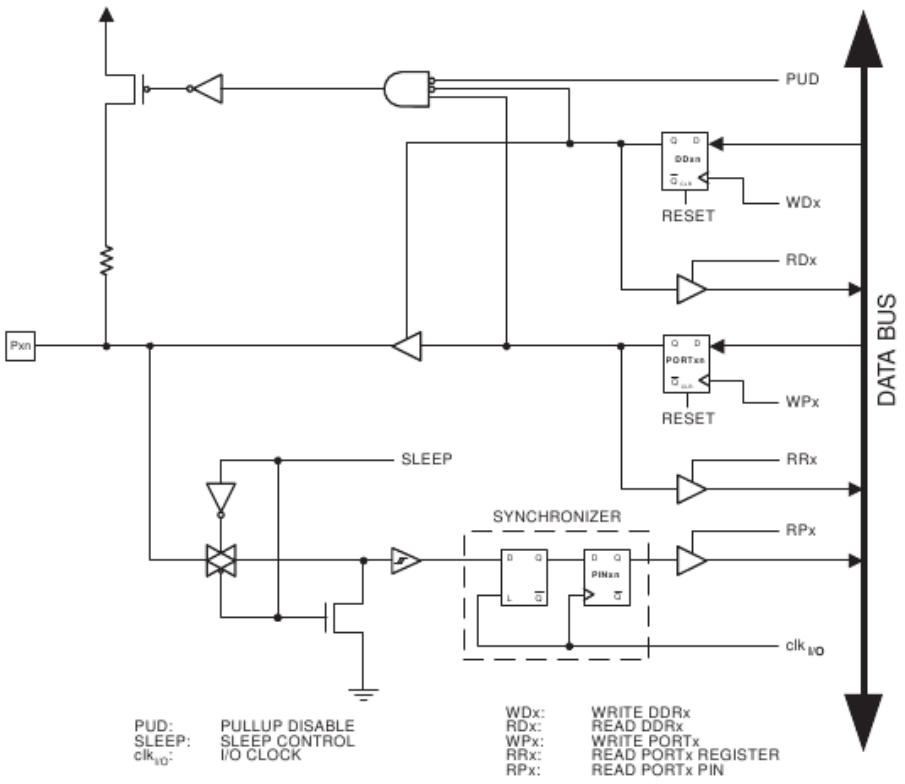
26 IO Ports

All AVR ports have true Read-Modify-Write functionality when used as general digital I/O ports. This means that the direction of one port pin can be changed without unintentionally changing the direction of any other pin with the SBI and CBI instructions. The same applies when changing drive value (if configured as output) or enabling/disabling of pull-up resistors (if configured as input). Each output buffer has symmetrical drive characteristics with both high sink and source capability. The pin driver is strong enough to drive LED displays directly. All port pins have indi- vidually selectable pull-up resistors with a supply-voltage invariant resistance. All I/O pins have protection diodes to both VCC and Ground.

Three I/O memory address locations are allocated for each port, one each for the Data Register – PORTx, Data Direction Register – DDRx, and the Port Input Pins – PINx. The Port Input Pins I/O location is read only, while the Data Register and the Data Direction Register are read/write. In addition, the Pull-up Disable – PUD bit in SFIOR disables the pull-up function for all pins in all ports when set.

26.1 Ports as General Digital IO

The ports are bi-directional I/O ports with optional internal pull-ups. The figure below shows a functional description of one I/O-port pin, here generically called Pxn.



26.1.1 Configuring a pin

Each port pin consists of three register bits: DDxn, PORTxn, and PINxn. The DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address. The DDxn bit in the DDRx Register selects the direction of this pin. If DDxn is written logic one, Pxn is configured as an output pin. If DDxn is written logic zero, Pxn is configured as an input pin. If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. The port pins are tri-stated when a reset condition becomes active, even if no clocks are running.

If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero). When switching between tri-state (DDxn, PORTxn = 0b00) and output high (DDxn, PORTxn = 0b11), an intermediate state with either pull-up enabled (DDxn, PORTxn = 0b01) or output low (DDxn, PORTxn = 0b10) must occur. Normally, the pull-up enabled state is fully accept- able, as a high-impedance environment will not notice the difference between a strong high driver and a pull-up. If this is not the case, the PUD bit in the SFIOR Register can be set to disable all pull-ups in all ports.

The table below shows all configuration options for a pin.

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

26.1.2 Reading the Pin Value

Independent of the setting of Data Direction bit DDxn, the port pin can be read through the PINxn Register bit.

26.1.3 Alternate Port Functions

Most IO pins have alternate functions associated with peripherals on the Mega16, these alternative functions are described in the section concerning the in dividual peripherals.

CHAPTER 27

Timers / Counters

There always seems to be much confusion about Timer/Counters, but a timer is just a counter that increments its value at a regular interval. The Timer/Counters on the Mega16 can take their input from an external (non-regular) source - i.e. acting as counters, or they can take their input from a regular periodic signal (of which the frequency/period is known) in which case they become timers.

Quite simply the counter counts up (or down, depending on configuration) on every active input (clock) edge.

Timers are usefull for generating delays, counting events over a specified interval and some other elegant solutions to everyday problems.

Timer Setup

Before using a timer the control registers for that timer should be set up according to the desired use. Things such as clock frequency, timer size, timer mode need to be taken into consideration. Specifics for individual timer/counters can be found in the relevant sections below.

27.1 Timer Interrupts

All timers come with various interrupts that are triggered under certain circumstances. These interrupts are individually enabled in the TIMSK IO register. As with all interrupts there are also corresponding interrupt flags, these are found in the TIFR IO register

- **Overflow Interrupt:**
  This interrupt occurs whenever the timer overflows, i.e. when the count value rolls over from the highest value (not necessarily 0xff or 0xffff) to 0.
- **Compare Match Interrupt:**
  This interrupt occurs when the value in the counter matches that stored in the compare match register.
- **Capture Interrupt:**
  This happens when an input capture event happens, the value in the counter at this time is then automatically stored in the Input Capture register.

27.2 Timer Modes

The different timer modes determine how the timer/counter operates.

27.2.1 Normal Mode

The simplest mode of operation is the normal mode. In this mode the counting direction is always up (incrementing), and no counter clear is performed. The counter simply overflows when it passes its maximum 8/16-bit value (TOP = 0xFF/0xFFFF) and then restarts from the bottom (0x00/0x0000).

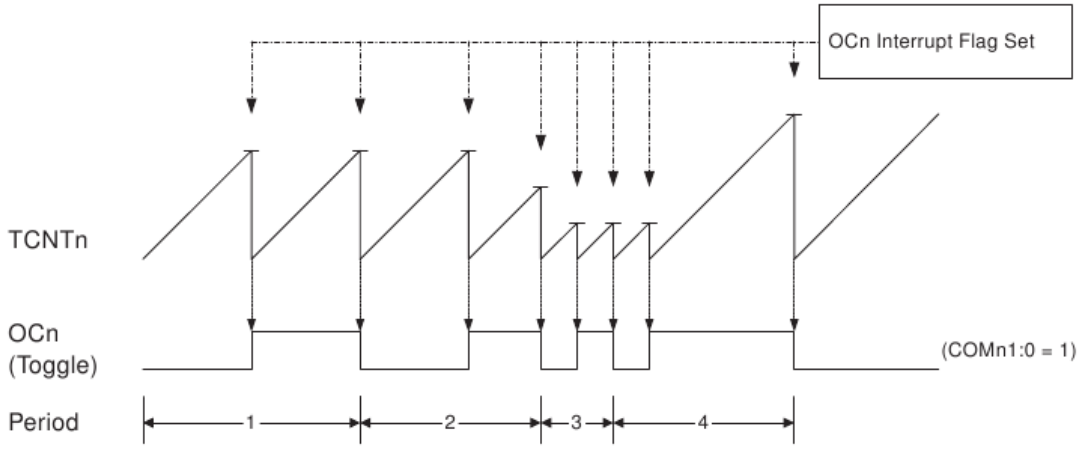
In normal operation the Timer/Counter Overflow Flag (TOVx) will be set in the same timer clock cycle as the TCNTx becomes zero. The TOVx Flag in this case behaves like a ninth bit, except that it is only set, not cleared. However, combined with the timer overflow interrupt that automatically clears the TOVx Flag, the timer resolution can be increased by software. There are no special cases to consider in the normal mode, a new counter value can be written anytime.

The output compare unit can be used to generate interrupts at some given time.

27.2.2 Clear Timer on Compare Match (CTC) Mode

In Clear Timer on Compare or CTC mode, the OCRx Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNTx) matches the OCRx.

The OCRx defines the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events. The timing diagram for the CTC mode is shown below. The counter value (TCNTx) increases until a compare match occurs between TCNTx and OCRx, and then counter (TCNTx) is cleared.



An interrupt can be generated each time the counter value reaches the TOP value by using the OCFx0 Flag. If the interrupt is enabled, the interrupt handler routine can be used for updating the TOP value. However, changing TOP to a value close to BOTTOM when the counter is running with none or a low prescaler value must be done with care since the CTC mode does not have the double buffering feature. If the new value written to OCRn is lower than the current value of TCNTn, the counter will miss the compare match. The counter will then have to count to its max- imum value (0xFF/0xFFFF) and wrap around starting at 0x00 before the compare match can occur.

For generating a waveform output in CTC mode, the OCx output can be set to toggle its logical level on each compare match by setting the Compare Output mode bits to toggle mode. The OCx value will not be visible on the port pin unless the data direction for the pin is set to output. The waveform generated will have a maximum frequency of $f_{OCx} = \frac{f_{clk_{I/O}}}{2}$ when OCRx is set to zero (0x00). The waveform frequency is defined by the following equation:

$$f_{OCn} = \frac{f_{clk_I/O}}{2.N.(1 + OCRn)}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024). As for the Normal mode of operation, the TOVx Flag is set in the same timer clock cycle that the counter counts from MAX to 0x00.

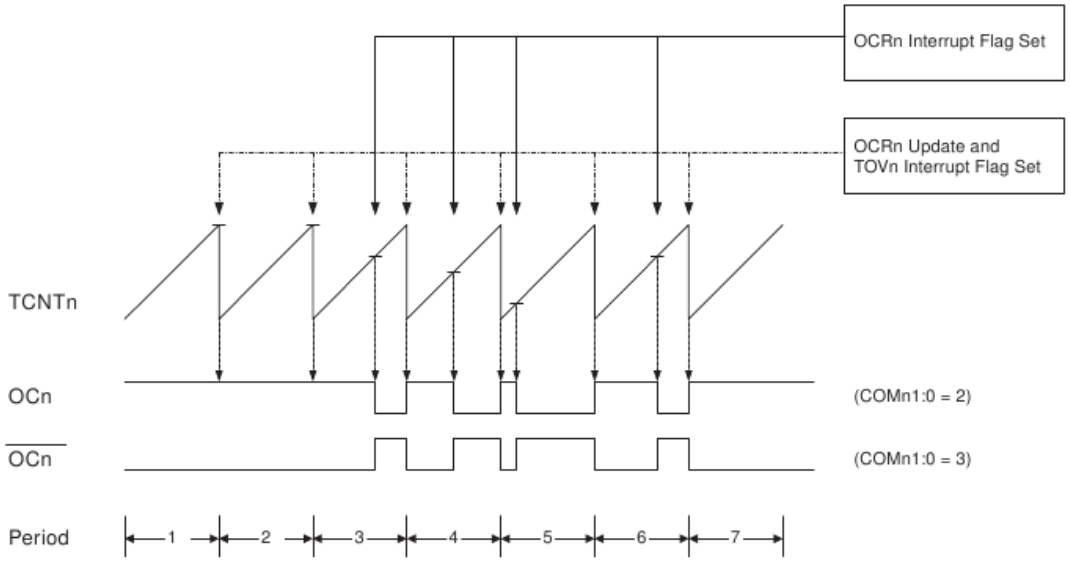
27.2.3 Fast PWM Mode

The fast Pulse Width Modulation or fast PWM mode provides a high frequency PWM waveform generation option. The fast PWM differs from the other PWM option by its single-slope operation. The counter counts from BOTTOM to MAX then restarts from BOTTOM.

In non-inverting Compare Output mode, the Output Compare (OCx) is cleared on the compare match between TCNTx and OCRx, and set at BOTTOM.

In inverting Compare Output mode, the output is set on compare match and cleared at BOTTOM. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct PWM mode that use dual-slope operation. This high frequency makes the fast PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), and therefore reduces total system cost.

In fast PWM mode, the counter is incremented until the counter value matches the MAX value. The counter is then cleared at the following timer clock cycle. The timing diagram for the fast PWM mode is shown below. The TCNTx value is in the timing diagram shown as a histogram for illustrating the single-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNTx slopes represent compare matches between OCRx and TCNTx.



The Timer/Counter Overflow Flag (TOVx) is set each time the counter reaches MAX. If the interrupt is enabled, the interrupt handler routine can be used for updating the compare value.

In fast PWM mode, the compare unit allows generation of PWM waveforms on the OCx pin. The actual OC0 value will only be visible on the port pin if the data direction for the port pin is set as output. The PWM waveform is generated by setting (or clearing) the OCx Register at the compare match between OCRx and TCNTx, and clearing (or setting) the OCx Register at the timer clock cycle the coun- ter is cleared (changes from MAX to BOTTOM).

The PWM frequency for the output can be calculated by the following equation:

$$f_{OCn_{PWM}} = \frac{f_{clk_I/O}}{N * MAX}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024). The extreme values for the OCRx Register represents special cases when generating a PWM waveform output in the fast PWM mode. If the OCRx is set equal to BOTTOM, the output will be a narrow spike for each MAX+1 timer clock cycle. Setting the OCRx equal to MAX will result in a constantly high or low output.

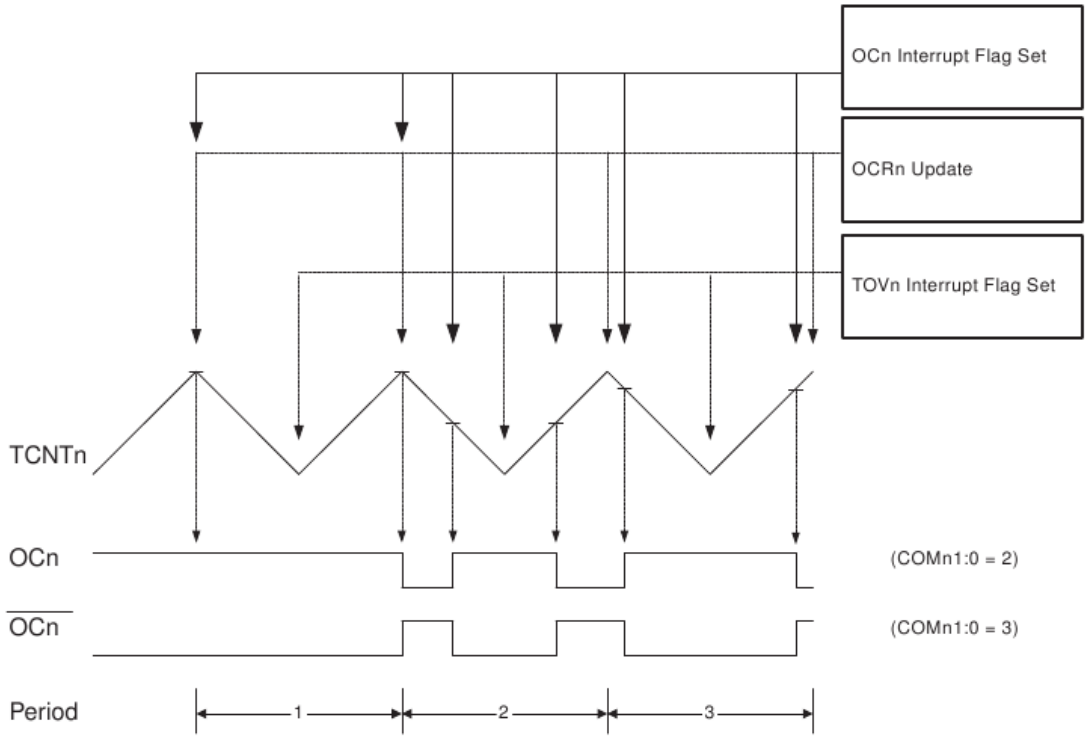
27.2.4 Phase Correct PWM Mode

The phase correct PWM mode provides a high resolution phase correct PWM waveform generation option. The phase correct PWM mode is based on a dual-slope operation.

The counter counts repeatedly from BOTTOM to MAX and then from MAX to BOTTOM. In noninverting Compare Output mode, the Output Compare (OCx) is cleared on the compare match between TCNTx and OCRx while upcounting, and set on the compare match while downcounting. In inverting Output Compare mode, the operation is inverted.

The dual-slope operation has lower maximum operation frequency than single slope operation. However, due to the symmetric feature of the dual-slope PWM modes, these modes are preferred for motor control applications.

In phase correct PWM mode the counter is incremented until the counter value matches MAX. When the counter reaches MAX, it changes the count direction. The TCNTx value will be equal to MAX for one timer clock cycle. The timing diagram for the phase correct PWM mode is shown below. The TCNTx value is in the timing diagram shown as a histogram for illustrating the dual-slope operation. The diagram includes non-inverted and inverted PWM outputs. The small horizontal line marks on the TCNTx slopes represent compare matches between OCRx and TCNTx.



The Timer/Counter Overflow Flag (TOVx) is set each time the counter reaches BOTTOM. The Interrupt Flag can be used to generate an interrupt each time the counter reaches the BOTTOM value.

In phase correct PWM mode, the compare unit allows generation of PWM waveforms on the OCx pin. The actual OCxvalue will only be visible on the port pin if the data direction for the port pin is set as output. The PWM waveform is generated by clearing (or setting) the OCx Register at the compare match between OCRx and TCNTx when the counter increments, and setting (or clearing) the OCx Register at compare match between OCRx and TCNTx when the counter decrements. The PWM frequency for the output when using phase correct PWM can be calculated by the following equation:

$$f_{OCn_{PWM}} = \frac{f_{clk_I/O}}{N * 2 * (MAX - 1)}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024). The extreme values for the OCRx Register represent special cases when generating a PWM waveform output in the phase correct PWM mode. If the OCRx is set equal to BOTTOM, the output will be continuously low and if set equal to MAX the output will be continuously high for noninverted PWM mode. For inverted PWM the output will have the opposite logic values.

At the very start of Period 2 in Figure 33 OCn has a transition from high to low even though there is no Compare Match. The point of this transition is to guarantee symmetry around BOTTOM. There are two cases that give a transition without Compare Match:

- OCRx changes its value from MAX. When the OCRx value is MAX the OCn pin value is the same as the result of a down-counting Compare Match. To ensure symmetry around BOTTOM the OCx value at MAX must be correspond to the result of an up-counting Compare Match.
- The Timer starts counting from a value higher than the one in OCRx, and for that reason misses the Compare Match and hence the OCx change that would have happened on the way up.

27.3 Timer Counter 0 - 8 bit

Timer/Counter0 is a general purpose, single compare unit, 8-bit Timer/Counter module. The main features are:

- Single Compare Unit Counter
- Clear Timer on Compare Match (Auto Reload)
- Glitch-free, Phase Correct Pulse Width Modulator (PWM)
- Frequency Generator
- External Event Counter
- 10-bit Clock Prescaler
- Overflow and Compare Match Interrupt Sources (TOV0 and OCF0)

27.3.1 Timer/Counter Control Register – TCCR0

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – FOC0: Force Output Compare**
  The FOC0 bit is only active when the WGM00 bit specifies a non-PWM mode. However, for ensuring compatibility with future devices, this bit must be set to zero when TCCR0 is written when operating in PWM mode. When writing a logical one to the FOC0 bit, an immediate compare match is forced on the Waveform Generation unit. The OC0 output is changed according to its COM01:0 bits setting. Note that the FOC0 bit is implemented as a strobe. Therefore it is the value present in the COM01:0 bits that determines the effect of the forced compare. A FOC0 strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR0 as TOP. The FOC0 bit is always read as zero.
- **Bit 3, 6 – WGM01:0: Waveform Generation Mode**
  These bits control the counting sequence of the counter, the source for the maximum (TOP) counter value, and what type of Waveform Generation to be used. Modes of operation sup- ported by the Timer/Counter unit are: Normal mode, Clear Timer on Compare Match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes. See table below.

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of Operation	TOP	Update of OCR0	TOV0 Flag Set-on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

Note: 1. The CTC0 and PWM0 bit definition names are now obsolete. Use the WGM01:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

- **Bit 5:4 – COM01:0: Compare Match Output Mode**
  These bits control the Output Compare pin (OC0) behavior. If one or both of the COM01:0 bits are set, the OC0 output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0 pin must be set in order to enable the output driver.

Table 39. Compare Output Mode, non-PWM Mode

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

Table 40. Compare Output Mode, Fast PWM Mode(1)

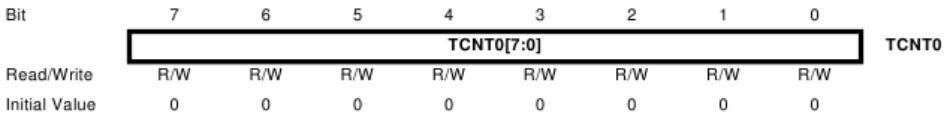
COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Reserved
1	0	Clear OC0 on compare match, set OC0 at BOTTOM, (non-inverting mode)
1	1	Set OC0 on compare match, clear OC0 at BOTTOM, (inverting mode)

Table 41. Compare Output Mode, Phase Correct PWM Mode(1)

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Reserved
1	0	Clear OC0 on compare match when up-counting. Set OC0 on compare match when downcounting.
1	1	Set OC0 on compare match when up-counting. Clear OC0 on compare match when downcounting.

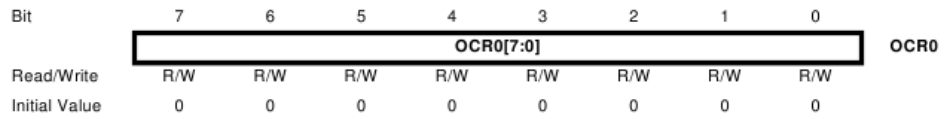
- **Bit 2:0 – CS02:0: Clock Select**
  The three Clock Select bits select the clock source to be used by the Timer/Counter.

Timer/Counter Register – TCNT0



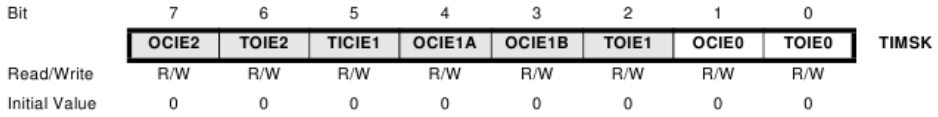
The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT0 Register blocks (removes) the compare match on the following timer clock. Modifying the counter (TCNT0) while the counter is running, introduces a risk of missing a compare match between TCNT0 and the OCR0 Register.

Output Compare Register – OCR0



The Output Compare Register contains an 8-bit value that is continuously compared with the counter value (TCNT0). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0 pin.

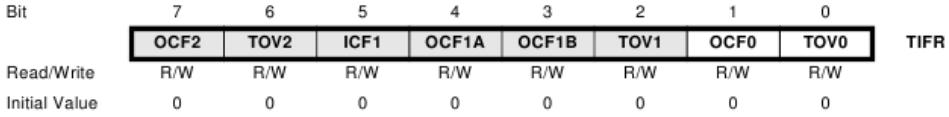
Timer/Counter Interrupt Mask Register – TIMSK



Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCIE0: Timer/Counter0 Output Compare Match Interrupt Enable**
  When the OCIE0 bit is written to one, and the I-bit in the Status Register is set (one), the Timer/Counter0 Compare Match interrupt is enabled. The corresponding interrupt is executed if a compare match in Timer/Counter0 occurs, i.e., when the OCF0 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.
- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**
  When the TOIE0 bit is written to one, and the I-bit in the Status Register is set (one), the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

Timer/Counter Interrupt Flag Register – TIFR



Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCF0: Output Compare Flag 0**
  The OCF0 bit is set (one) when a compare match occurs between the Timer/Counter0 and the data in OCR0 – Output Compare Register0. OCF0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0 is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0 (Timer/Counter0 Compare Match Interrupt Enable), and OCF0 are set (one), the Timer/Counter0 Compare Match Interrupt is executed.
- **Bit 0 – TOV0: Timer/Counter0 Overflow Flag**
  The bit TOV0 is set (one) when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, TOV0 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE0 (Timer/Counter0 Overflow Interrupt Enable), and TOV0 are set (one), the Timer/Counter0 Overflow interrupt is executed. In phase correct PWM mode, this bit is set when Timer/Counter0 changes counting direction at 0x00.

27.4 Timer Counter 1 - 16 bit

The 16-bit Timer/Counter unit allows accurate program execution timing (event management), wave generation, and signal timing measurement. The main features are:

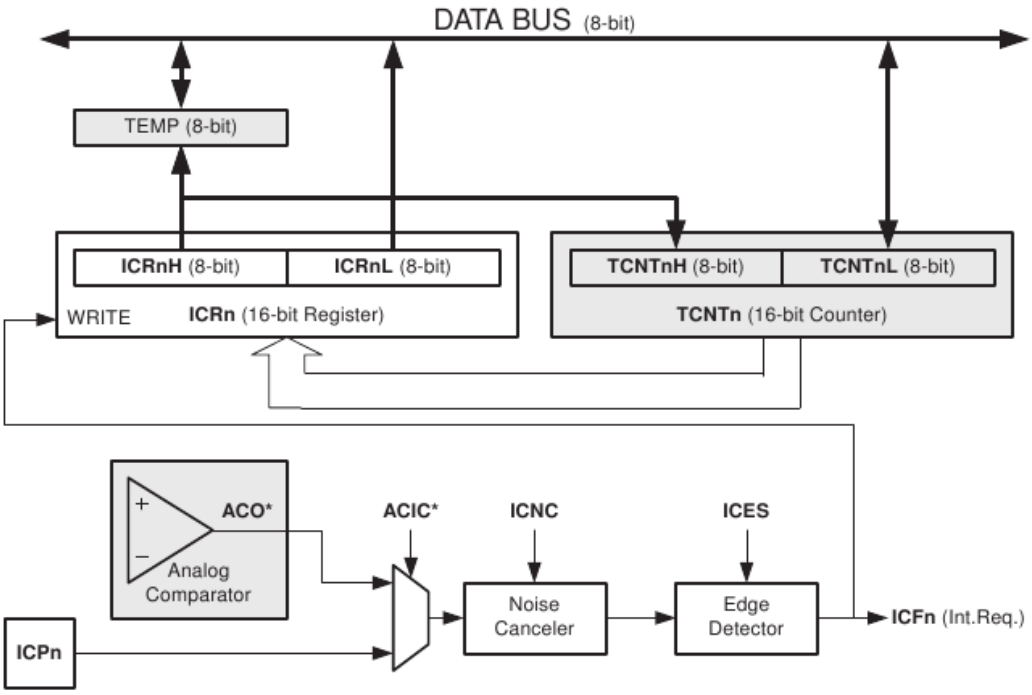
- True 16-bit Design (i.e., Allows 16-bit PWM)
- Two Independent Output Compare Units
- Double Buffered Output Compare Registers
- One Input Capture Unit
- Input Capture Noise Canceler
- Clear Timer on Compare Match (Auto Reload)
- Glitch-free, Phase Correct Pulse Width Modulator (PWM)
- Variable PWM Period
- Frequency Generator

- External Event Counter
- Four Independent Interrupt Sources (TOV1, OCF1A, OCF1B, and ICF1)

27.4.1 Input Capture Unit

The Timer/Counter incorporates an Input Capture unit that can capture external events and give them a time-stamp indicating time of occurrence. The external signal indicating an event, or multiple events, can be applied via the ICP1 pin or alternatively, via the Analog Comparator unit.

The time-stamps can then be used to calculate frequency, duty-cycle, and other features of the signal applied. Alternatively the time-stamps can be used for creating a log of the events. The Input Capture unit is illustrated by the block diagram shown below. The elements of the block diagram that are not directly a part of the Input Capture unit are gray shaded.



When a change of the logic level (an event) occurs on the Input Capture pin (ICP1), alternatively on the Analog Comparator output (ACO), and this change confirms to the setting of the edge detector, a capture will be triggered. When a capture is triggered, the 16-bit value of the counter (TCNT1) is written to the Input Capture Register (ICR1). The Input Capture Flag (ICF1) is set at the same system clock as the TCNT1 value is copied into ICR1 Register. If enabled the Input Capture Flag generates an Input Capture Interrupt. The ICF1 Flag is automatically cleared when the interrupt is executed. Alternatively the ICF1 Flag can be cleared by software by writing a logical one to its I/O bit location.

Reading the 16-bit value in the Input Capture Register (ICR1) is done by first reading the Low byte (ICR1L) and then the High byte (ICR1H). When the Low byte is read the High byte is copied into the High byte temporary register (TEMP). When the CPU reads the ICR1H I/O location it will access the TEMP Register.

The ICR1 Register can only be written when using a Waveform Generation mode that utilizes the ICR1 Register for defining the counter's TOP value. In these cases the Waveform Genera- tion mode (WGM13:0) bits must be set before the TOP value can be written to the ICR1 Register. When writing the ICR1 Register the High byte must be written to the ICR1H I/O loca- tion before the Low byte is written to ICR1L.

27.4.2 Timer/Counter1 Control Register A – TCCR1A

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – COM1A1:0: Compare Output Mode for Channel A**

- **Bit 5:4 – COM1B1:0: Compare Output Mode for Channel B**
  The COM1A1:0 and COM1B1:0 control the Output Compare pins (OC1A and OC1B respectively) behavior. If one or both of the COM1A1:0 bits are written to one, the OC1A output overrides the normal port functionality of the I/O pin it is connected to. If one or both of the COM1B1:0 bit are written to one, the OC1B output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit correspond- ing to the OC1A or OC1B pin must be set in order to enable the output driver. When the OC1A or OC1B is connected to the pin, the function of the COM1x1:0 bits is depen- dent of the WGM13:0 bits setting. The table below shows the COM1x1:0 bit functionality when the WGM13:0 bits are set to a normal or a CTC mode (non-PWM).

Table 44. Compare Output Mode, non-PWM

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	Toggle OC1A/OC1B on compare match
1	0	Clear OC1A/OC1B on compare match (Set output to low level)
1	1	Set OC1A/OC1B on compare match (Set output to high level)

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM13:0 settings, normal port operation, OCnA/OCnB disconnected.
1	0	Clear OC1A/OC1B on compare match, set OC1A/OC1B at BOTTOM, (non-inverting mode)
1	1	Set OC1A/OC1B on compare match, clear OC1A/OC1B at BOTTOM, (inverting mode)

Table 46. Compare Output Mode, Phase Correct and Phase and Frequency Correct PWM

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 9 or 14: Toggle OCnA on Compare Match, OCnB disconnected (normal port operation). For all other WGM13:0 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on compare match when up-counting. Set OC1A/OC1B on compare match when downcounting.
1	1	Set OC1A/OC1B on compare match when up-counting. Clear OC1A/OC1B on compare match when downcounting.

- **Bit 3 – FOC1A: Force Output Compare for Channel A**

- **Bit 2 – FOC1B: Force Output Compare for Channel B**
  The FOC1A/FOC1B bits are only active when the WGM13:0 bits specifies a non-PWM mode. However,

for ensuring compatibility with future devices, these bits must be set to zero when TCCR1A is written when operating in a PWM mode. When writing a logical one to the FOC1A/FOC1B bit, an immediate compare match is forced on the Waveform Generation unit. The OC1A/OC1B output is changed according to its COM1x1:0 bits setting. Note that the FOC1A/FOC1B bits are implemented as strobes. Therefore it is the value present in the COM1x1:0 bits that determine the effect of the forced compare.

A FOC1A/FOC1B strobe will not generate any interrupt nor will it clear the timer in Clear Timer on Compare match (CTC) mode using OCR1A as TOP. The FOC1A/FOC1B bits are always read as zero.

- **Bit 1:0 – WGM11:0: Waveform Generation Mode**
Combined with the WGM13:2 bits found in the TCCR1B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of wave- form generation to be used, see Table 47. Modes of operation supported by the Timer/Counter unit are: Normal mode (counter), Clear Timer on Compare match (CTC) mode, and three types of Pulse Width Modulation (PWM) modes.

27.4.3 Timer/Counter1 Control Register B – TCCR1B

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

- **Bit 7 – ICNC1: Input Capture Noise Canceler**
Setting this bit (to one) activates the Input Capture Noise Canceler. When the Noise Canceler is activated, the input from the Input Capture Pin (ICP1) is filtered. The filter function requires four successive equal valued samples of the ICP1 pin for changing its output. The Input Capture is therefore delayed by four Oscillator cycles when the Noise Canceler is enabled.

- **Bit 6 – ICES1: Input Capture Edge Select**
This bit selects which edge on the Input Capture Pin (ICP1) that is used to trigger a capture event. When the ICES1 bit is written to zero, a falling (negative) edge is used as trigger, and when the ICES1 bit is written to one, a rising (positive) edge will trigger the capture. When a capture is triggered according to the ICES1 setting, the counter value is copied into the Input Capture Register (ICR1). The event will also set the Input Capture Flag (ICF1), and this can be used to cause an Input Capture Interrupt, if this interrupt is enabled. When the ICR1 is used as TOP value (see description of the WGM13:0 bits located in the TCCR1A and the TCCR1B Register), the ICP1 is disconnected and consequently the Input Capture function is disabled.

- **Bit 5 – Reserved Bit**
This bit is reserved for future use. For ensuring compatibility with future devices, this bit must be written to zero when TCCR1B is written.

- **Bit 4:3 – WGM13:2: Waveform Generation Mode**
See TCCR1A Register description.

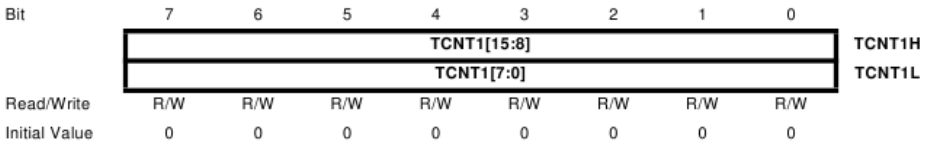
- **Bit 2:0 – CS12:0: Clock Select**
The three Clock Select bits select the clock source to be used by the Timer/Counter, see the figure below.

Table 48. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

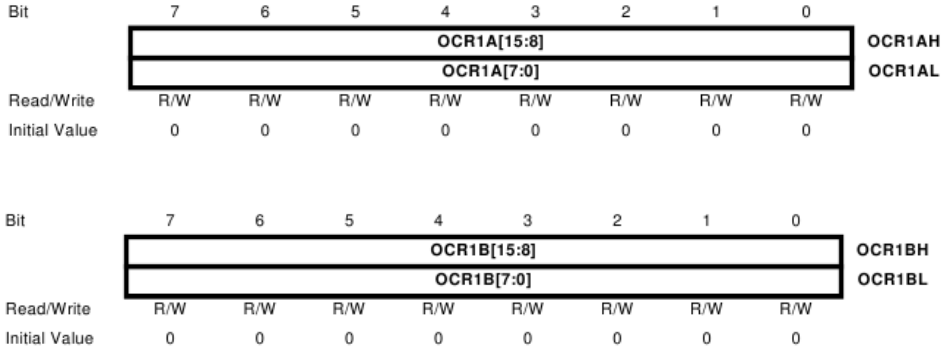
If external pin modes are used for the Timer/Counter1, transitions on the T1 pin will clock the counter even if the pin is configured as an output. This feature allows software control of the counting.

27.4.4 Timer/Counter1 – TCNT1H and TCNT1L



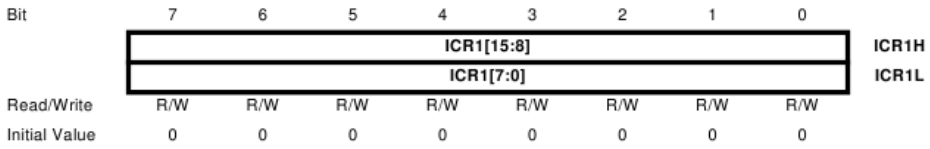
The two Timer/Counter I/O locations (TCNT1H and TCNT1L, combined TCNT1) give direct access, both for read and for write operations, to the Timer/Counter unit 16-bit counter. To ensure that both the high and Low bytes are read and written simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See the section on accessing 16-bit registers. Modifying the counter (TCNT1) while the counter is running introduces a risk of missing a com- pare match between TCNT1 and one of the OCR1x Registers. Writing to the TCNT1 Register blocks (removes) the compare match on the following timer clock for all compare units.

27.4.5 Output Compare Registers



The Output Compare Registers contain a 16-bit value that is continuously compared with the counter value (TCNT1). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC1x pin. The Output Compare Registers are 16-bit in size. To ensure that both the high and Low bytes are written simultaneously when the CPU writes to these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See the “Accessing 16-bit Registers” section

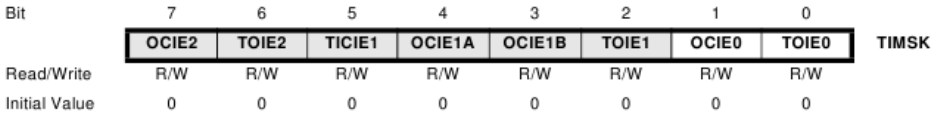
27.4.6 Input Capture Register 1 – ICR1H and ICR1L



The Input Capture is updated with the counter (TCNT1) value each time an event occurs on the ICP1 pin (or optionally on the analog comparator output for Timer/Counter1). The Input Capture can be used for defining the counter TOP value.

The Input Capture Register is 16-bit in size. To ensure that both the high and Low bytes are read simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See the “Accessing 16-bit Registers” section.

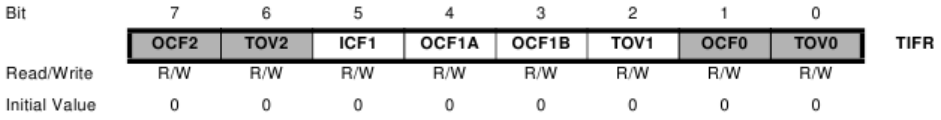
27.4.7 Timer/Counter Interrupt Mask Register – TIMSK



Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 5 – TICIE1: Timer/Counter1, Input Capture Interrupt Enable**
  When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Input Capture Interrupt is enabled. The corresponding Interrupt Vector is executed when the ICF1 Flag, located in TIFR, is set.
- **Bit 4 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**
  When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1A Flag, located in TIFR, is set.
- **Bit 3 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable**
  When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare B match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1B Flag, located in TIFR, is set.
- **Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**
  When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow Interrupt is enabled. The corresponding Interrupt Vector is executed when the TOV1 Flag, located in TIFR, is set.

27.4.8 Timer/Counter Interrupt Flag Register – TIFR



Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 5 – ICF1: Timer/Counter1, Input Capture Flag**
  This flag is set when a capture event occurs on the ICP1 pin. When the Input Capture Register (ICR1) is set by the WGM13:0 to be used as the TOP value, the ICF1 Flag is set when the counter reaches the TOP value. ICF1 is automatically cleared when the Input Capture Interrupt Vector is executed. Alternatively, ICF1 can be cleared by writing a logic one to its bit location.
- **Bit 4 – OCF1A: Timer/Counter1, Output Compare A Match Flag**
  This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register A (OCR1A). Note that a Forced Output Compare (FOC1A) strobe will not set the OCF1A Flag. OCF1A is automatically cleared when the Output Compare Match A Interrupt Vector is executed. Alternatively, OCF1A can be cleared by writing a logic one to its bit location.
- **Bit 3 – OCF1B: Timer/Counter1, Output Compare B Match Flag**
  This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register B (OCR1B). Note that a forced output compare (FOC1B) strobe will not set the OCF1B Flag. OCF1B is automatically cleared when the Output Compare Match B Interrupt Vector is exe- cuted. Alternatively, OCF1B can be cleared by writing a logic one to its bit location.
- **Bit 2 – TOV1: Timer/Counter1, Overflow Flag**
  The setting of this flag is dependent of the WGM13:0 bits setting. In normal and CTC modes, the TOV1 Flag is set when the timer overflows. TOV1 is automatically cleared when the Timer/Counter1 Overflow interrupt vector is executed. Alternatively, TOV1 can be cleared by writing a logic one to its bit location.

27.5 Timer Counter 2 - 8 bit

Timer/Counter2 is a general purpose, single compare unit, 8-bit Timer/Counter module. The main features are:

- Single Compare unit Counter
- Clear Timer on Compare Match (Auto Reload)
- Glitch-free, Phase Correct Pulse Width Modulator (PWM)
- Frequency Generator
- 10-bit Clock Prescaler
- Overflow and Compare Match Interrupt Sources (TOV2 and OCF2)
- Allows clocking from External 32 kHz Watch Crystal Independent of the I/O Clock

27.5.1 Timer/Counter Control Register – TCCR2

Bit	7	6	5	4	3	2	1	0	
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	TCCR2
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – FOC2: Force Output Compare**
  The FOC2 bit is only active when the WGM bits specify a non-PWM mode. However, for ensur- ing compatibility with future devices, this bit must be set to zero when TCCR2 is written when operating in PWM mode. When writing a logical one to the FOC2 bit, an immediate compare match is forced on the waveform generation unit. The OC2 output is changed according to its COM21:0 bits setting. Note that the FOC2 bit is implemented as a strobe. Therefore it is the value present in the COM21:0 bits that determines the effect of the forced compare. A FOC2 strobe will not generate any interrupt, nor will it clear the timer in CTC mode using OCR2 as TOP. The FOC2 bit is always read as zero.
- **Bit 3, 6 – WGM21:0: Waveform Generation Mode**
  These bits control the counting sequence of the counter, the source for the maximum (TOP) counter value, and what type of waveform generation to be used. Modes of operation supported by the Timer/Counter unit are: Normal mode, Clear Timer on Compare match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes.

Table 50. Waveform Generation Mode Bit Description(1)

Mode	WGM21 (CTC2)	WGM20 (PWM2)	Timer/Counter Mode of Operation	TOP	Update of OCR2	TOV2 Flag Set on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR2	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

- **Bit 5:4 – COM21:0: Compare Match Output Mode**
  These bits control the Output Compare pin (OC2) behavior. If one or both of the COM21:0 bits are set, the OC2 output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to OC2 pin must be set in order to enable the output driver.

Table 51. Compare Output Mode, non-PWM Mode

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Toggle OC2 on compare match
1	0	Clear OC2 on compare match
1	1	Set OC2 on compare match

Table 52. Compare Output Mode, Fast PWM Mode[1]

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Reserved
1	0	Clear OC2 on compare match, set OC2 at BOTTOM, (non-inverting mode)
1	1	Set OC2 on compare match, clear OC2 at BOTTOM, (inverting mode)

Table 53. Compare Output Mode, Phase Correct PWM Mode[1]

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected.
0	1	Reserved
1	0	Clear OC2 on compare match when up-counting. Set OC2 on compare match when downcounting.
1	1	Set OC2 on compare match when up-counting. Clear OC2 on compare match when downcounting.

- **Bit 2:0 – CS22:0: Clock Select**
  The three Clock Select bits select the clock source to be used by the Timer/Counter, see table below.

Table 54. Clock Select Bit Description

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk_{T2S}/(No prescaling)
0	1	0	clk_{T2S}/8 (From prescaler)
0	1	1	clk_{T2S}/32 (From prescaler)
1	0	0	clk_{T2S}/64 (From prescaler)
1	0	1	clk_{T2S}/128 (From prescaler)
1	1	0	clk_{T2S}/256 (From prescaler)
1	1	1	clk_{T2S}/1024 (From prescaler)

27.5.2 Timer/Counter Register – TCNT2

Bit	7	6	5	4	3	2	1	0	
	TCNT2[7:0]								TCNT2
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Timer/Counter Register gives direct access, both for read and write operations, to the Timer/Counter unit 8-bit counter. Writing to the TCNT2 Register blocks (removes) the compare match on the following timer clock. Modifying the counter (TCNT2) while the counter is running, introduces a risk of missing a compare match between TCNT2 and the OCR2 Register.

27.5.3 Output Compare Register – OCR2

Bit	7	6	5	4	3	2	1	0	
	OCR2[7:0]								OCR2
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Output Compare Register contains an 8-bit value that is continuously compared with the counter value (TCNT2). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC2 pin.

27.5.4 Asynchronous Status Register – ASSR

Bit	7	6	5	4	3	2	1	0	
	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB	ASSR
Read/Write	R	R	R	R	R/W	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 3 – AS2: Asynchronous Timer/Counter2**
  When AS2 is written to zero, Timer/Counter 2 is clocked from the I/O clock, clkI/O. When AS2 is written to one, Timer/Counter2 is clocked from a Crystal Oscillator connected to the Timer Oscillator 1 (TOSC1) pin. When the value of AS2 is changed, the contents of TCNT2, OCR2, and TCCR2 might be corrupted.
- **Bit 2 – TCN2UB: Timer/Counter2 Update Busy**
  When Timer/Counter2 operates asynchronously and TCNT2 is written, this bit becomes set. When TCNT2 has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCNT2 is ready to be updated with a new value.
- **Bit 1 – OCR2UB: Output Compare Register2 Update Busy**
  When Timer/Counter2 operates asynchronously and OCR2 is written, this bit becomes set. When OCR2 has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that OCR2 is ready to be updated with a new value.
- **Bit 0 – TCR2UB: Timer/Counter Control Register2 Update Busy**
  When Timer/Counter2 operates asynchronously and TCCR2 is written, this bit becomes set. When TCCR2 has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that TCCR2 is ready to be updated with a new value. If a write is performed to any of the three Timer/Counter2 Registers while its update busy flag is set, the updated value might get corrupted and cause an unintentional interrupt to occur. The mechanisms for reading TCNT2, OCR2, and TCCR2 are different. When reading TCNT2, the actual timer value is read. When reading OCR2 or TCCR2, the value in the temporary storage register is read.

Asynchronous Operation of Timer/Counter2

When Timer/Counter2 operates asynchronously, some considerations must be taken.

- Warning: When switching between asynchronous and synchronous clocking of Timer/Counter2, the Timer Registers TCNT2, OCR2, and TCCR2 might be corrupted. A safe procedure for switching clock source is:
  1. Disable the Timer/Counter2 interrupts by clearing OCIE2 and TOIE2.
  2. Select clock source by setting AS2 as appropriate.
  3. Write new values to TCNT2, OCR2, and TCCR2.
  4. To switch to asynchronous operation: Wait for TCN2UB, OCR2UB, and TCR2UB.
  5. Clear the Timer/Counter2 Interrupt Flags.
  6. Enable interrupts, if needed.
- The Oscillator is optimized for use with a 32.768 kHz watch crystal. Applying an external clock to the TOSC1 pin may result in incorrect Timer/Counter2 operation. The CPU main clock frequency must be more than four times the Oscillator frequency.
- When writing to one of the registers TCNT2, OCR2, or TCCR2, the value is transferred to a temporary register, and latched after two positive edges on TOSC1. The user should not write a new value before the contents of the temporary register have been transferred to its destination. Each of the three mentioned registers have their individual temporary register, which means for example that writing to TCNT2 does not disturb an OCR2 write in progress. To detect that a transfer to the destination register has taken place, the Asynchronous Status Register – ASSR has been implemented.

- When entering Power-save or Extended Standby mode after having written to TCNT2, OCR2, or TCCR2, the user must wait until the written register has been updated if Timer/Counter2 is used to wake up the device. Otherwise, the MCU will enter sleep mode before the changes are effective. This is particularly important if the Output Compare2 interrupt is used to wake up the device, since the output compare function is disabled during writing to OCR2 or TCNT2. If the write cycle is not finished, and the MCU enters sleep mode before the OCR2UB bit returns to zero, the device will never receive a compare match interrupt, and the MCU will not wake up.
- If Timer/Counter2 is used to wake the device up from Power-save or Extended Standby mode, precautions must be taken if the user wants to re-enter one of these modes: The interrupt logic needs one TOSC1 cycle to be reset. If the time between wake-up and reentering sleep mode is less than one TOSC1 cycle, the interrupt will not occur, and the device will fail to wake up. If the user is in doubt whether the time before re-entering Powersave or Extended Standby mode is sufficient, the following algorithm can be used to ensure that one TOSC1 cycle has elapsed:
  1. Write a value to TCCR2, TCNT2, or OCR2.
  2. Wait until the corresponding Update Busy Flag in ASSR returns to zero.
  3. Enter Power-save or Extended Standby mode.
- Description of wake up from Power-save or Extended Standby mode when the timer is clocked asynchronously: When the interrupt condition is met, the wake up process is started on the following cycle of the timer clock, that is, the timer is always advanced by at least one before the processor can read the counter value. After wake-up, the MCU is halted for four cycles, it executes the interrupt routine, and resumes execution from the instruction following SLEEP.
- When the asynchronous operation is selected, the 32.768 kHz Oscillator for Timer/Counter2 is always running, except in Power-down and Standby modes. After a Power-up Reset or wake-up from Power-down or Standby mode, the user should be aware of the fact that this Oscillator might take as long as one second to stabilize. The user is advised to wait for at least one second before using Timer/Counter2 after power-up or wake-up from Power-down or Standby mode. The contents of all Timer/Counter2 Registers must be considered lost after a wake-up from Power-down or Standby mode due to unstable clock signal upon startup, no matter whether the Oscillator is in use or a clock signal is applied to the TOSC1 pin.
- Reading of the TCNT2 Register shortly after wake-up from Power-save may give an incorrect result. Since TCNT2 is clocked on the asynchronous TOSC clock, reading TCNT2 must be done through a register synchronized to the internal I/O clock domain. Synchronization takes place for every rising TOSC1 edge. When waking up from Powersave mode, and the I/O clock (clkI/O) again becomes active, TCNT2 will read as the previous value (before entering sleep) until the next rising TOSC1 edge. The phase of the TOSC clock after waking up from Power-save mode is essentially unpredictable, as it depends on the wake-up time. The recommended procedure for reading TCNT2 is thus as follows:
  1. Write any value to either of the registers OCR2 or TCCR2.
  2. Wait for the corresponding Update Busy Flag to be cleared.
  3. Read TCNT2.
- During asynchronous operation, the synchronization of the Interrupt Flags for the asynchronous timer takes three processor cycles plus one timer cycle. The timer is therefore advanced by at least one before the processor can read the timer value causing the setting of the Interrupt Flag. The output compare pin is changed on the timer clock and is not synchronized to the processor clock.

27.5.5 Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – OCIE2: Timer/Counter2 Output Compare Match Interrupt Enable**
  When the OCIE2 bit is written to one and the I-bit in the Status Register is set (one), the Timer/Counter2 Compare Match interrupt is enabled. The corresponding interrupt is executed if a compare match in Timer/Counter2 occurs, i.e., when the OCF2 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

- **Bit 6 – TOIE2: Timer/Counter2 Overflow Interrupt Enable**
  When the TOIE2 bit is written to one and the I-bit in the Status Register is set (one), the Timer/Counter2 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter2 occurs, i.e., when the TOV2 bit is set in the Timer/Counter Interrupt Flag Register – TIFR.

27.5.6 Timer/Counter Interrupt Flag Register – TIFR

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – OCF2: Output Compare Flag 2**
  The OCF2 bit is set (one) when a compare match occurs between the Timer/Counter2 and the data in OCR2 – Output Compare Register2. OCF2 is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF2 is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE2 (Timer/Counter2 Compare match Interrupt Enable), and OCF2 are set (one), the Timer/Counter2 Compare match Interrupt is executed.

- **Bit 6 – TOV2: Timer/Counter2 Overflow Flag**
  The TOV2 bit is set (one) when an overflow occurs in Timer/Counter2. TOV2 is cleared by hard- ware when executing the corresponding interrupt handling vector. Alternatively, TOV2 is cleared by writing a logic one to the flag. When the SREG I-bit, TOIE2 (Timer/Counter2 Overflow Inter- rupt Enable), and TOV2 are set (one), the Timer/Counter2 Overflow interrupt is executed. In PWM mode, this bit is set when Timer/Counter2 changes counting direction at 0x00.

27.5.7 Special Function IO Register – SFIOR

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – PSR2: Prescaler Reset Timer/Counter2**
  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.

CHAPTER

28 Analog to Digital Converter

Features

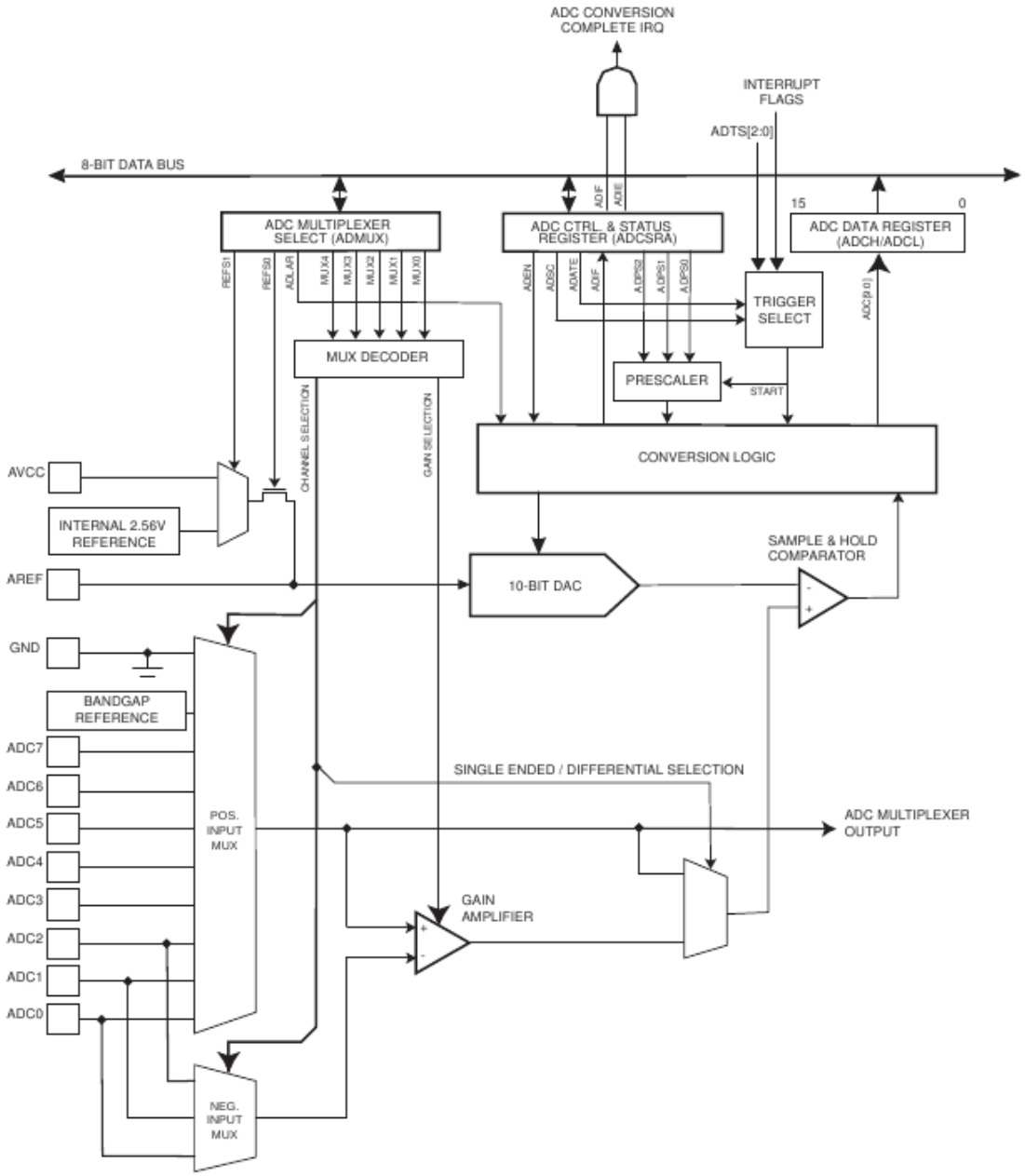
- 10-bit Resolution
- 0.5 LSB Integral Non-linearity
- ±2 LSB Absolute Accuracy
- 13 - 260 µs Conversion Time
- Up to 15 kSPS at Maximum Resolution
- 8 Multiplexed Single Ended Input Channels
- 7 Differential Input Channels
- 2 Differential Input Channels with Optional Gain of 10x and 200x
- Optional Left adjustment for ADC Result Readout
- 0 - VCC ADC Input Voltage Range
- Selectable 2.56V ADC Reference Voltage
- Free Running or Single Conversion Mode
- ADC Start Conversion by Auto Triggering on Interrupt Sources
- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

The ATmega16 features a 10-bit successive approximation ADC. The ADC is connected to an 8-channel Analog Multiplexer which allows 8 single-ended voltage inputs constructed from the pins of Port A. The single-ended voltage inputs refer to 0V (GND).

The device also supports 16 differential voltage input combinations. Two of the differential inputs (ADC1, ADC0 and ADC3, ADC2) are equipped with a programmable gain stage, providing amplification steps of 0 dB (1x), 20 dB (10x), or 46 dB (200x) on the differential input voltage before the A/D conversion. Seven differential analog input channels share a common negative terminal (ADC1), while any other ADC input can be selected as the positive input terminal. If 1x or 10x gain is used, 8-bit resolution can be expected. If 200x gain is used, 7-bit resolution can be expected.

The ADC contains a Sample and Hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion. A block diagram of the ADC is shown below. The ADC has a separate analog supply voltage pin, AVCC. AVCC must not differ more than ±0.3 V from VCC.

Internal reference voltages of nominally 2.56V or AVCC are provided On-chip. The voltage reference may be externally decoupled at the AREF pin by a capacitor for better noise performance.



28.1 Operation

The ADC converts an analog input voltage to a 10-bit digital value through successive approximation. The minimum value represents GND and the maximum value represents the voltage on the AREF pin minus 1 LSB. Optionally, AVCC or an internal 2.56V reference voltage may be connected to the AREF pin by writing to the REFSn bits in the ADMUX Register. The internal voltage reference may thus be decoupled by an external capacitor at the AREF pin to improve noise immunity.

The analog input channel and differential gain are selected by writing to the MUX bits in ADMUX. Any of the ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended inputs to the ADC. A selection of ADC input pins can be selected as positive and negative inputs to the differential gain amplifier.

If differential channels are selected, the differential gain stage amplifies the voltage difference between the selected input channel pair by the selected gain factor. This amplified value then becomes the analog input to the ADC. If single ended channels are used, the gain amplifier is bypassed altogether.

The ADC is enabled by setting the ADC Enable bit, ADEN in ADCSRA. Voltage reference and input channel selections will not go into effect until ADEN is set. The ADC does not consume power when ADEN is cleared, so it is recommended to switch off the ADC before entering power saving sleep modes. The ADC generates a 10-bit result which is presented in the ADC Data Registers, ADCH and ADCL. By default, the result is presented

right adjusted, but can optionally be presented left adjusted by setting the ADLAR bit in ADMUX. If the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH, to ensure that the content of the Data Registers belongs to the same conversion. Once ADCL is read, ADC access to Data Registers is blocked. This means that if ADCL has been read, and a conversion completes before ADCH is read, neither register is updated and the result from the conversion is lost. When ADCH is read, ADC access to the ADCH and ADCL Registers is re-enabled.

The ADC has its own interrupt which can be triggered when a conversion completes. When ADC access to the Data Registers is prohibited between reading of ADCH and ADCL, the interrupt will trigger even if the result is lost.

28.2 Starting a Conversion

A single conversion is started by writing a logical one to the ADC Start Conversion bit, ADSC. This bit stays high as long as the conversion is in progress and will be cleared by hardware when the conversion is completed.

If a different data channel is selected while a conversion is in progress, the ADC will finish the current conversion before performing the channel change.

Alternatively, a conversion can be triggered automatically by various sources. Auto Triggering is enabled by setting the ADC Auto Trigger Enable bit, ADATE in ADCSRA. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in SFIOR (see description of the ADTS bits for a list of the trigger sources). When a positive edge occurs on the selected trigger signal, the ADC prescaler is reset and a conversion is started. This provides a method of starting con- versions at fixed intervals. If the trigger signal is still set when the conversion completes, a new conversion will not be started. If another positive edge occurs on the trigger signal during con- version, the edge will be ignored. Note that an Interrupt Flag will be set even if the specific interrupt is disabled or the global interrupt enable bit in SREG is cleared. A conversion can thus be triggered without causing an interrupt. However, the Interrupt Flag must be cleared in order to trigger a new conversion at the next interrupt event.

Using the ADC Interrupt Flag as a trigger source makes the ADC start a new conversion as soon as the ongoing conversion has finished. The ADC then operates in Free Running mode, constantly sampling and updating the ADC Data Register. The first conversion must be started by writing a logical one to the ADSC bit in ADCSRA. In this mode the ADC will perform successive conversions independently of whether the ADC Interrupt Flag, ADIF is cleared or not.

If Auto Triggering is enabled, single conversions can be started by writing ADSC in ADCSRA to one. ADSC can also be used to determine if a conversion is in progress. The ADSC bit will be read as one during a conversion, independently of how the conversion was started.

28.3 Prescaling and Conversion Timing

By default, the successive approximation circuitry requires an input clock frequency between 50 kHz and 200 kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200 kHz to get a higher sample rate. The ADC module contains a prescaler, which generates an acceptable ADC clock frequency from any CPU frequency above 100 kHz. The prescaling is set by the ADPS bits in ADCSRA. The prescaler starts counting from the moment the ADC is switched on by setting the ADEN bit in ADCSRA. The prescaler keeps running for as long as the ADEN bit is set, and is continuously reset when ADEN is low.

When initiating a single ended conversion by setting the ADSC bit in ADCSRA, the conversion starts at the following rising edge of the ADC clock cycle.

A normal conversion takes 13 ADC clock cycles. The first conversion after the ADC is switched on (ADEN in ADCSRA is set) takes 25 ADC clock cycles in order to initialize the analog circuitry. The actual sample-and-hold takes place 1.5 ADC clock cycles after the start of a normal conversion and 13.5 ADC clock cycles after the start of a first conversion. When a conversion is complete, the result is written to the ADC Data Registers, and ADIF is set. In single conversion mode, ADSC is cleared simultaneously. The software may then set ADSC again, and a new conversion will be initiated on the first rising ADC clock edge.

When Auto Triggering is used, the prescaler is reset when the trigger event occurs. This assures a fixed delay from the trigger event to the start of conversion. In this mode, the sample-and-hold takes place 2 ADC clock cycles after the rising edge on the trigger source signal. Three addi- tional CPU clock cycles are used for

synchronization logic. When using Differential mode, along with Auto triggering from a source other than the ADC Conversion Complete, each conversion

Condition	Sample & Hold (Cycles from Start of Conversion)	Conversion Time (Cycles)
First conversion	13.5	25
Normal conversions, single ended	1.5	13
Auto Triggered conversions	2	13.5
Normal conversions, differential	1.5/2.5	13/14

28.4 Differential Gain Channels

When using differential gain channels, certain aspects of the conversion need to be taken into consideration. Differential conversions are synchronized to the internal clock CK_{ADC2} equal to half the ADC clock. This synchronization is done automatically by the ADC interface in such a way that the sample-and-hold occurs at a specific phase of CK_{ADC2}. A conversion initiated by the user (i.e., all single conversions, and the first free running conversion) when CK_{ADC2} is low will take the same amount of time as a single ended conversion (13 ADC clock cycles from the next prescaled clock cycle). A conversion initiated by the user when CK_{ADC2} is high will take 14 ADC clock cycles due to the synchronization mechanism. In Free Running mode, a new conversion is initi- ated immediately after the previous conversion completes, and since CK_{ADC2} is high at this time, all automatically started (i.e., all but the first) free running conversions will take 14 ADC clock cycles. The gain stage is optimized for a bandwidth of 4 kHz at all gain settings. Higher frequencies may be subjected to non-linear amplification. An external low-pass filter should be used if the input signal contains higher frequency components than the gain stage bandwidth. Note that the ADC clock frequency is independent of the gain stage bandwidth limitation. For example, the ADC clock period may be 6 µs, allowing a channel to be sampled at 12 kSPS, regardless of the band- width of this channel. If differential gain channels are used and conversions are started by Auto Triggering, the ADC must be switched off between conversions. When Auto Triggering is used, the ADC prescaler is reset before the conversion is started. Since the gain stage is dependent of a stable ADC clock prior to the conversion, this conversion will not be valid. By disabling and then re-enabling the ADC between each conversion (writing ADEN in ADCSRA to “0” then to “1”), only extended con- versions are performed. The result from the extended conversions will be valid.

28.5 Changing Channel or Reference Selection

The MUXn and REFS1:0 bits in the ADMUX Register are single buffered through a temporary register to which the CPU has random access. This ensures that the channels and reference selection only takes place at a safe point during the conversion. The channel and reference selection is continuously updated until a conversion is started. Once the conversion starts, the channel and reference selection is locked to ensure a sufficient sampling time for the ADC. Continuous updating resumes in the last ADC clock cycle before the conversion completes (ADIF in ADCSRA is set). Note that the conversion starts on the following rising ADC clock edge after ADSC is written. The user is thus advised not to write new channel or reference selection values to ADMUX until one ADC clock cycle after ADSC is written. If Auto Triggering is used, the exact time of the triggering event can be indeterministic. Special care must be taken when updating the ADMUX Register, in order to control which conversion will be affected by the new settings. If both ADATE and ADEN is written to one, an interrupt event can occur at any time. If the ADMUX Register is changed in this period, the user cannot tell if the next conversion is based on the old or the new settings. ADMUX can be safely updated in the following ways:

1. When ADATE or ADEN is cleared.
2. During conversion, minimum one ADC clock cycle after the trigger event.
3. After a conversion, before the Interrupt Flag used as trigger source is cleared.

When updating ADMUX in one of these conditions, the new settings will affect the next ADC conversion. Special care should be taken when changing differential channels. Once a differential channel has been selected, the gain stage may take as much as 125 µs to stabilize to the new value. Thus conversions should not be started within the first 125 µs after selecting a new differential channel. Alternatively, conversion results obtained within this period should be discarded. The same settling time should be observed for the first differential conversion after changing ADC reference (by changing the REFS1:0 bits in ADMUX).

28.6 ADC Input Channels

When changing channel selections, the user should observe the following guidelines to ensure that the correct channel is selected:

- In Single Conversion mode, always select the channel before starting the conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the conversion to complete before changing the channel selection.
- In Free Running mode, always select the channel before starting the first conversion. The channel selection may be changed one ADC clock cycle after writing one to ADSC. However, the simplest method is to wait for the first conversion to complete, and then change the channel selection. Since the next conversion has already started automatically, the next result will reflect the previous channel selection. Subsequent conversions will reflect the new channel selection.
- When switching to a differential gain channel, the first conversion result may have a poor accuracy due to the required settling time for the automatic offset cancellation circuitry. The user should preferably disregard the first conversion result.

28.7 ADC Voltage Reference

The reference voltage for the ADC (V_{REF}) indicates the conversion range for the ADC. Single ended channels that exceed V_{REF} will result in codes close to 0x3FF. V_{REF} can be selected as either AVCC, internal 2.56V reference, or external AREF pin. AVCC is connected to the ADC through a passive switch. The internal 2.56V reference is gener- ated from the internal bandgap reference (VBG) through an internal amplifier. In either case, the external AREF pin is directly connected to the ADC, and the reference voltage can be made more immune to noise by connecting a capacitor between the AREF pin and ground. VREF can also be measured at the AREF pin with a high impedant voltmeter. Note that V_{REF} is a high impedance source, and only a capacitive load should be connected in a system. If the user has a fixed voltage source connected to the AREF pin, the user may not use the other reference voltage options in the application, as they will be shorted to the external voltage. If no external voltage is applied to the AREF pin, the user may switch between AVCC and 2.56V as reference selection. The first ADC conversion result after switching reference voltage source may be inaccurate, and the user is advised to discard this result. If differential channels are used, the selected reference should not be closer to AVCC than indicated in the Electrical characteristics section of the datasheet.

28.8 ADC Conversion Result

After the conversion is complete (ADIF is high), the conversion result can be found in the ADC Result Registers (ADCL, ADCH). For single ended conversion, the result is

$$ADC = \frac{V_{in} * 1024}{V_{REF}}$$

where VIN is the voltage on the selected input pin and VREF the selected voltage reference. 0x000 represents ground, and 0x3FF represents the selected reference voltage minus one LSB.

ADC Multiplexer Selection Register – ADMUX

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**
  These bits select the voltage reference for the ADC, as shown below. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

- **Bit 5 – ADLAR: ADC Left Adjust Result**
  The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions.
- **Bits 4:0 – MUX4:0: Analog Channel and Gain Selection Bits**
  The value of these bits selects which combination of analog inputs are connected to the ADC. These bits also select the gain for the differential channels. See the table below for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x

ADC Control and Status Register A – ADCSRA

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ADEN: ADC Enable**
  Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**
  In Single Conversion mode, write this bit to one to start each conversion. In Free Running Mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC. ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

- **Bit 5 – ADATE: ADC Auto Trigger Enable**
  When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a con- version on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in SFIOR.

- **Bit 4 – ADIF: ADC Interrupt Flag**
  This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.

- **Bit 3 – ADIE: ADC Interrupt Enable**
  When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.

- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**
  These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

The ADC Data Register – ADCL and ADCH

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

When an ADC conversion is complete, the result is found in these two registers. If differential channels are used, the result is presented in two's complement form. When ADCL is read, the ADC Data Register is not updated until ADCH is read. Consequently, if the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH. The ADLAR bit in ADMUX, and the MUXn bits in ADMUX affect the way the result is read from the registers. If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

28.9 Special FunctionIO Register – SFIOR

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

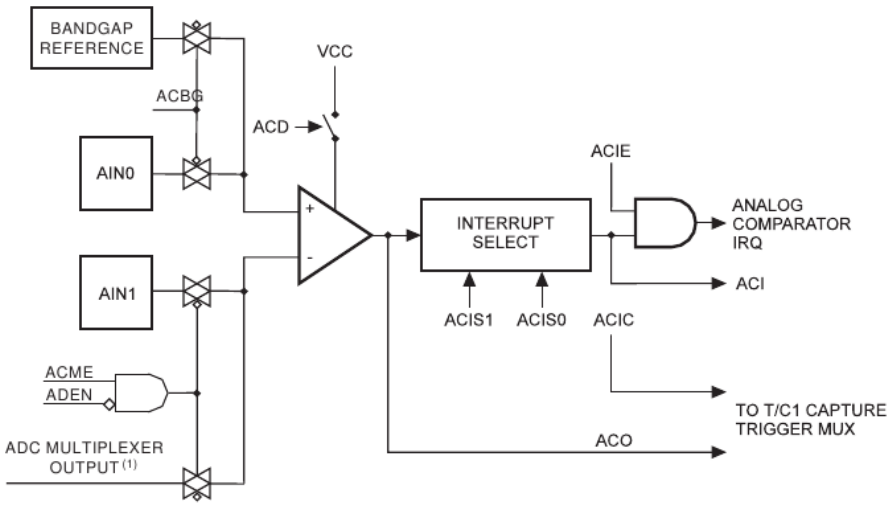
- **Bit 7:5 – ADTS2:0: ADC Auto Trigger Source**
  If ADATE in ADCSRA is written to one, the value of these bits selects which source will trigger an ADC conversion. If ADATE is cleared, the ADTS2:0 settings will have no effect. A conversion will be triggered by the rising edge of the selected Interrupt Flag. Note that switching from a trig ger source that is cleared to a trigger source that is set, will generate a positive edge on the trigger signal. If ADEN in ADCSRA is set, this will start a conversion. Switching to Free Running mode (ADTS[2:0]=0) will not cause a trigger event, even if the ADC Interrupt Flag is set.

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

CHAPTER 29

Analog Comparator

The Analog Comparator compares the input values on the positive pin AIN0 and negative pin AIN1. When the voltage on the positive pin AIN0 is higher than the voltage on the negative pin AIN1, the Analog Comparator Output, ACO, is set. The comparator's output can be set to trigger the Timer/Counter1 Input Capture function. In addition, the comparator can trigger a separate interrupt, exclusive to the Analog Comparator. The user can select Interrupt triggering on comparator output rise, fall or toggle. A block diagram of the comparator and its surrounding logic is shown below.



Special Function IO Register – SFIOR

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 3 – ACME: Analog Comparator Multiplexer Enable**
  When this bit is written logic one and the ADC is switched off (ADEN in ADCSRA is zero), the ADC multiplexer selects the negative input to the Analog Comparator. When this bit is written logic zero, AIN1 is applied to the negative input of the Analog Comparator.

Analog Comparator Control and Status Register – ACSR

Bit	7	6	5	4	3	2	1	0	
	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

- **Bit 7 – ACD: Analog Comparator Disable**
  When this bit is written logic one, the power to the Analog Comparator is switched off. This bit can be set at any time to turn off the Analog Comparator. This will reduce power consumption in active and Idle mode. When changing the ACD bit, the Analog Comparator Interrupt must be disabled by clearing the ACIE bit in ACSR. Otherwise an interrupt can occur when the bit is changed.
- **Bit 6 – ACBG: Analog Comparator Bandgap Select**
  When this bit is set, a fixed bandgap reference voltage replaces the positive input to the Analog Comparator. When this bit is cleared, AIN0 is applied to the positive input of the Analog Compar- ator.
- **Bit 5 – ACO: Analog Comparator Output**
  The output of the Analog Comparator is synchronized and then directly connected to ACO. The synchronization introduces a delay of 1 - 2 clock cycles.
- **Bit 4 – ACI: Analog Comparator Interrupt Flag**
  This bit is set by hardware when a comparator output event triggers the interrupt mode defined by ACIS1 and ACIS0. The Analog Comparator Interrupt routine is executed if the ACIE bit is set and the I-bit in SREG is set. ACI is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ACI is cleared by writing a logic one to the flag.
- **Bit 3 – ACIE: Analog Comparator Interrupt Enable**
  When the ACIE bit is written logic one and the I-bit in the Status Register is set, the Analog Comparator Interrupt is activated. When written logic zero, the interrupt is disabled.
- **Bit 2 – ACIC: Analog Comparator Input Capture Enable**
  When written logic one, this bit enables the Input Capture function in Timer/Counter1 to be triggered by the Analog Comparator. The comparator output is in this case directly connected to the Input Capture front-end logic, making the comparator utilize the noise canceler and edge select features of the Timer/Counter1 Input Capture interrupt. When written logic zero, no connection between the Analog Comparator and the Input Capture function exists. To make the comparator trigger the Timer/Counter1 Input Capture interrupt, the TICIE1 bit in the Timer Interrupt Mask Register (TIMSK) must be set.
- **Bits 1, 0 – ACIS1, ACIS0: Analog Comparator Interrupt Mode Select**
  These bits determine which comparator events that trigger the Analog Comparator interrupt. The different settings are shown below.

ACIS1	ACIS0	Interrupt Mode
0	0	Comparator Interrupt on Output Toggle
0	1	Reserved
1	0	Comparator Interrupt on Falling Output Edge
1	1	Comparator Interrupt on Rising Output Edge

  When changing the ACIS1/ACIS0 bits, the Analog Comparator Interrupt must be disabled by clearing its Interrupt Enable bit in the ACSR Register. Otherwise an interrupt can occur when the bits are changed.

Analog Comparator Multiplexed Input

It is possible to select any of the ADC7..0 pins to replace the negative input to the Analog Com- parator. The ADC multiplexer is used to select this input, and consequently, the ADC must be switched off to utilize this feature. If the Analog Comparator Multiplexer Enable bit (ACME in SFIOR) is set and the ADC is switched off (ADEN in ADCSRA is zero), MUX2..0 in ADMUX select the input pin to replace the negative input to the Analog Comparator, as shown below. If ACME is cleared or ADEN is set, AIN1 is applied to the negative input to the Analog Comparator.

ACME	ADEN	MUX2..0	Analog Comparator Negative Input
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

CHAPTER

30 Watchdog Timer

The Watchdog Timer is clocked from a separate On-chip Oscillator which runs at 1 MHz. This is the typical value at V_{CC} = 5V. See characterization data for typical values at other V_{CC} levels. By controlling the Watchdog Timer prescaler, the Watchdog Reset interval can be adjusted as shown in the table in the register description. The WDR – Watchdog Reset – instruction resets the Watchdog Timer. The Watchdog Timer is also reset when it is disabled and when a Chip Reset occurs. Eight different clock cycle periods can be selected to determine the reset period. If the reset period expires without another Watchdog Reset, the ATmega16 resets and executes from the Reset Vector.

To prevent unintentional disabling of the Watchdog, a special turn-off sequence must be fol- lowed when the Watchdog is disabled. Refer to the description of the Watchdog Timer Control Register for details.

Watchdog Timer Control Register – WDTCR

Bit	7	6	5	4	3	2	1	0	
	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0	WDTCR
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..5 – Res: Reserved Bits**
  These bits are reserved bits in the ATmega16 and will always read as zero.
- **Bit 4 – WDTOE: Watchdog Turn-off Enable**
  This bit must be set when the WDE bit is written to logic zero. Otherwise, the Watchdog will not be disabled. Once written to one, hardware will clear this bit after four clock cycles. Refer to the description of the WDE bit for a Watchdog disable procedure.
- **Bit 3 – WDE: Watchdog Enable**
  When the WDE is written to logic one, the Watchdog Timer is enabled, and if the WDE is written to logic zero, the Watchdog Timer function is disabled. WDE can only be cleared if the WDTOE bit has logic level one. To disable an enabled Watchdog Timer, the following procedure must be followed:
  1. In the same operation, write a logic one to WDTOE and WDE. A logic one must be writ- ten to WDE even though it is set to one before the disable operation starts.
  2. Within the next four clock cycles, write a logic 0 to WDE. This disables the Watchdog.
- **Bits 2..0 – WDP2, WDP1, WDP0: Watchdog Timer Prescaler 2, 1, and 0**
  The WDP2, WDP1, and WDP0 bits determine the Watchdog Timer prescaling when the Watch- dog Timer is enabled. The different prescaling values and their corresponding Timeout Periods are shown in the table below.

WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Typical Time-out at V_{CC} = 3.0V	Typical Time-out at V_{CC} = 5.0V
0	0	0	16K (16,384)	17.1 ms	16.3 ms
0	0	1	32K (32,768)	34.3 ms	32.5 ms
0	1	0	64K (65,536)	68.5 ms	65 ms
0	1	1	128K (131,072)	0.14 s	0.13 s
1	0	0	256K (262,144)	0.27 s	0.26 s
1	0	1	512K (524,288)	0.55 s	0.52 s
1	1	0	1,024K (1,048,576)	1.1 s	1.0 s
1	1	1	2,048K (2,097,152)	2.2 s	2.1 s

The following code example shows one assembly function for turning off the WDT. The example assumes that interrupts are controlled (for example by disabling interrupts globally) so that no interrupts will occur during execution of these functions.

```
WDT_off:
; Reset WDT
WDR
; Write logical one to WDTOE and WDE
in r16, WDTCR
ori r16, (1<<WDTOE)|(1<<WDE)
out WDTCR, r16
; Turn off WDT
ldi r16, (0<<WDE)
out WDTCR, r16
ret
```

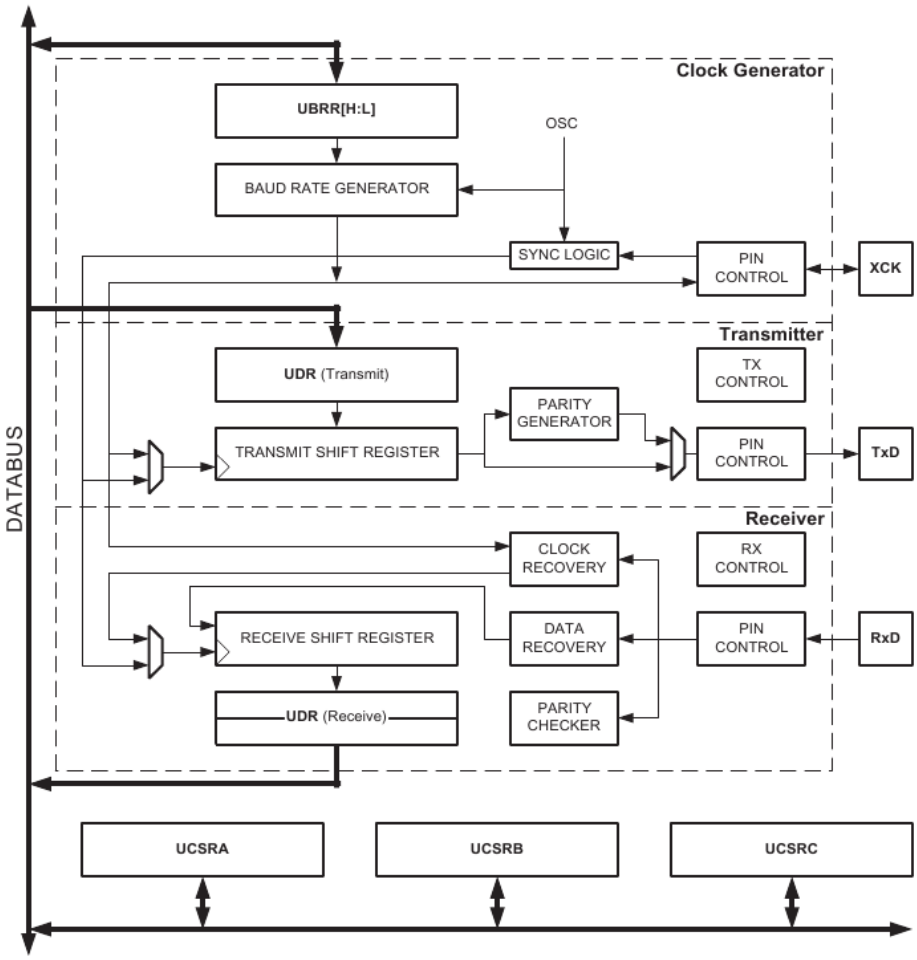
CHAPTER 31

Universal Asyncronous Receiver Transmitter

The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a highly flexible serial communication device. The main features are:

- Full Duplex Operation (Independent Serial Receive and Transmit Registers)
- Asynchronous or Synchronous Operation
- Master or Slave Clocked Synchronous Operation
- High Resolution Baud Rate Generator
- Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits
- Odd or Even Parity Generation and Parity Check Supported by Hardware
- Data OverRun Detection
- Framing Error Detection
- Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter
- Three Separate Interrupts on TX Complete, TX Data Register Empty, and RX Complete
- Multi-processor Communication Mode
- Double Speed Asynchronous Communication Mode

A simplified block diagram of the USART transmitter is shown below. CPU accessible I/O Registers and I/O pins are shown in bold.



The dashed boxes in the block diagram separate the three main parts of the USART (listed from the top): Clock Generator, Transmitter and Receiver. Control Registers are shared by all units. The clock generation logic consists of synchronization logic for external clock input used by synchronous Slave operation, and the baud rate generator. The XCK (Transfer Clock) pin is only used by Synchronous Transfer mode. The Transmitter consists of a single write buffer, a serial Shift Register, parity generator and control logic for handling different serial frame formats. The write buffer allows a continuous transfer of data without any delay between frames. The Receiver is the most complex part of the USART module due to its clock and data recovery units. The recovery units are used for asynchronous data reception. In addition to the recovery units, the receiver includes a parity checker, control logic, a Shift Register and a two level receive buffer (UDR). The receiver supports the same frame formats as the transmitter, and can detect frame error, data overrun and parity errors.

USART Initialization

The USART has to be initialized before any communication can take place. The initialization pro- cess normally consists of setting the baud rate, setting frame format and enabling the Transmitter or the Receiver depending on the usage. For interrupt driven USART operation, the Global Interrupt Flag should be cleared (and interrupts globally disabled) when doing the initialization. Before doing a re-initialization with changed baud rate or frame format, be sure that there are no ongoing transmissions during the period the registers are changed. The TXC Flag can be used to check that the Transmitter has completed all transfers, and the RXC Flag can be used to check that there are no unread data in the receive buffer. Note that the TXC Flag must be cleared before each transmission (before UDR is written) if it is used for this purpose. The following simple USART initialization code example shows one assembly function. The examples assume asynchronous operation using polling (no interrupts enabled) and a fixed frame format. The baud rate is given as a function parameter. For the assembly code, the baud rate parameter is assumed to be stored in the r17:r16 regis- ters. When the function writes to the UCSRC Register, the URSEL bit (MSB) must be set due to the sharing of I/O location by UBRRH and UCSRC.

```
1 USART_Init:
2 ; Set baud rate
3 out UBRRH, r17
```

```
4  out UBRRL, r16
5  ; Enable receiver and transmitter
6  ldi r16, (1<<RXEN)|(1<<TXEN)
7  out UCSRB,r16
8  ; Set frame format: 8data, 2stop bit
9  ldi r16, (1<<URSEL)|(1<<USBS)|(3<<UCSZ0)
10 out UCSRC,r16
11 ret
```

The USART Transmitter is enabled by setting the Transmit Enable (TXEN) bit in the UCSRB Register. When the Transmitter is enabled, the normal port operation of the TxD pin is overridden by the USART and given the function as the transmitter's serial output. The baud rate, mode of operation and frame format must be set up once before doing any transmissions. If synchro- nous operation is used, the clock on the XCK pin will be overridden and used as transmission clock.

Transmitter Flags and Interrupts

The USART transmitter has two flags that indicate its state: USART Data Register Empty (UDRE) and Transmit Complete (TXC). Both flags can be used for generating interrupts. The Data Register Empty (UDRE) Flag indicates whether the transmit buffer is ready to receive new data. This bit is set when the transmit buffer is empty, and cleared when the transmit buffer contains data to be transmitted that has not yet been moved into the Shift Register. For compatibility with future devices, always write this bit to zero when writing the UCSRA Register. When the Data Register empty Interrupt Enable (UDRIE) bit in UCSRB is written to one, the USART Data Register Empty Interrupt will be executed as long as UDRE is set (provided that global interrupts are enabled). UDRE is cleared by writing UDR. When interrupt-driven data transmission is used, the Data Register Empty Interrupt routine must either write new data to UDR in order to clear UDRE or disable the Data Register empty Interrupt, otherwise a new interrupt will occur once the interrupt routine terminates.

The Transmit Complete (TXC) Flag bit is set one when the entire frame in the transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer. The TXC Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXC Flag is useful in half-duplex communication interfaces (like the RS485 standard), where a transmitting application must enter receive mode and free the communication bus immediately after completing the transmission. When the Transmit Compete Interrupt Enable (TXCIE) bit in UCSRB is set, the USART Transmit Complete Interrupt will be executed when the TXC Flag becomes set (provided that global interrupts are enabled). When the transmit complete interrupt is used, the interrupt handling routine does not have to clear the TXC Flag, this is done automatically when the interrupt is executed.

Disabling the Transmitter

The disabling of the transmitter (setting the TXEN to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the transmit Shift Register and transmit Buffer Register do not contain data to be transmitted. When disabled, the transmitter will no lon- ger override the TxD pin.

Data Reception – The USART Receiver

The USART Receiver is enabled by writing the Receive Enable (RXEN) bit in the UCSRB Regis- ter to one. When the receiver is enabled, the normal pin operation of the RxD pin is overridden by the USART and given the function as the receiver's serial input. The baud rate, mode of operation and frame format must be set up once before any serial reception can be done. If synchronous operation is used, the clock on the XCK pin will be used as transfer clock.

Receive Complete Flag and Interrupt

The USART Receiver has one flag that indicates the receiver state. The Receive Complete (RXC) Flag indicates if there are unread data present in the receive buffer. This flag is one when unread data exist in the receive buffer, and zero when the receive buffer is empty (i.e., does not contain any unread data). If the receiver is disabled (RXEN = 0), the receive buffer will be flushed and consequently the RXC bit will become zero. When the Receive Complete Interrupt Enable (RXCIE) in UCSRB is set, the USART Receive Complete Interrupt will be executed as long as the RXC Flag is set (provided that global interrupts are enabled). When interrupt-driven

data reception is used, the receive complete routine must read the received data from UDR in order to clear the RXC Flag, otherwise a new interrupt will occur once the interrupt routine terminates.

Disabling the Receiver

In contrast to the Transmitter, disabling of the Receiver will be immediate. Data from ongoing receptions will therefore be lost. When disabled (i.e., the RXEN is set to zero) the Receiver will no longer override the normal function of the RxD port pin. The receiver buffer FIFO will be flushed when the receiver is disabled. Remaining data in the buffer will be lost

Flushing the Receive Buffer

The receiver buffer FIFO will be flushed when the Receiver is disabled, i.e., the buffer will be emptied of its contents. Unread data will be lost. If the buffer has to be flushed during normal operation, due to for instance an error condition, read the UDR I/O location until the RXC Flag is cleared.

Accessing UBRRH/ UCSRC Registers

When doing a write access of this I/O location, the high bit of the value written, the USART Register Select (URSEL) bit, controls which one of the two registers that will be written. If URSEL is zero during a write operation, the UBRRH value will be updated. If URSEL is one, the UCSRC setting will be updated.

Doing a read access to the UBRRH or the UCSRC Register is a more complex operation. However, in most applications, it is rarely necessary to read any of these registers.

The read access is controlled by a timed sequence. Reading the I/O location once returns the UBRRH Register contents. If the register location was read in previous system clock cycle, read- ing the register in the current clock cycle will return the UCSRC contents. Note that the timed sequence for reading the UCSRC is an atomic operation. Interrupts must therefore be controlled (for example by disabling interrupts globally) during the read operation.

USART I/O Data Register – UDR

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR (Read)
	TXB[7:0]								UDR (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR. The Transmit Data Buffer Reg- ister (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB). For 5-, 6-, or 7-bit characters the upper unused bits will be ignored by the Transmitter and set to zero by the Receiver.

The transmit buffer can only be written when the UDRE Flag in the UCSRA Register is set. Data written to UDR when the UDRE Flag is not set, will be ignored by the USART Transmitter. When data is written to the transmit buffer, and the Transmitter is enabled, the Transmitter will load the data into the transmit Shift Register when the Shift Register is empty. Then the data will be serially transmitted on the TxD pin.

The receive buffer consists of a two level FIFO. The FIFO will change its state whenever the receive buffer is accessed. Due to this behavior of the receive buffer, do not use read modify write instructions (SBI and CBI) on this location. Be careful when using bit test instructions (SBIC and SBIS), since these also will change the state of the FIFO.

USART Control and Status Register A – UCSRA

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- **Bit 7 – RXC: USART Receive Complete**
  This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the receiver is disabled, the receive buffer will be flushed and consequently the RXC bit will become zero. The RXC Flag can be used to generate a Receive Complete interrupt (see description of the RXCIE bit).

- **Bit 6 – TXC: USART Transmit Complete**
  This flag bit is set when the entire frame in the transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDR). The TXC Flag bit is auto- matically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXC Flag can generate a Transmit Complete interrupt (see description of the TXCIE bit).

- **Bit 5 – UDRE: USART Data Register Empty**
  The UDRE Flag indicates if the transmit buffer (UDR) is ready to receive new data. If UDRE is one, the buffer is empty, and therefore ready to be written. The UDRE Flag can generate a Data Register empty Interrupt (see description of the UDRIE bit). UDRE is set after a reset to indicate that the transmitter is ready.

- **Bit 4 – FE: Frame Error**
  This bit is set if the next character in the receive buffer had a Frame Error when received. i.e., when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDR) is read. The FE bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRA.

- **Bit 3 – DOR: Data OverRun**
  This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the receive Shift Register, and a new start bit is detected. This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- **Bit 2 – PE: Parity Error**
  This bit is set if the next character in the receive buffer had a Parity Error when received and the parity checking was enabled at that point (UPM1 = 1). This bit is valid until the receive buffer (UDR) is read. Always set this bit to zero when writing to UCSRA.

- **Bit 1 – U2X: Double the USART Transmission Speed**
  This bit only has effect for the asynchronous operation. Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

- **Bit 0 – MPCM: Multi-processor Communication Mode**
  This bit enables the Multi-processor Communication mode. When the MPCM bit is written to one, all the incoming frames received by the USART receiver that do not contain address infor- mation will be ignored. The transmitter is unaffected by the MPCM setting.

USART Control and Status Register B – UCSRB

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – RXCIE: RX Complete Interrupt Enable**
  Writing this bit to one enables interrupt on the RXC Flag. A USART Receive Complete Interrupt will be generated only if the RXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the RXC bit in UCSRA is set.

- **Bit 6 – TXCIE: TX Complete Interrupt Enable**
  Writing this bit to one enables interrupt on the TXC Flag. A USART Transmit Complete Interrupt will be generated only if the TXCIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the TXC bit in UCSRA is set.

- **Bit 5 – UDRIE: USART Data Register Empty Interrupt Enable**
  Writing this bit to one enables interrupt on the UDRE Flag. A Data Register Empty Interrupt will be generated only if the UDRIE bit is written to one, the Global Interrupt Flag in SREG is written to one and the UDRE bit in UCSRA is set.

- **Bit 4 – RXEN: Receiver Enable**
  Writing this bit to one enables the USART Receiver. The Receiver will override normal port oper- ation for the RxD pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the FE, DOR, and PE Flags.

- **Bit 3 – TXEN: Transmitter Enable**
  Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the TxD pin when enabled. The disabling of the Transmitter (writing TXEN to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the transmit Shift Register and transmit Buffer Register do not contain data to be transmitted. When disabled, the transmitter will no longer override the TxD port.

- **Bit 2 – UCSZ2: Character Size**
  The UCSZ2 bits combined with the UCSZ1:0 bit in UCSRC sets the number of data bits (Character Size) in a frame the receiver and transmitter use.

- **Bit 1 – RXB8: Receive Data Bit 8**
  RXB8 is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDR.

- **Bit 0 – TXB8: Transmit Data Bit 8**
  TXB8 is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDR.

USART Control and Status Register C – UCSRC

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

- **Bit 7 – URSEL: Register Select**
  This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.

- **Bit 6 – UMSEL: USART Mode Select**
  This bit selects between Asynchronous and Synchronous mode of operation.

UMSEL	Mode
0	Asynchronous Operation
1	Synchronous Operation

- **Bit 5:4 – UPM1:0: Parity Mode**
  These bits enable and set type of parity generation and check. If enabled, the transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM0 setting. If a mismatch is detected, the PE Flag in UCSRA will be set.

UPM1	UPM0	Parity Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

- **Bit 3 – USBS: Stop Bit Select**
  This bit selects the number of Stop Bits to be inserted by the Transmitter. The Receiver ignores this setting.

USBS	Stop Bit(s)
0	1-bit
1	2-bit

- **Bit 2:1 – UCSZ1:0: Character Size**
  The UCSZ1:0 bits combined with the UCSZ2 bit in UCSRB sets the number of data bits (Char- acter Size) in a frame the Receiver and Transmitter use.

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

- **Bit 0 – UCPOL: Clock Polarity**
  This bit is used for Synchronous mode only. Write this bit to zero when Asynchronous mode is used. The UCPOL bit sets the relationship between data output change and data input sample, and the synchronous clock (XCK).

USART Baud Rate Registers – UBRRL and UBRRH

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

- **Bit 15 – URSEL: Register Select**
  This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH.

- **Bit 14:12 – Reserved Bits**
  These bits are reserved for future use. For compatibility with future devices, these bit must be written to zero when UBRRH is written.

- **Bit 11:0 – UBRR11:0: USART Baud Rate Register**
  This is a 12-bit register which contains the USART baud rate. The UBRRH contains the four most significant bits, and the UBRRL contains the 8 least significant bits of the USART baud rate. Ongoing transmissions by the transmitter and receiver will be corrupted if the baud rate is changed. Writing UBRRL will trigger an immediate update of the baud rate prescaler.

31.0.1 Examples of Baud Rate Settings

For standard crystal and resonator frequencies, the most commonly used baud rates for asynchronous operation can be generated by using the UBRR settings in the table below. UBRR values which yield an actual baud rate differing less than 0.5% from the target baud rate, are bold in the table. Higher error ratings are acceptable, but the receiver will have less noise resistance when the error ratings are high, especially for large serial frames.

Baud Rate (bps)	f_{osc} = 8.0000 MHz			
	U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error
2400	207	0.2%	416	-0.1%
4800	103	0.2%	207	0.2%
9600	51	0.2%	103	0.2%
14.4k	34	-0.8%	68	0.6%
19.2k	25	0.2%	51	0.2%
28.8k	16	2.1%	34	-0.8%
38.4k	12	0.2%	25	0.2%
57.6k	8	-3.5%	16	2.1%
76.8k	6	-7.0%	12	0.2%
115.2k	3	8.5%	8	-3.5%
230.4k	1	8.5%	3	8.5%
250k	1	0.0%	3	0.0%
0.5M	0	0.0%	1	0.0%
1M	–	–	0	0.0%
Max (1)	0.5 Mbps		1 Mbps	

CHAPTER

32 EEPROM

See the relevant section in the Memory Chapter.

Part IV MCU Tricks and Tips

CHAPTER

33 Storing Data in Non-Volatile Memory

Example 1 Storing and Retrieving Data in EEPROM

```
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST
.def TMP1=R16
.def MESSAGE_offset=r19
.eseg
.org 0x000
EEPMESSAGE:     .db "Hello",0x00
.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 $02A       ;locate code past the interupt vectors
START:     \section*{Example 1 Storing and Retrieving Data in EEPROM}
\begin{lstlisting}
      ldi TMP1, LOW(RAMEND)     ;initialise the stack pointer
      out SPL, TMP1
      ldi TMP1, HIGH(RAMEND)
      out SPH, TMP1
      call READ_MESSAGES_FROM_EEPROM
MAIN_LOOP:
            NOP
      NOP
      RJMP MAIN_LOOP

READ_MESSAGES_FROM_EEPROM:
      LDI XL, 0x00
      LDI XH, 0x00
      ;read in message
      LDI YL, low(MESSAGE)
      LDI YH,     high(MESSAGE)
      call Read_Bytes
      RET

;* reads bytes in from EEPROM starting at address  contained in X and stores them in SRAM in the
; location pointed to by Y, until a null is received.
READ_BYTES:
      SBIC EECR, EEWE     ; wait to make sure there is no active write
      RJMP READ_BYTES
      out EEARH, XH
      OUT EEARL, XL
      SBI EECR, EERE
      in tmp2, EEDR
      st y+, tmp2
      adiw xl, 1
      cpi tmp2, 0x00 ; test for null
      BRNE READ_BYTES
      ret
```

Example 2 Storing and Retrieving Data in EEPROM

```
 .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST
.def TMP1=R16        ;
.def TMP2=R17        ;
.def TMP3=R18
.eseg
.org 0x000
MESSAGES: .db "Message 1",0x00,"Message2",0x00

.dseg
Message1: .BYTE 16 ;reserve 16 bytes of RAM for this message
Message2:    .BYTE 16 ;reserve 16 bytes of RAM for this message

.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 $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 READ_MESSAGE1_FROM_EEPROM
MAIN_LOOP:
      NOP
      NOP
      NOP
      RJMP MAIN_LOOP

;Reads in the 1rst message, does not check for array overflows.
;NB the simulator does not read in the .eep file generated by
;the compiler, so you have to manually enter data into the EEPROM
;in the simulator.
READ_MESSAGE1_FROM_EEPROM:
;Set up address registers.
      LDI XL, 0x00
      LDI XH, 0x00
      LDI YL, low(Message1)
      LDI YH,     high(message1)

READ_BYTE:
      SBIC EECR, EEWE     ; wait to make sure there is no active write
      RJMP READ_BYTE
      out EEARH, XH
      OUT EEARL, XL
      SBI EECR, EERE
      in tmp2, EEDR
      st y+, tmp2
      adiw xl, 1
      cpi tmp2, 0x00 ; test for null
      BRNE READ_BYTE
      ret
```

Example 1 Storing and Retrieving Data in Program Memory

```
  .INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
;.DEVICE ATmega16 -- done in m16def.inc
.NOLIST
.INCLUDE "m16def.inc"
.LIST
.def TMP1=R16        ;
.def TMP2=R17        ;
.def TMP3=R18
.dseg
SRAM_VAR_1:     .BYTE 4 ;reserve 4 bytes of RAM for this variable
.cseg                ;Tell the assembler that everything below this is in the code segment
.org 0xf00
constants: .DB 0xDE, 0x73, 0b01010101
```

```
.org 0x000          ;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 READ_IN_CONSTANTS

MAIN_LOOP:
      NOP
      NOP
      NOP
      RJMP MAIN_LOOP

READ_IN_CONSTANTS:
      ;Setup pointer to .cseg locations
      LDI ZL, LOW(2*constants) ;low address byte of the array constants
      ;(the 2* is used because the program memory is orgainised in 16-bit words)
      LDI ZH, high(2*constants) ; high address byte of the array constants
      ;Setup pointer to .desg locations
      LDI YL, low(SRAM_VAR_1) ;low address byte of the array SRAM_VAR_1
      LDI YH, high(SRAM_VAR_1) ; high address byte of the array SRAM_VAR_1

      ;transfer first byte
      LPM TMP1, Z+
      ST Y+, TMP1
      ;transfer 2nd byte
      LPM TMP1, Z+
      ST Y+, TMP1
      ;transfer 3rd byte
      LPM TMP1, Z+
      ST Y+, TMP1
      RET
```

Example 1 Storing a Lookup Table in EEPROM

```
.INCLUDEPATH "/usr/share/avra" ;set the inlude path to the correct place
.INCLUDE "m16def.inc"
.def ZERO=R19
.def TMP1=R16       ;
.def TMP2=R17       ;
.def TMP3=R18
.eseg
.org 0x000
EE_LOOKUP:      .db 0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80
.dseg
RAM_LOOKUP:     .BYTE 10 ;reserve 16 bytes of RAM for lookuptable
.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 $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 READ_LOOKUP_FROM_EEPROM
     CLR ZERO

     LDI TMP1, 0x00
     OUT PORTB, TMP1; all leds initially off.
     LDI TMP1, 0xff
     OUT DDRB, TMP1 ; make portb outputs

     ; now lookup a value in the4 lookup table
     ldi TMP1, 5;  ; we will use a value of 5, to light up the 5th led
     ;set X to point to the beginning of the lookuptable
      LDI XL, low(RAM_LOOKUP)
      LDI XH,     high(RAM_LOOKUP)
      ; add out value to the address
      ADD XL, TMP1
      ADC XH, ZERO
```

```
37        ; X now contains the address of the 5th element, i.e. 0x10
38        LD TMP2, X
39        ; tmp2 now contains the value of the 5th element
40        OUT PORTB, TMP2
41  MAIN_LOOP:
42        NOP
43        NOP
44        NOP
45        RJMP MAIN_LOOP
46
47  READ_LOOKUP_FROM_EEPROM:
48  ;Set up address registers.
49        LDI XL, low(EE_LOOKUP)
50        LDI XH, high(EE_LOOKUP)
51        LDI YL, low(RAM_LOOKUP)
52        LDI YH,     high(RAM_LOOKUP)
53       LDI TMP1, 0x00
54  READ_BYTE:
55        SBIC EECR, EEWE      ; wait to make sure there is no active write
56        RJMP READ_BYTE
57        out EEARH, XH
58        OUT EEARL, XL
59        SBI EECR, EERE
60        in tmp2, EEDR
61        st y+, tmp2
62        adiw xl, 1
63        inc TMP1
64        CPI TMP1, 8
65        BRNE READ_BYTE
66        ret
```

CHAPTER 34

Lookup Tables

Applications for lookup tables

Lookup tables can be used whenever there is a need for associating a particular value with an "index". This can be used when there is no logical or mathematical way of determining the value or when doing so would take too long.

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.

Choosing a location for lookup tables

The location of a lookup table depends on the availability of resuorces and the required speed of operation.

EEPROM

EEPROM provides 512 bytes of space that will not be required by code, it is however rather slow.

Program Memory

Storing lookup tables in the flash based program memory provides the largest storage location (provided it is not needed by code) and is faster then EEPROM, but still slower than SRAM.

SRAM

Lookup tables in SRAM are the fastest, but they can use up the valuable resource rather quickly, given that there is only 1kB of SRAM. It is also volatile, which means that the lookup table would have to be stored in a non-volatile memory location and then transferred to SRAM on startup.

CHAPTER 35

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.

```
 .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 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

; 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.

CHAPTER

36 Acessing 16-bit Registers

Some of the registers in the ATMega16 are treated as 16-bit registers and accessing them requires some care.

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

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

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

The following code examples show how to access the 16-bit Timer Registers assuming that no interrupts updates the temporary register. The same principle can be used directly for accessing othe 16-bit registers.

```
1  ; Set TCNT1 to 0x01FF
2 ldi r17,0x01
3 ldi r16,0xFF
4 out TCNT1H,r17
5 out TCNT1L,r16
6
7 ; Read TCNT1 into r17:r16
8 in r16,TCNT1L
9 in r17,TCNT1H
```

CHAPTER 37

Bitmasks

Using bitmasks is a staple of embedded IO, sometimes only some of the bits in a register need to be altered (and in the case of a PORTx value need to change at the same time). The example below shows how to do this.

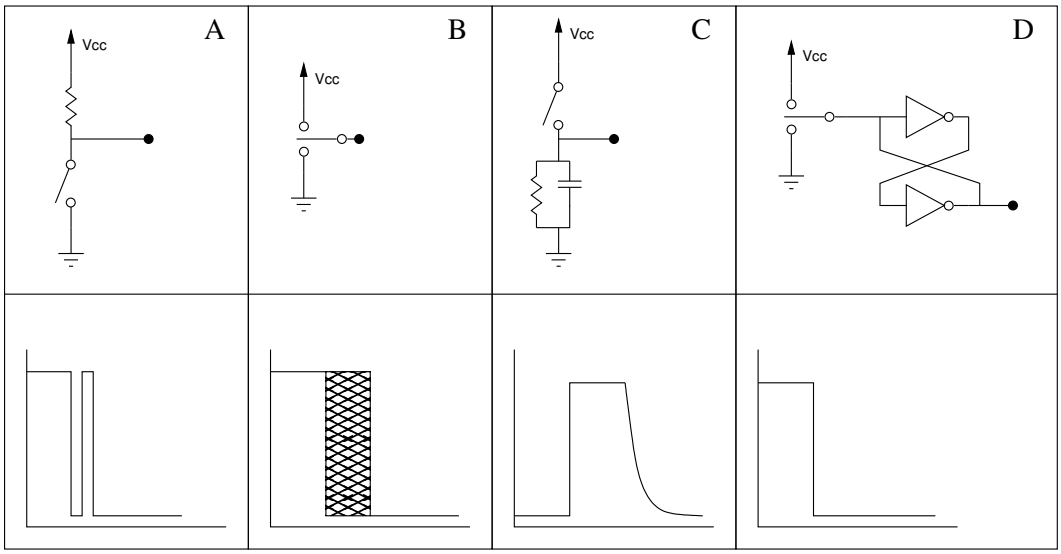
```
1  ; We want to set the 4 lower bits of portb to the following values 0b0110
2  ; without altering the high bits in any way.
3  in tmp2, PORTB ; read in the current values
4  andi tmp2, 0b11110000 ; the lower bits are zeroed, leaving the high bits unaltered
5  ori tmp2, 0b00000110; bits 1 & 2 are set to one
6  out PORTB, tmp2
```

In C this would look like: PORTB=(PORTB&0b11110000)|0b00000110.

CHAPTER 38

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.

Part V Using the Toolchain

CHAPTER 39

Installing the Virtual Machine

Windows

- Install VirtualBox
- Copy the Virtual Drive interfacing1_disk.vdi to your hard drive.
- Create a new Virtual machine and when asked for the disk, point VirtualBox to the provided .vdi
- Start the Virtual machine.
- Login to the virtual machine with the credentials provided.
- Choose the "Insert Guest Addions CDROM" from the window manager.
- Virtual Machine ready to go.

Linux

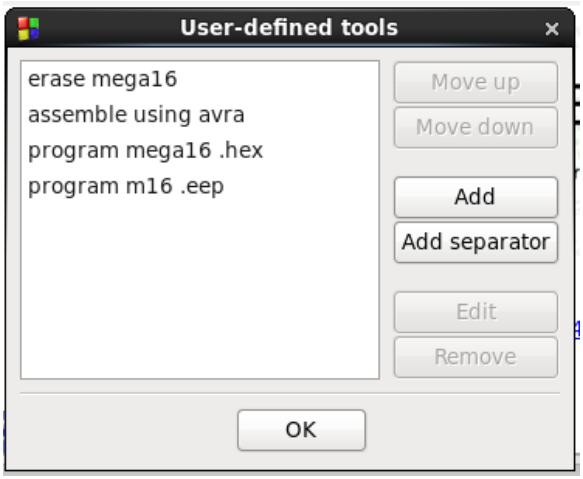
- Install VirtualBox (yum install VirtualBox)
- Copy the Virtual Drive interfacing1_disk.vdi to your hard drive.
- Create a new Virtual machine and when asked for the disk, point VirtualBox to the provided .vdi
- Start the Virtual machine.
- Login to the virtual machine with the credentials provided.
- Choose the "Insert Guest Addions CDROM" from the window manager.
- Virtual Machine ready to go.

CHAPTER

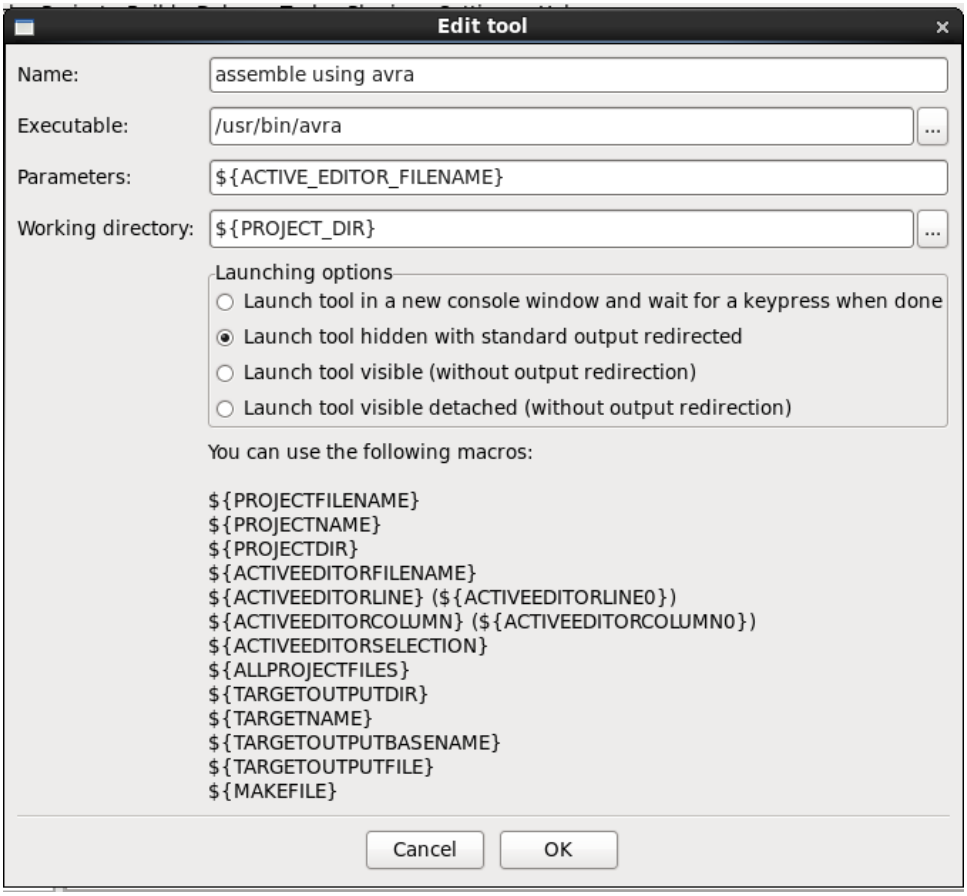
40 Installing under a native environment

40.1 Linux

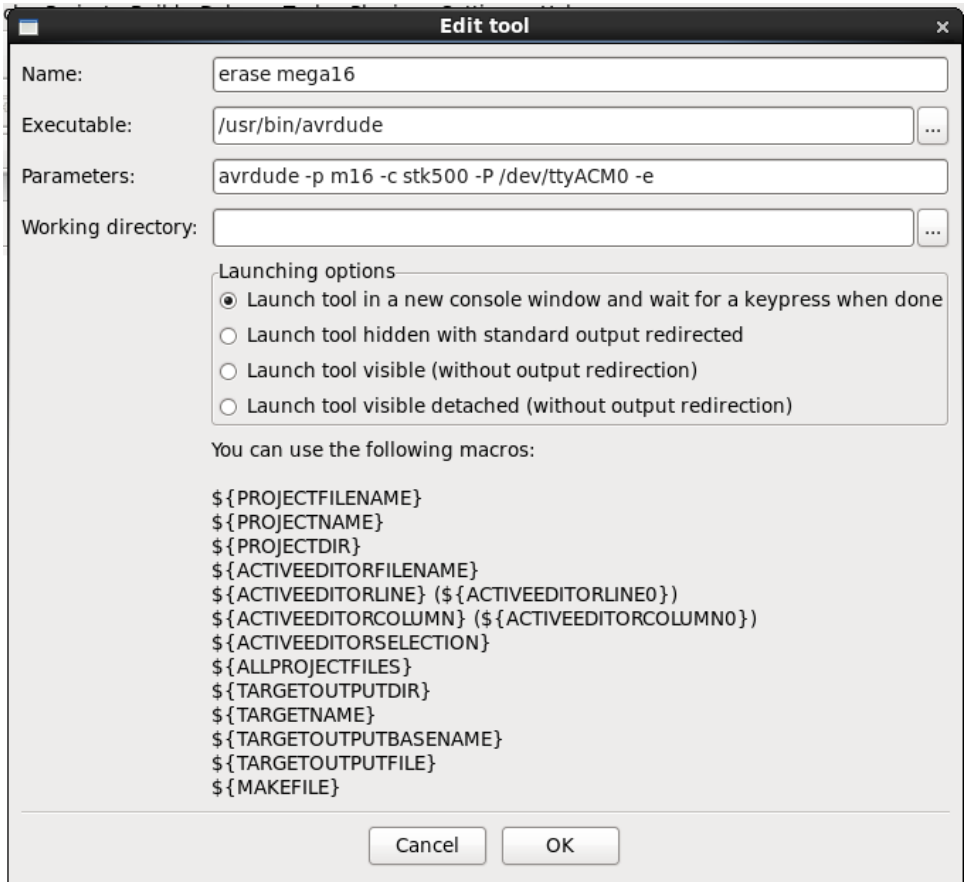
- Install the assembler, AVRA.
- Install CodeBlocks
- Install AVRDude
- Configure CodeBlocks
  - Creating a new 'tool' from the tools menu.



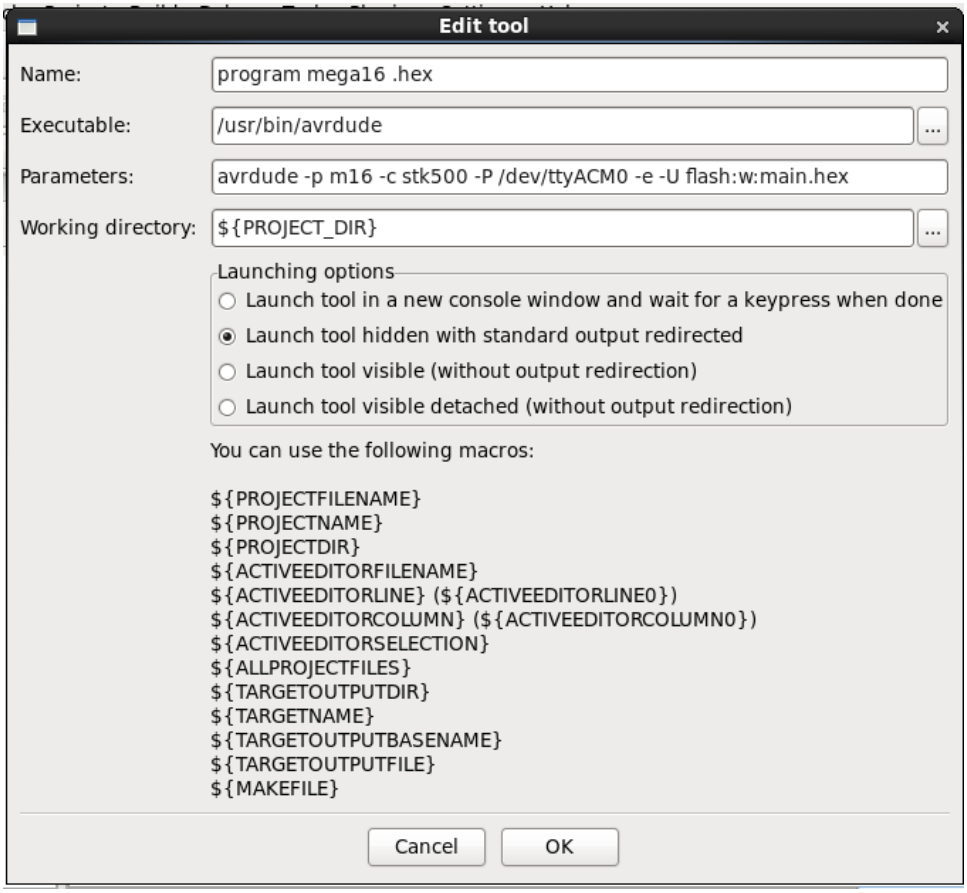
  - Add the 'avr-assemble' tool



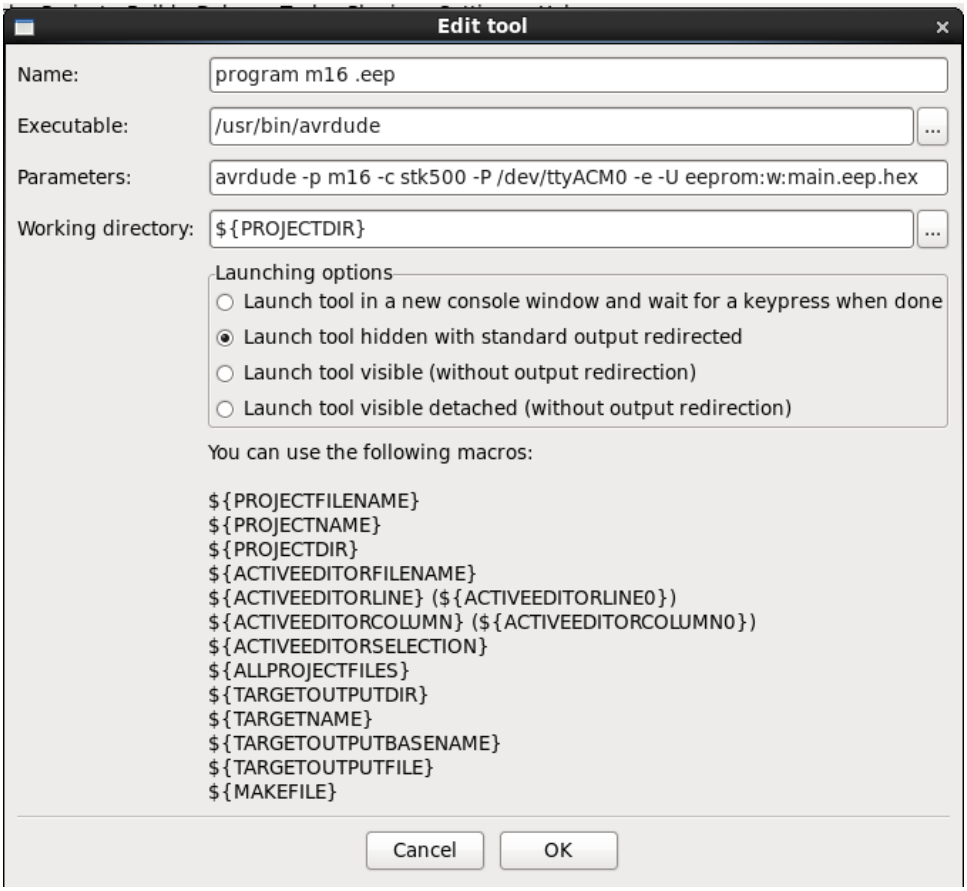
– Add the erase tool



– Add the program flash tool



– Add the program EEPROM tool



40.2 Windows

This is not supported (or recommended in this course), but Windows equivalents of AVRA, AVRDude and CodeBlocks are available.

CHAPTER

41 Using the Toolchain

41.1 Setting up a project in Codeblocks

When starting a new project in CodeBlocks remember to use the 'No Compiler' option.

The main file of any project should be called main.asm, this is not a restriction on name for any reason other than that the 'assemble' tool will assemble the file called main.asm.

41.2 Using the AVRA assembler

The assembler tool (if set up correctly, which it is in the Virtual Machine) will assemble correctly.

41.3 Using AVRDude

AVRDude will program correctly provided the USB programmer is plugged in and switched through to the virtual machine.

41.4 Viewing .hex and .eep files

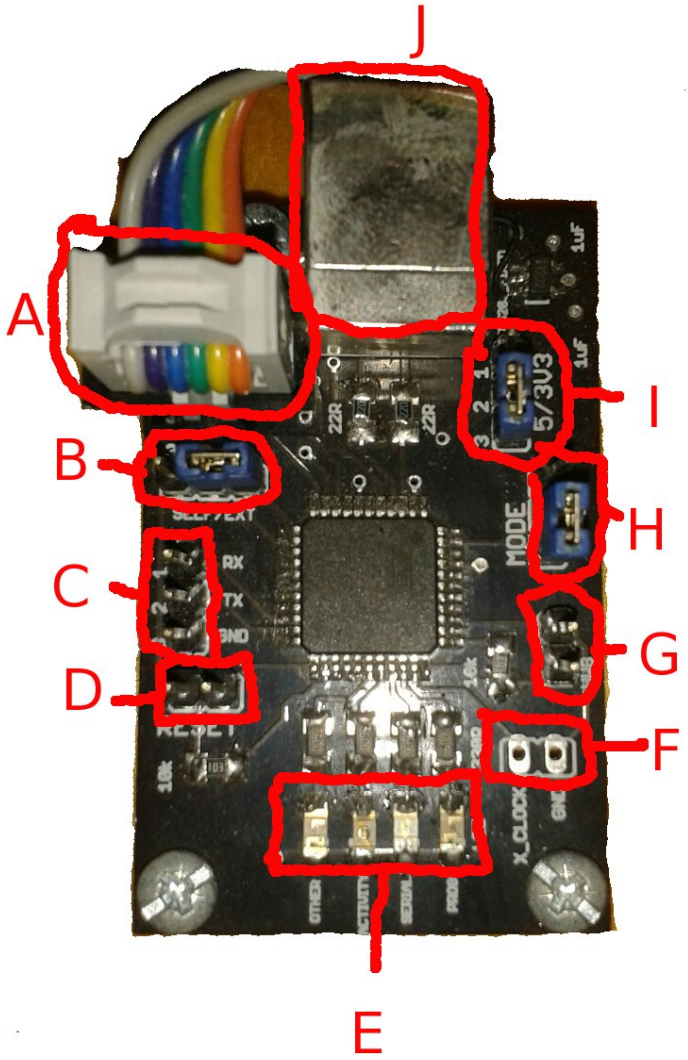
Not currently Supported.

Part VI The Mega16 Devboard

CHAPTER 42

USB Programmer & Serial Converter

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.

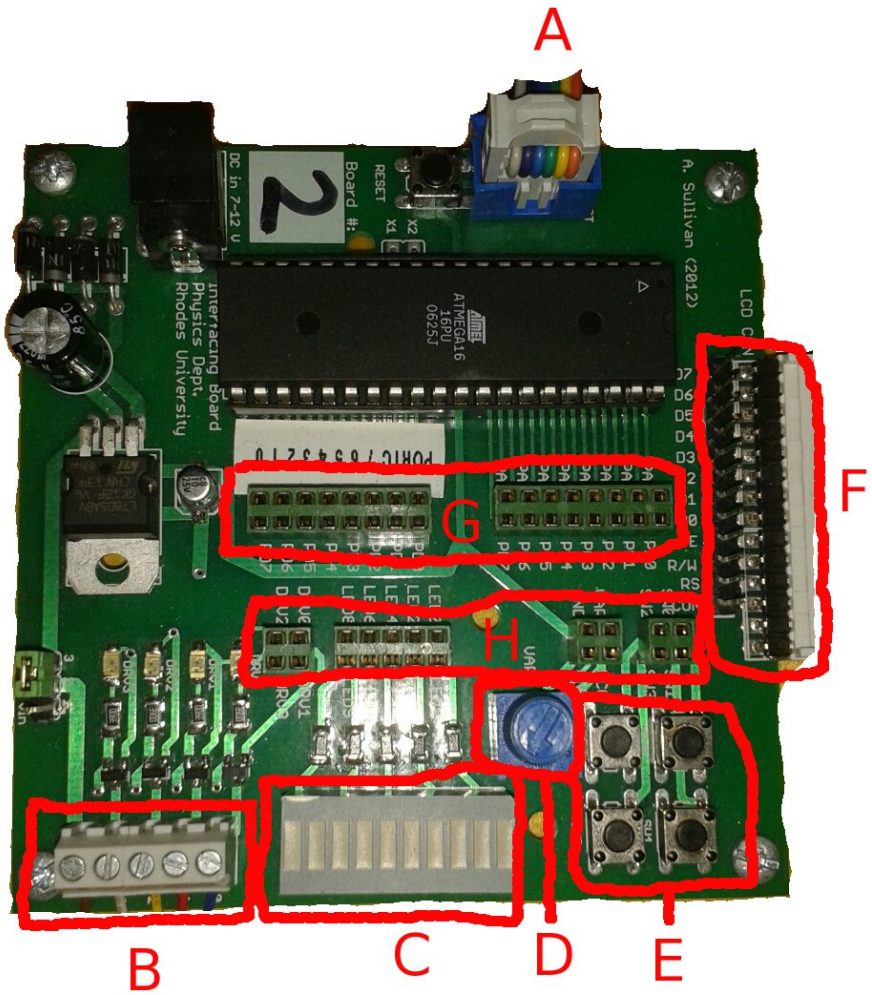


- A: This is the programming header, it is a standard Atmel 6-pin programmer ISP header. The jumper to the left should be on to supply power to the target device or off if the target device has its own power.

- 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.

CHAPTER

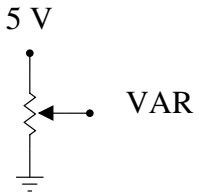
43 Devboard



- **A:** Programming and power connection for the devboard. Power is supplied from the programmer's USB connection. Programming connections connect directly with the MISO, MOSI,SCK and reset pins on the MCU.
- **B:** Stepper motor connection.
- **C:** LEDs 0-9
- **D:** Potentiometer to supply variable voltage to VAR connection. 0-5 V.
- **E:** Push to make buttons. (Pressed button connects output to ground - an internal pullup resistor is required to ensure a logic 1 when not pressed.)
- **F:** LCD connector.
- **G:** Port connections to PORTA-D on MCU.
- **H:** Connections to Stepper motor drive, LEDs, buttons and potentiometer.

43.1 Analog Section Schematic

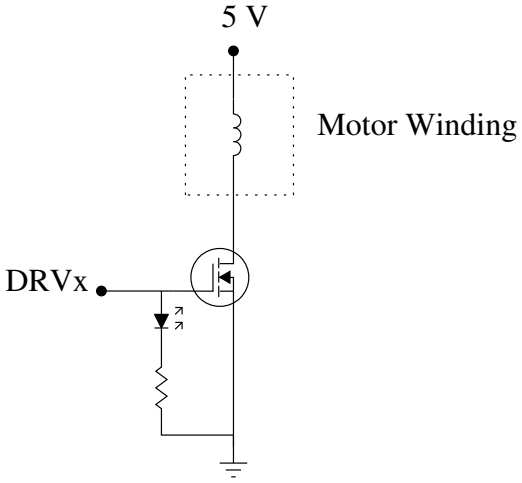
The potentiometer is connected between 5 V and 0 V so that the VAR connection can be set to any voltage in that range.



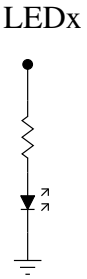
43.2 The LCD

See LCD Data sheet extract in appendices.

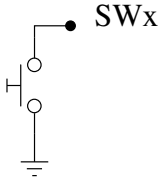
43.3 The Stepper Motor



43.4 The LEDs



43.5 Switches



PART

VII Appendices

CHAPTER 44

Appendix A - AVR Resources

44.1 m16def.inc Include File

```
;***** THIS IS A MACHINE GENERATED FILE - DO NOT EDIT ********************
;***** Created: 2005-01-11 10:30 ******* 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              : 2005-01-11
;* Version           : 2.14
;* 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
```

```
54
55
56 ; ***** I/O REGISTER DEFINITIONS ****************************************
57 ; NOTE:
58 ; Definitions marked "MEMORY MAPPED"are extended I/O ports
59 ; and cannot be used with IN/OUT instructions
60 .equ SREG = 0x3f
61 .equ SPH  = 0x3e
62 .equ SPL  = 0x3d
63 .equ OCR0 = 0x3c
64 .equ GICR = 0x3b
65 .equ GIFR = 0x3a
66 .equ TIMSK     = 0x39
67 .equ TIFR = 0x38
68 .equ SPMCSR    = 0x37
69 .equ TWCR = 0x36
70 .equ MCUCR     = 0x35
71 .equ MCUCSR    = 0x34
72 .equ TCCR0     = 0x33
73 .equ TCNT0     = 0x32
74 .equ OSCCAL    = 0x31
75 .equ OCDR = 0x31
76 .equ SFIOR     = 0x30
77 .equ TCCR1A    = 0x2f
78 .equ TCCR1B    = 0x2e
79 .equ TCNT1H    = 0x2d
80 .equ TCNT1L    = 0x2c
81 .equ OCR1AH    = 0x2b
82 .equ OCR1AL    = 0x2a
83 .equ OCR1BH    = 0x29
84 .equ OCR1BL    = 0x28
85 .equ ICR1H     = 0x27
86 .equ ICR1L     = 0x26
87 .equ TCCR2     = 0x25
88 .equ TCNT2     = 0x24
89 .equ OCR2 = 0x23
90 .equ ASSR = 0x22
91 .equ WDTCR     = 0x21
92 .equ UBRRH     = 0x20
93 .equ UCSRC     = 0x20
94 .equ EEARH     = 0x1f
95 .equ EEARL     = 0x1e
96 .equ EEDR = 0x1d
97 .equ EECR = 0x1c
98 .equ PORTA     = 0x1b
99 .equ DDRA = 0x1a
100 .equ PINA = 0x19
101 .equ PORTB     = 0x18
102 .equ DDRB = 0x17
103 .equ PINB = 0x16
104 .equ PORTC     = 0x15
105 .equ DDRC = 0x14
106 .equ PINC = 0x13
107 .equ PORTD     = 0x12
108 .equ DDRD = 0x11
109 .equ PIND = 0x10
110 .equ SPDR = 0x0f
111 .equ SPSR = 0x0e
112 .equ SPCR = 0x0d
113 .equ UDR  = 0x0c
114 .equ UCSRA     = 0x0b
115 .equ UCSRB     = 0x0a
116 .equ UBRRL     = 0x09
117 .equ ACSR = 0x08
118 .equ ADMUX     = 0x07
119 .equ ADCSRA    = 0x06
120 .equ ADCH = 0x05
121 .equ ADCL = 0x04
122 .equ TWDR = 0x03
123 .equ TWAR = 0x02
124 .equ TWSR = 0x01
125 .equ TWBR = 0x00
126
127
128 ; ***** BIT DEFINITIONS ************************************************
129
130 ; ***** TIMER_COUNTER_0 **************
131 ; 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  ; Comparet 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
.equ ADHSM     = 3  ; ADC High Speed Mode
.equ ADTS0     = 5  ; ADC High Speed Mode
.equ ADTS1     = 6  ; ADC Auto Trigger Source
.equ ADTS2     = 7  ; ADC Auto Trigger Source


; ***** 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
```

```
366 .equ SPDR6      = 6  ; SPI Data Register bit 6
367 .equ SPDR7      = 7  ; SPI Data Register bit 7
368
369 ; SPSR - SPI Status Register
370 .equ SPI2X      = 0  ; Double SPI Speed Bit
371 .equ WCOL = 6  ; Write Collision Flag
372 .equ SPIF = 7  ; SPI Interrupt Flag
373
374 ; SPCR - SPI Control Register
375 .equ SPR0 = 0  ; SPI Clock Rate Select 0
376 .equ SPR1 = 1  ; SPI Clock Rate Select 1
377 .equ CPHA = 2  ; Clock Phase
378 .equ CPOL = 3  ; Clock polarity
379 .equ MSTR = 4  ; Master/Slave Select
380 .equ DORD = 5  ; Data Order
381 .equ SPE  = 6  ; SPI Enable
382 .equ SPIE = 7  ; SPI Interrupt Enable
383
384
385 ; ***** USART ************************
386 ; UDR - USART I/O Data Register
387 .equ UDR0 = 0  ; USART I/O Data Register bit 0
388 .equ UDR1 = 1  ; USART I/O Data Register bit 1
389 .equ UDR2 = 2  ; USART I/O Data Register bit 2
390 .equ UDR3 = 3  ; USART I/O Data Register bit 3
391 .equ UDR4 = 4  ; USART I/O Data Register bit 4
392 .equ UDR5 = 5  ; USART I/O Data Register bit 5
393 .equ UDR6 = 6  ; USART I/O Data Register bit 6
394 .equ UDR7 = 7  ; USART I/O Data Register bit 7
395
396 ; UCSRA - USART Control and Status Register A
397 .equ USR  = UCSRA   ; For compatibility
398 .equ MPCM = 0  ; Multi-processor Communication Mode
399 .equ U2X  = 1  ; Double the USART transmission speed
400 .equ UPE  = 2  ; Parity Error
401 .equ PE   = UPE     ; For compatibility
402 .equ DOR  = 3  ; Data overRun
403 .equ FE   = 4  ; Framing Error
404 .equ UDRE = 5  ; USART Data Register Empty
405 .equ TXC  = 6  ; USART Transmitt Complete
406 .equ RXC  = 7  ; USART Receive Complete
407
408 ; UCSRB - USART Control and Status Register B
409 .equ UCR  = UCSRB   ; For compatibility
410 .equ TXB8 = 0  ; Transmit Data Bit 8
411 .equ RXB8 = 1  ; Receive Data Bit 8
412 .equ UCSZ2      = 2  ; Character Size
413 .equ CHR9 = UCSZ2   ; For compatibility
414 .equ TXEN = 3  ; Transmitter Enable
415 .equ RXEN = 4  ; Receiver Enable
416 .equ UDRIE      = 5  ; USART Data register Empty Interrupt Enable
417 .equ TXCIE      = 6  ; TX Complete Interrupt Enable
418 .equ RXCIE      = 7  ; RX Complete Interrupt Enable
419
420 ; UCSRC - USART Control and Status Register C
421 .equ UCPOL      = 0  ; Clock Polarity
422 .equ UCSZ0      = 1  ; Character Size
423 .equ UCSZ1      = 2  ; Character Size
424 .equ USBS = 3  ; Stop Bit Select
425 .equ UPM0 = 4  ; Parity Mode Bit 0
426 .equ UPM1 = 5  ; Parity Mode Bit 1
427 .equ UMSEL     = 6  ; USART Mode Select
428 .equ URSEL      = 7  ; Register Select
429
430 .equ UBRRHI     = UBRRH   ; For compatibility
431
432 ; ***** TWI ***************************
433 ; TWBR - TWI Bit Rate register
434 .equ I2BR = TWBR    ; For compatibility
435 .equ TWBR0      = 0  ;
436 .equ TWBR1      = 1  ;
437 .equ TWBR2      = 2  ;
438 .equ TWBR3      = 3  ;
439 .equ TWBR4      = 4  ;
440 .equ TWBR5      = 5  ;
441 .equ TWBR6      = 6  ;
442 .equ TWBR7      = 7  ;
443
```

```
444 ; TWCR - TWI Control Register
445 .equ I2CR = TWCR    ; For compatibility
446 .equ TWIE = 0   ; TWI Interrupt Enable
447 .equ I2IE = TWIE    ; For compatibility
448 .equ TWEN = 2   ; TWI Enable Bit
449 .equ I2EN = TWEN    ; For compatibility
450 .equ ENI2C      = TWEN    ; For compatibility
451 .equ TWWC = 3   ; TWI Write Collition Flag
452 .equ I2WC = TWWC    ; For compatibility
453 .equ TWSTO      = 4   ; TWI Stop Condition Bit
454 .equ I2STO      = TWSTO   ; For compatibility
455 .equ TWSTA      = 5   ; TWI Start Condition Bit
456 .equ I2STA      = TWSTA   ; For compatibility
457 .equ TWEA = 6   ; TWI Enable Acknowledge Bit
458 .equ I2EA = TWEA    ; For compatibility
459 .equ TWINT      = 7   ; TWI Interrupt Flag
460 .equ I2INT      = TWINT   ; For compatibility
461
462 ; TWSR - TWI Status Register
463 .equ I2SR = TWSR    ; For compatibility
464 .equ TWPS0      = 0   ; TWI Prescaler
465 .equ TWS0 = TWPS0   ; For compatibility
466 .equ I2GCE      = TWPS0   ; For compatibility
467 .equ TWPS1      = 1   ; TWI Prescaler
468 .equ TWS1 = TWPS1   ; For compatibility
469 .equ TWS3 = 3   ; TWI Status
470 .equ I2S3 = TWS3    ; For compatibility
471 .equ TWS4 = 4   ; TWI Status
472 .equ I2S4 = TWS4    ; For compatibility
473 .equ TWS5 = 5   ; TWI Status
474 .equ I2S5 = TWS5    ; For compatibility
475 .equ TWS6 = 6   ; TWI Status
476 .equ I2S6 = TWS6    ; For compatibility
477 .equ TWS7 = 7   ; TWI Status
478 .equ I2S7 = TWS7    ; For compatibility
479
480 ; TWDR - TWI Data register
481 .equ I2DR = TWDR    ; For compatibility
482 .equ TWD0 = 0   ; TWI Data Register Bit 0
483 .equ TWD1 = 1   ; TWI Data Register Bit 1
484 .equ TWD2 = 2   ; TWI Data Register Bit 2
485 .equ TWD3 = 3   ; TWI Data Register Bit 3
486 .equ TWD4 = 4   ; TWI Data Register Bit 4
487 .equ TWD5 = 5   ; TWI Data Register Bit 5
488 .equ TWD6 = 6   ; TWI Data Register Bit 6
489 .equ TWD7 = 7   ; TWI Data Register Bit 7
490
491 ; TWAR - TWI (Slave) Address register
492 .equ I2AR = TWAR    ; For compatibility
493 .equ TWGCE      = 0   ; TWI General Call Recognition Enable Bit
494 .equ TWA0 = 1   ; TWI (Slave) Address register Bit 0
495 .equ TWA1 = 2   ; TWI (Slave) Address register Bit 1
496 .equ TWA2 = 3   ; TWI (Slave) Address register Bit 2
497 .equ TWA3 = 4   ; TWI (Slave) Address register Bit 3
498 .equ TWA4 = 5   ; TWI (Slave) Address register Bit 4
499 .equ TWA5 = 6   ; TWI (Slave) Address register Bit 5
500 .equ TWA6 = 7   ; TWI (Slave) Address register Bit 6
501
502
503 ; ***** ANALOG_COMPARATOR *************
504 ; SFIOR - Special Function IO Register
505 .equ ACME = 3   ; Analog Comparator Multiplexer Enable
506
507 ; ACSR - Analog Comparator Control And Status Register
508 .equ ACIS0      = 0   ; Analog Comparator Interrupt Mode Select bit 0
509 .equ ACIS1      = 1   ; Analog Comparator Interrupt Mode Select bit 1
510 .equ ACIC = 2   ; Analog Comparator Input Capture Enable
511 .equ ACIE = 3   ; Analog Comparator Interrupt Enable
512 .equ ACI  = 4   ; Analog Comparator Interrupt Flag
513 .equ ACO  = 5   ; Analog Compare Output
514 .equ ACBG = 6   ; Analog Comparator Bandgap Select
515 .equ ACD  = 7   ; Analog Comparator Disable
516
517
518 ; ***** AD_CONVERTER *****************
519 ; ADMUX - The ADC multiplexer Selection Register
520 .equ MUX0 = 0   ; Analog Channel and Gain Selection Bits
521 .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 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   ;
.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


; ***** 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
```

```
678 .equ PORTC2    = 2  ; Port C Data Register bit 2
679 .equ PC2  = 2  ; For compatibility
680 .equ PORTC3    = 3  ; Port C Data Register bit 3
681 .equ PC3  = 3  ; For compatibility
682 .equ PORTC4    = 4  ; Port C Data Register bit 4
683 .equ PC4  = 4  ; For compatibility
684 .equ PORTC5    = 5  ; Port C Data Register bit 5
685 .equ PC5  = 5  ; For compatibility
686 .equ PORTC6    = 6  ; Port C Data Register bit 6
687 .equ PC6  = 6  ; For compatibility
688 .equ PORTC7    = 7  ; Port C Data Register bit 7
689 .equ PC7  = 7  ; For compatibility
690
691 ; DDRC - Port C Data Direction Register
692 .equ DDC0 = 0  ; Port C Data Direction Register bit 0
693 .equ DDC1 = 1  ; Port C Data Direction Register bit 1
694 .equ DDC2 = 2  ; Port C Data Direction Register bit 2
695 .equ DDC3 = 3  ; Port C Data Direction Register bit 3
696 .equ DDC4 = 4  ; Port C Data Direction Register bit 4
697 .equ DDC5 = 5  ; Port C Data Direction Register bit 5
698 .equ DDC6 = 6  ; Port C Data Direction Register bit 6
699 .equ DDC7 = 7  ; Port C Data Direction Register bit 7
700
701 ; PINC - Port C Input Pins
702 .equ PINC0     = 0  ; Port C Input Pins bit 0
703 .equ PINC1     = 1  ; Port C Input Pins bit 1
704 .equ PINC2     = 2  ; Port C Input Pins bit 2
705 .equ PINC3     = 3  ; Port C Input Pins bit 3
706 .equ PINC4     = 4  ; Port C Input Pins bit 4
707 .equ PINC5     = 5  ; Port C Input Pins bit 5
708 .equ PINC6     = 6  ; Port C Input Pins bit 6
709 .equ PINC7     = 7  ; Port C Input Pins bit 7
710
711
712 ; ***** PORTD ************************
713 ; PORTD - Port D Data Register
714 .equ PORTD0    = 0  ; Port D Data Register bit 0
715 .equ PD0  = 0  ; For compatibility
716 .equ PORTD1    = 1  ; Port D Data Register bit 1
717 .equ PD1  = 1  ; For compatibility
718 .equ PORTD2    = 2  ; Port D Data Register bit 2
719 .equ PD2  = 2  ; For compatibility
720 .equ PORTD3    = 3  ; Port D Data Register bit 3
721 .equ PD3  = 3  ; For compatibility
722 .equ PORTD4    = 4  ; Port D Data Register bit 4
723 .equ PD4  = 4  ; For compatibility
724 .equ PORTD5    = 5  ; Port D Data Register bit 5
725 .equ PD5  = 5  ; For compatibility
726 .equ PORTD6    = 6  ; Port D Data Register bit 6
727 .equ PD6  = 6  ; For compatibility
728 .equ PORTD7    = 7  ; Port D Data Register bit 7
729 .equ PD7  = 7  ; For compatibility
730
731 ; DDRD - Port D Data Direction Register
732 .equ DDD0 = 0  ; Port D Data Direction Register bit 0
733 .equ DDD1 = 1  ; Port D Data Direction Register bit 1
734 .equ DDD2 = 2  ; Port D Data Direction Register bit 2
735 .equ DDD3 = 3  ; Port D Data Direction Register bit 3
736 .equ DDD4 = 4  ; Port D Data Direction Register bit 4
737 .equ DDD5 = 5  ; Port D Data Direction Register bit 5
738 .equ DDD6 = 6  ; Port D Data Direction Register bit 6
739 .equ DDD7 = 7  ; Port D Data Direction Register bit 7
740
741 ; PIND - Port D Input Pins
742 .equ PIND0     = 0  ; Port D Input Pins bit 0
743 .equ PIND1     = 1  ; Port D Input Pins bit 1
744 .equ PIND2     = 2  ; Port D Input Pins bit 2
745 .equ PIND3     = 3  ; Port D Input Pins bit 3
746 .equ PIND4     = 4  ; Port D Input Pins bit 4
747 .equ PIND5     = 5  ; Port D Input Pins bit 5
748 .equ PIND6     = 6  ; Port D Input Pins bit 6
749 .equ PIND7     = 7  ; Port D Input Pins bit 7
750
751
752 ; ***** WATCHDOG *********************
753 ; WDTCR - Watchdog Timer Control Register
754 .equ WDP0 = 0  ; Watch Dog Timer Prescaler bit 0
755 .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
```

```
834
835
836
837 ; ***** INTERRUPT VECTORS ********************************************
838 .equ INT0addr   = 0x0002  ; External Interrupt Request 0
839 .equ INT1addr   = 0x0004  ; External Interrupt Request 1
840 .equ OC2addr    = 0x0006  ; Timer/Counter2 Compare Match
841 .equ OVF2addr   = 0x0008  ; Timer/Counter2 Overflow
842 .equ ICP1addr   = 0x000a  ; Timer/Counter1 Capture Event
843 .equ OC1Aaddr   = 0x000c  ; Timer/Counter1 Compare Match A
844 .equ OC1Baddr   = 0x000e  ; Timer/Counter1 Compare Match B
845 .equ OVF1addr   = 0x0010  ; Timer/Counter1 Overflow
846 .equ OVF0addr   = 0x0012  ; Timer/Counter0 Overflow
847 .equ SPIaddr    = 0x0014  ; Serial Transfer Complete
848 .equ URXCaddr   = 0x0016  ; USART, Rx Complete
849 .equ UDREaddr   = 0x0018  ; USART Data Register Empty
850 .equ UTXCaddr   = 0x001a  ; USART, Tx Complete
851 .equ ADCCaddr   = 0x001c  ; ADC Conversion Complete
852 .equ ERDYaddr   = 0x001e  ; EEPROM Ready
853 .equ ACIaddr    = 0x0020  ; Analog Comparator
854 .equ TWIaddr    = 0x0022  ; 2-wire Serial Interface
855 .equ INT2addr   = 0x0024  ; External Interrupt Request 2
856 .equ OC0addr    = 0x0026  ; Timer/Counter0 Compare Match
857 .equ SPMRaddr   = 0x0028  ; Store Program Memory Ready
858
859 .equ INT_VECTORS_SIZE     = 42 ; size in words
860
861 #endif  /* _M16DEF_INC_ */
862
863 ; ***** END OF FILE ****************************************************
```

ATMega16 Register Summary

ATmega16(L)

Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C	9
$3E ($5E)	SPH	–	–	–	–	–	SP10	SP9	SP8	12
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	12
$3C ($5C)	OCR0	Timer/Counter0 Output Compare Register								85
$3B ($5B)	GICR	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	48, 69
$3A ($5A)	GIFR	INTF1	INTF0	INTF2	–	–	–	–	–	70
$39 ($59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	85, 115, 133
$38 ($58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	86, 115, 133
$37 ($57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	250
$36 ($56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	180
$35 ($55)	MCUCR	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00	32, 68
$34 ($54)	MCUCSR	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF	41, 69, 231
$33 ($53)	TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	83
$32 ($52)	TCNT0	Timer/Counter0 (8 Bits)								85
$31[1] ($51)[1]	OSCCAL	Oscillator Calibration Register								30
	OCDR	On-Chip Debug Register								227
$30 ($50)	SFIOR	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	57,88,134,201,221
$2F ($4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	110
$2E ($4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	113
$2D ($4D)	TCNT1H	Timer/Counter1 – Counter Register High Byte								114
$2C ($4C)	TCNT1L	Timer/Counter1 – Counter Register Low Byte								114
$2B ($4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte								114
$2A ($4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte								114
$29 ($49)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte								114
$28 ($48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte								114
$27 ($47)	ICR1H	Timer/Counter1 – Input Capture Register High Byte								114
$26 ($46)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte								114
$25 ($45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	128
$24 ($44)	TCNT2	Timer/Counter2 (8 Bits)								130
$23 ($43)	OCR2	Timer/Counter2 Output Compare Register								130
$22 ($42)	ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB	131
$21 ($41)	WDTCR	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0	43
$20[2] ($40)[2]	UBRRH	URSEL	–	–	–	UBRR[11:8]				167
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	166
$1F ($3F)	EEARH	–	–	–	–	–	–	–	EEAR8	19
$1E ($3E)	EEARL	EEPROM Address Register Low Byte								19
$1D ($3D)	EEDR	EEPROM Data Register								19
$1C ($3C)	EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE	19
$1B ($3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	66
$1A ($3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	66
$19 ($39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	66
$18 ($38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	66
$17 ($37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	66
$16 ($36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	66
$15 ($35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	67
$14 ($34)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	67
$13 ($33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	67
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	67
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	67
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	67
$0F ($2F)	SPDR	SPI Data Register								142
$0E ($2E)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X	142
$0D ($2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	140
$0C ($2C)	UDR	USART I/O Data Register								163
$0B ($2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	164
$0A ($2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	165
$09 ($29)	UBRRL	USART Baud Rate Register Low Byte								167
$08 ($28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	202
$07 ($27)	ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	217
$06 ($26)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	219
$05 ($25)	ADCH	ADC Data Register High Byte								220
$04 ($24)	ADCL	ADC Data Register Low Byte								220
$03 ($23)	TWDR	Two-wire Serial Interface Data Register								182
$02 ($22)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	182

ATMega16 Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
$01 ($21)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	181
$00 ($20)	TWBR	Two-wire Serial Interface Bit Rate Register								180

Notes:
1. When the OCDEN Fuse is unprogrammed, the OSCCAL Register is always accessed on this address. Refer to the debugger specific documentation for details on how to use the OCDR Register.
2. Refer to the USART description for details on how to access UBRRH and UCSRC.
3. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
4. Some of the Status Flags are cleared by writing a logical one to them. Note that the CBI and SBI instructions will operate on all bits in the I/O Register, writing a one back into any flag read as set, thus clearing the flag. The CBI and SBI instructions work with registers $00 to $1F only.

AVR Instruction Set Summary

ATmega16(L)

Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← $FF – Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← $00 – Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • ($FF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← $FF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C	2
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP	k	Direct Jump	PC ← k	None	3
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL	k	Direct Subroutine Call	PC ← k	None	4
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd – Rr	Z, N,V,C,H	1
CPC	Rd,Rr	Compare with Carry	Rd – Rr – C	Z, N,V,C,H	1
CPI	Rd,K	Compare Register with Immediate	Rd – K	Z, N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1 / 2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1 / 2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1 / 2

AVR Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1 / 2
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None	2
LD	Rd, - X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, - Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None	2
ST	- X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None	2
ST	- Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None	3
SPM		Store Program Memory	(Z) ← R1:R0	None	-
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2
BIT AND BIT-TEST INSTRUCTIONS					
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow.	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1

AVR Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
MCU CONTROL INSTRUCTIONS					
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep function)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None	1
BREAK		Break	For On-Chip Debug Only	None	N/A

CHAPTER 45

Appendix B - LCD Resources

LCD.asm

```
.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
```

```
54            rjmp Loop_busy
55            LCD_WRITE ;
56            LCD_RS_LO ; rs lo
57            pop r16
58            ret
59
60 ; write char in r16 to LCD
61 Write_char:     ;rcall Check_busy
62          push r17
63            rcall Check_busy
64            LCD_WRITE
65            LCD_RS_HI
66            ser r17
67            out ddrc, r17  ; c output
68            out portc, R16
69            LCD_E_HI
70            LCD_E_LO
71            clr r17
72            out ddrc, r17
73            ;rcall delay
74            pop r17
75            ret
76 ;write instruction in r16 to LCD
77 Write_instruc:
78          push r17
79          rcall Check_busy
80            LCD_WRITE
81            LCD_RS_LO
82            ser r17
83            out ddrc, r17  ; c output
84            out portc, R16
85            ;rcall delay
86            LCD_E_HI
87          LCD_E_LO
88            clr r17
89            out ddrc, r17
90          ;rcall delay
91            pop r17
92            ret
93
94
95 Init_LCD: push r16
96            clr r16
97            out ddrc, r16
98            out portc, r16
99            sbi ddrb, 2    ;reg sel output
100           sbi ddrb, 0    ; enable output
101           sbi portb, 2
102           sbi portb, 0
103           sbi ddrb, 1    ; rw output
104           ldi r16, 0x38
105           rcall Write_instruc
106           ldi r16, 0x0c
107           rcall Write_instruc
108           ldi r16, 0x06
109           rcall Write_instruc
110           ldi r16, 0x01
111           rcall Write_instruc
112         pop r16
113           ret
```

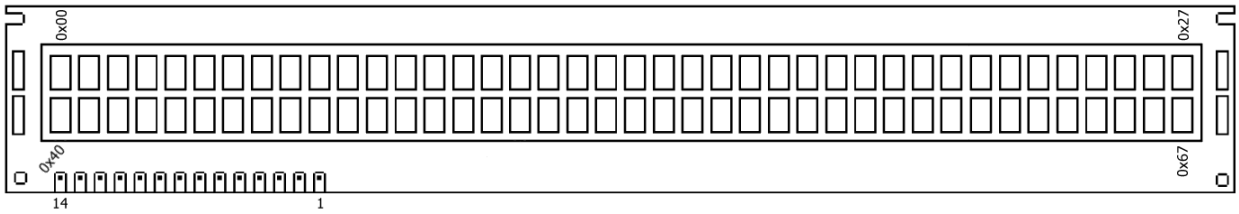
LCD Datasheet Extract
The *Extended Concise* LCD Data Sheet for HD44780

Version: 25.6.1999

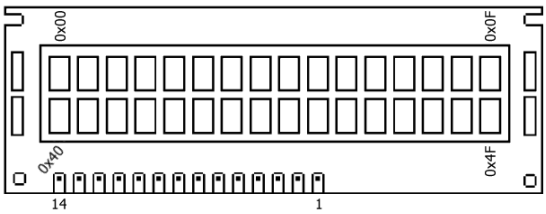
Instruction	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0	Description	Clock-Cycles
NOP	0	0	0	0	0	0	0	0	0	0	No Operation	0
Clear Display	0	0	0	0	0	0	0	0	0	1	Clear display & set address counter to zero	165
Cursor Home	0	0	0	0	0	0	0	0	1	x	Set adress counter to zero, return shifted display to original position. DD RAM contents 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 :

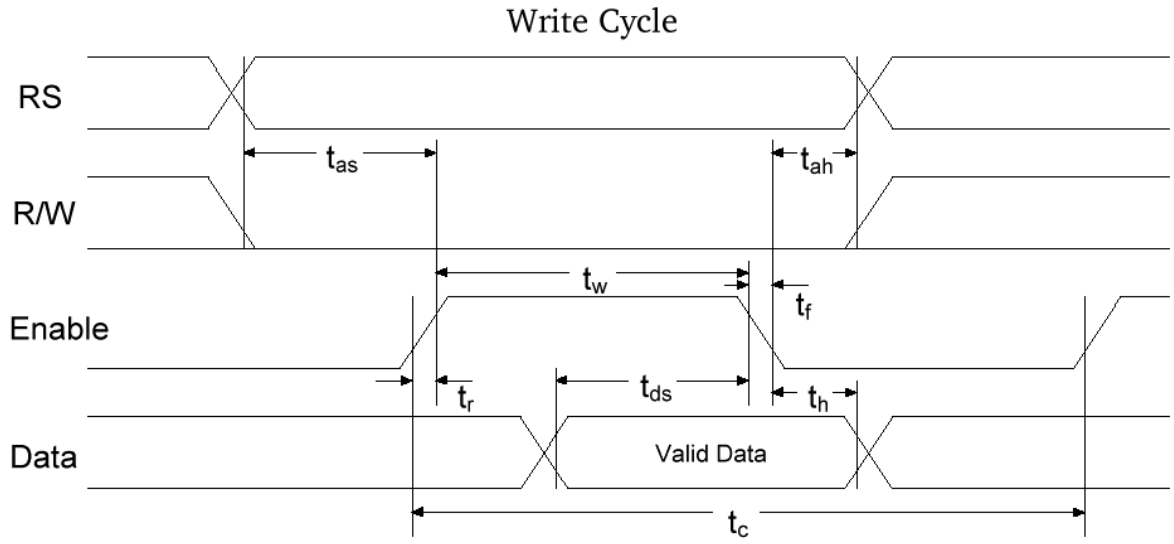


Pin No	Name	Function	Description
1	Vss	Power	GND
2	Vdd	Power	+ 5 V
3	Vee	Contrast Adj.	(-2) 0 - 5 V
4	RS	Command	Register Select
5	R/W	Command	Read / Write
6	E	Command	Enable (Strobe)
7	D0	I/O	Data LSB
8	D1	I/O	Data
9	D2	I/O	Data
10	D3	I/O	Data
11	D4	I/O	Data
12	D5	I/O	Data
13	D6	I/O	Data
14	D7	I/O	Data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

LCD Datasheet Extract

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

CHAPTER

46 Appendix C - ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com