AVR Project Book

(Revised)

Abdul Maalik Khan

Do It Yourself Book series from DigiSoft

Table of Content

List of Figures

Dedicated

To ALLAH Almighty,

All of HIS Messengers

Especially The Last Prophet MUHAMMAD (Peace be upon Him),

And My Family

ALLÂH is the Light of the Heavens and the Earth. The parable of His Light is as (if there were) a niche and within it a lamp, the lamp is in glass, the glass as it were a brilliant star, lit from a blessed tree, an olive, neither of the east nor of the west, whose oil would almost glow forth (of itself), though no fire touched it. Light upon Light! ALLÂH guides to HIS Light whom HE wills. And ALLÂH sets forth parables for mankind, and ALLÂH is All-Knower of everything. (NOOR: 35)

Preface

In our country the base of Engineering is not very broad, especially in the field of Electronics. Usually all the items has to be imported from other countries. Not only just components but Books, Kits, programmers, tools, Softwares, etc. This becomes a major bottleneck for someone studying the subject or involved as hobbyist, as he does not find the required items in local market and if he is unable to import. The idea of a Project Book came into my mind as I see that after spending more than 10 years in this field the situation is not changing. Importing everything required is not a practical thing to do for a majority of professionals in the developing countries. So I thought that there should be Books that will provide in one pack, all necessary Hardware, Software and guidelines to build some very basic projects. So the one who will use it will not need to buy any thing to work on the projects described in the book (other than a host computer and some inexpensive tools).

One more unique feature of this book (other than having all Hardware in a pack) is that a JTAG programmer/debugger is provided with the book. A JTAG Debugger is a very helpful tool and increases the learning curve significantly.

The Book is divided into six major sections; namely IO Ports, Timers, External Interrupt, UART, PWM and ADC. Although only these sections does not cover the complete Microprocessor. Having knowledge and a hand full of experience of these peripherals of the processor makes a good programming base for someone. After that he can build the systems using other peripherals of the Microprocessor.

While writing this book it is assumed that the reader knows the C language and is also familiar with some microcontroller programming. If the user is not familiar with C or Microcontroller basics, he should consult the relevant books first. This book is not a C language Programming Book or a Book to learn Microprocessor Fundamentals.

When I have started thinking about a project book, the first problem was to choose a microcontroller for the book. I decided ATMEGA16 from ATMEL as the processor to be discussed in this book, because of the peripheral it has, availability of JTAG Debugger, availability of GCC Compiler for AVR and availability of the processor in local market. The PCBs of this Project Book is built keeping in mind that they should be helpful to the user during running the projects from the book and even after he has finished and wanted to build something of his own.

Although most of the items are provided with the Project Book, some Tools will be required (or will help) to run the projects and are listed in “Items required to perform projects”.

Abdul Malik Khan


DigiSoft Islamabad, Pakistan
June 2008

Back Ground Information

Overview of the Processor

• High-performance, Low-power AVR® 8-bit Microcontroller
• Advanced RISC Architecture
  – 131 Powerful Instructions – Most Single-clock Cycle Execution
  – 32 x 8 General Purpose Working Registers

• High Endurance Non-volatile Memory segments
  – 16K Bytes of In-System Self-programmable Flash program memory
  – 512 Bytes EEPROM
  – 1K Byte Internal SRAM

• JTAG (IEEE std. 1149.1 Compliant) Interface

• 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

• I/O and Packages
  – 32 Programmable I/O Lines, 40 pin DIP package

• Operating Voltages
  – 4.5 - 5.5V for ATmega16

• Power Consumption @ 1 MHz, 3V, and 25°C for ATmega16L
  – Active: 1.1 mA, Idle Mode: 0.35 mA, Power-down Mode: < 1 µA

Introduction to AVR Studio

AVR Studio® is an Integrated Development Environment for writing and debugging AVR applications in Windows® 98/XP/ME/2000 and Windows NT® environments.

AVR Studio provides a project management tool, source file editor and chip simulator. It also interfaces with In-Circuit Emulators and development boards available for the AVR 8-bit RISC family of microcontrollers. Simplifying the development tasks, AVR Studio allows customers to significantly reduce time-to-market.

Features of AVR Studio
- Integrated Development Environment for Writing, Compiling and Debugging Software
- Fully Symbolic Source-level Debugger
- Configurable Memory Views, Including SRAM, EEPROM, Flash, Registers, and I/Os
- Unlimited Number of Break Points
- Online HTML Help

- Variable Watch/Edit Window with Drag-and-drop Function
- Extensive Program Flow Control Options
- Simulator Port Activity Logging and Pin Input Stimuli
- File Parser Support for COFF, UBROF6, UBROF8, and Hex Files
- Support for C, Pascal, BASIC and Assembly Languages

Introduction to AVR GCC (WinAVR)

The AVR GCC plug-in is a GUI front-end to GNU make and avr-gcc. The plug-in requires GNU make and avr-gcc for basic operations and avr-objdump from the AVR GNU binutils for generating list files. To avoid problems setting up the build environment, it is recommended to install the WinAVR distribution available at the WinAVR project's home page.

The plug-in component will automatically detect an installed WinAVR distribution and set up the required tools accordingly. An AVR GCC plug-in project is a collection of source files and configurations. A configuration is a set of options that specify how to build and link the files in a project. On creating a new project, the "default" configuration is created. A user can choose to continue using this configuration, adding/removing options as the project evolves or create one or more new configurations to use in the project.

AVFR-GCC Features

Integration of avr-gcc and Make in AVR Studio

Start the compiler, clean the project, set project options and debug the project from AVR Studio. Tools from the WinAVR distribution are detected by the plug-in.

GUI Controls to Manipulate Project Settings

Custom compile options can be set for specific files or all files in the project. Linker options can also be set. There are controls for optimization level, include directories, libraries, memory segments and more.

A Project Tree for Managing Project Files

A project tree provides easy access to and manipulation of every file in the project.

Work with Several Configurations

It is possible to define several sets of build options, called configurations.

Build Output

A build output view shows raw output from GNU make and avr-gcc. Error and warning messages that contain reference to a file and line can be double-clicked to open this file and put a marker on the line.

External Makefile

The plug-in allows the user to define a makefile to use for the build process.

Map and List Files

Map and list files can be generated on each build.

External Dependencies

The plug-in keeps track of dependencies on libraries and header files that are not part of a project.

GNU Assembler projects

A user can set up and work with projects that consist of both assembly files (.s) and C files. avr-gcc is used to assemble the .s files.

Projects

The projects presented in the following chapters are carefully selected that may be of interest of a newbie and experienced programmers at the same time.

A line by line explanation is given for the early projects, however general code description is given for the more complex projects, where a line by line explanation is not suitable.

Every project chapter of this book has the following format

- Introduction
- Brief Register summary (for more details of registers, consult ATMEGA16 DataSheet)
- Project Description
- Schematic of the project
- Board Layout to carryout project
- Code
- Brief explanation of the code
- Project Description, schematic, Board Layout, Code and explanation for the subsequent projects

Almost everything required to setup the environment for running projects is included in the book. However items listed in “Items required to perform projects” will be required or will help in effectively running the projects.

Making the Software Environment for running projects

Install AVR Studio and WinAVR from the CD. Installation procedure of the Software is given in the corresponding Directories.

It is a good idea to copy the projects folder from CD on the root directory of one of your Hard disk. If you choose to copy it to some other location, the file references given in this book may become different than what you will see on your computer.

Hardware Preparation for running projects

Connect JTAG Board and Project Board using the 10 pin Ribbon cable. Connect the JTAG Board to the PC using the 9 pin Cable supplied. If your computer does not have a Serial port, you may need to buy a USB to Serial Converter to connect PC and JTAG Board. Connect the appropriate power plug of the wall adapter to the Project Board (the JTAG Board will get the power from the Project Board through Ribbon Cable). The voltage selection on the wall adapter should be between 8V and 12V. Check that the power LED on the JTAG Board is ON. If you find the power LED on JTAG Board is OFF, you may need to reverse the polarity of the power plug.

The processor in the AVR Project Board is configured to work on 4 MHz internal RC oscillator. All projects presented in this book use this setting as default. If the default setting is required to be changed, refer to the AVR JTAG user guide for changing Fuse settings.

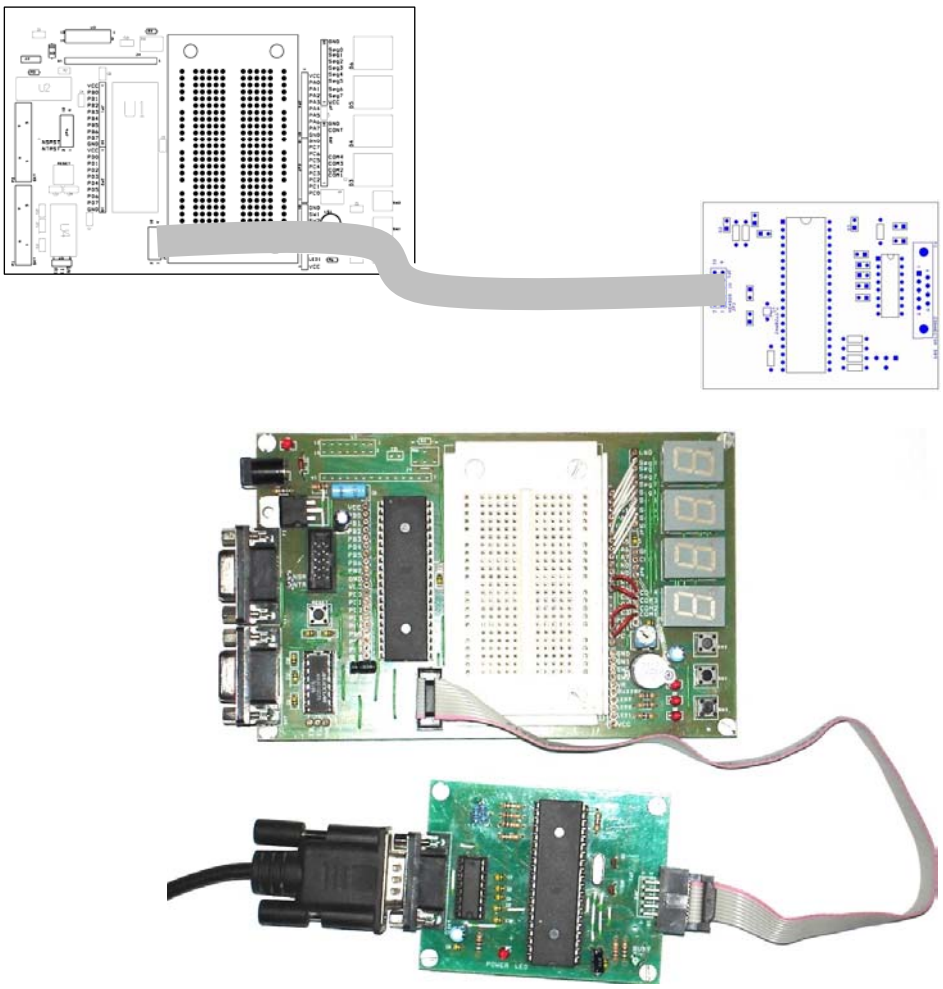


Figure II-1: Connecting Project Board with JTAG

The Prophet (saw) said,

"You should not be envious of anyone, except he upon whom ALLAH Bestows (the knowledge of) the Quran and he studies and practices it through the day and night; and the other is he whom ALLAH Gives wealth and he spends it in the cause of ALLAH, throughout the night and day." (Bukhari, Muslim)

1. I/O Ports

I/O Ports are the most basic interface to the Microprocessor. The port pins are grouped using 8 bit ports. There are 4 ports in ATMEGA16, namely, PORTA, PORTB, PORTC and PORTD.

Each port pin consists of three register bits: DDxn[1], PORTxn, and PINxn. 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 programmed as output, The PORTxn bit sets the logical value of the port pin. If programmed as input and the PORTxn is written logic one, 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.

The PINxn register is a read only register that reflects the current logical value on the port pin. The PINx register is primarily used when the port is programmed as input.

Summary of Registers

Special Function Register

– SFIOR (0x30)

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10	PORTx
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	

PORT x Data Register

– PORTx (0x1B – POARTA, 0x18 – PORTB, 0x15 – PORTC, 0x12 – PORTD)

Bit	7	6	5	4	3	2	1	0	
	PORTx7	PORTx6	PORTx5	PORTx4	PORTx3	PORTx2	PORTx1	PORTx0	PORTx
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	

PORT x Direction Register

– DDRx (0x1A – DDRA, 0x17 – DDRB, 0x14 – DDRC, 0x11 – DDRD)

[1] x = A..D, n = 0..7

Bit	7	6	5	4	3	2	1	0	
	DDx7	DDx6	DDx5	DDx4	DDx3	DDx2	DDx1	DDx0	DDRx
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	

PORT x input pins Address

– PINx (0x19 – PINA, 0x16 – PINB, 0x13 – PINC, 0x10 – PIND)

Bit	7	6	5	4	3	2	1	0	
	PINx7	PINx6	PINx5	PINx4	PINx3	PINx2	PINx1	PINx0	DDRx
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

The Register summary is provided here to check Bits Name in a Byte, Read / Write access to each bit, Initial Value of Each Bit, etc. For a details description of Bits in these Registers, please refer to the Data Sheet.

DDRxn	PORTxn	Description
0	0	Pin Configured as input, Pull up disabled. (Floating)
0	1	Pin Configured as input, Pull up enabled.
1	0	Pin Configured as output with a Low Level on pin. Pin can sink Current.
1	1	Pin Configured as output with a High Level on pin. Pin can source Current.

Project 1.a, LEDs

Although connecting an LED to AVR Microcontroller is the easiest thing to do it may be a good start to connecting things to microcontroller. The purpose of this project is to enable the microprocessor to express any event or something that a user might be interested in.

The Current capability of AVR Microprocessor is strong enough to drive an LED. A limiting resistor is however added to control the Light intensity. In order to instruct the Microprocessor to produce voltage level on its pin, the direction of the port pin is to be programmed as output. This is done through DDRx Register. The Reset Value of the DDRx register is such that all port pins are programmed as inputs, so we will exclusively program them as outputs. The direction setting is to be done on the first time only as during the life cycle of the program the direction of the pin will remain as output. After this a logic LOW on the pin will produce a Low Voltage Level and glow the LED and a logic HI on the pin turns LED off.

The schematic of the project is given below. Port C is chosen to drive LED because it is the closest port from LED.
Note: Pin 2, 3, 4 and 5 of PORTC is connected to JTAG Board for debugging purpose, and can not be used as general purpose pins.

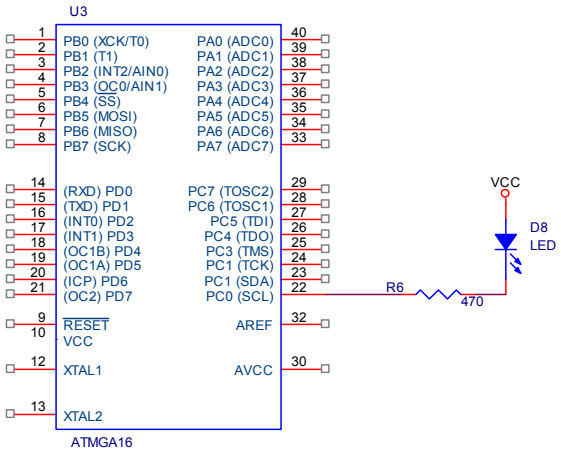


Figure III-1: Schematic for LED Interfacing

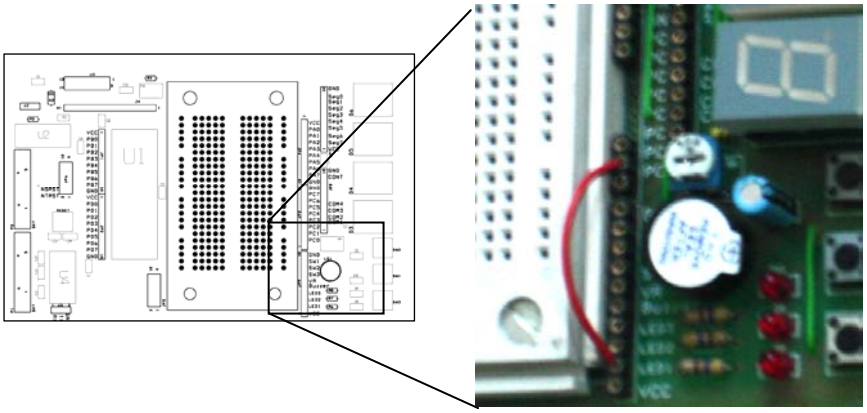


Figure III-2: Layout of wiring on Project Board for LED Interfacing

The LED can light by providing a LOW voltage on the Pin2 of the JP1 Connector. A wire is to be used to connect the connector pins to complete the circuit. After connecting wire if Bit 0 of the PORTC is made LOW at any time, the LED will light.

Following is the code that will make LED Blink at a rate of 0.5 Hz i.e. 1 sec on and 1 sec off. The Code is found on the CD at \<AVR BOOK>\CH1_Ports\LED

```
#include <avr/io.h>
int main (void)
{
      While(1){
            PORTC &= 0xFE;       //make Bit 0 of PORTB Low to turn LED on
            Delay1s();
            PORTC |= 0x01;       //make Bit 0 of PORTB High to turn LED off
            Delay1s();
      }
}
```

Function Delay1s is defined as

```
void Delay1s(void)
{
volatile unsigned int cnt;

      for (cnt = 0; cnt < 55555; cnt++); //~1sec Delay on a 1MHz clock
}
```

Project 1.b, Switches

Although connecting a push switch to AVR Microcontroller is the as easiest as connecting the LED to the port but it is also a part of connecting things to microcontroller. The purpose of this project is to get some user input to the microprocessor so that it could know when the user wants certain thing to happen. An LED is added to show the user that the Switch is acknowledged.

In order to read from a pin, it has to be programmed as **input**. While a pin of AVR Microcontroller is in input configuration it does not drive the port pin and the port pin is said to be **floating**. A floating pin can be taken to any voltage level by even a weak drive or EMI. In order to connect an other (partly) floating component (like Switch) a pull up is required to make the clear logic level on the pin.

The schematic of the project is given below. Port C is used as it is the closest port from the Switches and LEDs connector. Note that the Pin 2, 3, 4 and 5 are left unconnected as they are connected to JTAG.

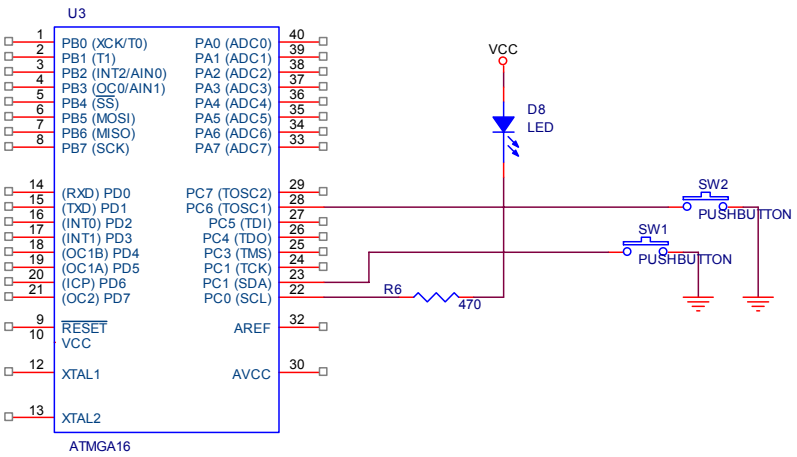


Figure III-3: Schematic for switch Interfacing

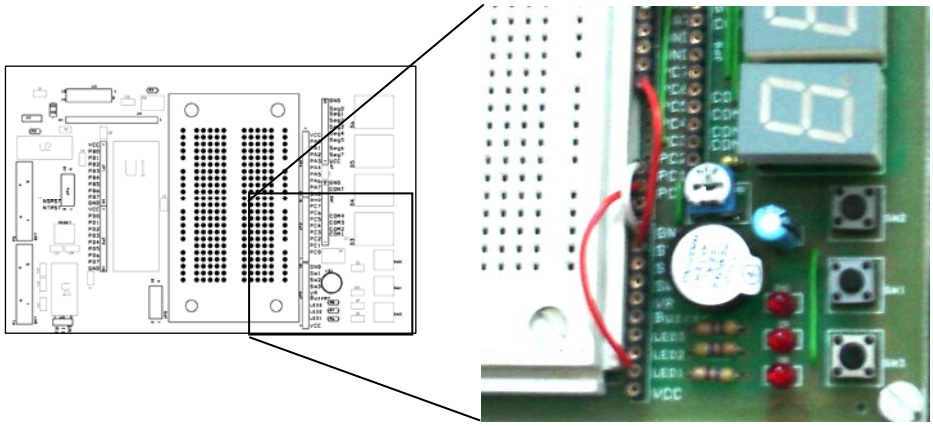


Figure III-4: Layout of wiring on Project Board for Switch Interfacing

We will have to program the port pin as input and also enable the internal pull-up so that the Noise from other components (EMI) will not make a false indication.

So if we check the status of **pin** it will normally read as **1** but when the user will push the button, the pin will become Low, and it will be read as **0** for the time the user is pressing the key. So in order to reliably detect the key, we have to check it at a rate faster than the smartest user could press the key. Fortunately, it is not a very big deal with the AVR.

Following is the code that checks the status of the key press on SW1 and turn on an LED for the time the key is pressed. The Code is found on the CD at \<AVR BOOK>\CH1_Ports\Switch1

```
1.  #include <avr/io.h>
2.
3.  /*Following is the code that checks the
4.  status of the key press on SW1 and turn
5.  on an LED for the time the key is pressed*/
6.
7.  int main (void)
8.  {
9.  DDRC = 0x01;                    //make Bit 0 of PORTC to output
10. PORTC = 0x02;                   //Enable Pull-ups
11. while(1) {
12.    if (PINC & 0x02) {
13.       PORTC |= 0x01;        //make Bit 0 of PORTC High to turn LED off
14.    } else {
15.       PORTC &= 0xFE;        //make Bit 0 of PORTC Low to turn LED on
16.    }
17. }
```

```
18. }
```

Line by Line Explanation of the code

Line 1: Inclusion of IO Header File
Line 2 - 6: Comments
Line 7 - 8: Start of Function Main
Line 9 - 10: Initialization
Line 11: Start of Infinite Loop
Line 12: Check for the Switch
Line 13: Turn LED Off if Condition was true (i.e. the switch was not pressed)
Line 14: Else
Line 15: Turn LED On if Condition was not true (i.e. the switch was pressed)
Line 16: Conclusion of If statement for the Check of Switch
Line 17: Conclusion of Infinite Loop
Line 18: Conclusion of the Application

The following code turns LED on the key press of SW1 and turn off the LED on press of SW2. The Code is found on the CD at <AVR BOOK>\CH1_Ports\Switch2

```
1.  #include <avr/io.h>
2.  int main (void)
3.  {
4.  DDRC = 0x01;                        //make Bit 0 of PORTC to output
5.  PORTC = 0x42;                       //Enable Pull-ups
6.     while(1) {
7.        if ((PINC & 0x02) == 0) {
8.           PORTC &= 0xFE;          //make Bit 0 of PORTB Low to turn LED on
9.        }
10.       if ((PINC & 0x40) == 0) {
11.          PORTC |= 0x01;          //make Bit 0 of PORTB High to turn LED off
12.       }
13.    }
14. }
```

Line 1: Inclusion of IO Header File
Line 2-3: Start of Function Main
Line 4-5: Initialization
Line 6: Start of Infinite Loop
Line 7: Check for the Switch 1
Line 8: Turn LED On if Condition was true (i.e. the switch was pressed)
Line 9: Conclusion of If statement for the Check of Switch
Line 10: Check for the Switch 2
Line 11: Turn LED Off if Condition was true (i.e. the switch was pressed)
Line 12: Conclusion of If statement for the Check of Switch
Line 13: Conclusion of Infinite Loop
Line 14: Conclusion of the Application

Something to try

1. Moving party Lights.
   Three LEDs can be made moving if they glow in sequence that creates a movement impression.

2. 3 bit binary counter
   Three LEDs can be used the make a binary display in which each LED represents a bit. The value represented by the LED can be increased by a push of a button.

3. 3 bit Dice
   A Dice can be made by three LEDs (as they can represent a number between 0-7). The system should change the value at a very high rate, so that nobody could estimate it, and it should keep running for the period in which the button is pushed.

4. Combinational Lock
   Using three switches a combinational lock can be made, in which the user has press the switches in certain order to open the lock. The Lock status can be shown on a LED.

So whatever you have been given is but (a passing) enjoyment for this worldly life, but that which is with ALLÂH (i.e. Paradise) is better and more lasting for those who believe (in the Oneness of ALLÂH) and put their trust in their Lord (Sho'a'raa: 36)

2. Timers

ATMEGA16 has two 8 bits and one 16 bit Timer / counter.

Following is some definition of terms specifically used in Counter.

BOTTOM	The counter reaches the BOTTOM when it becomes 0x00.
MAX	The counter reaches its Maximum when it becomes 0xFF (or 0xFFFF).
TOP	The counter reaches the TOP when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be the fixed value 'MAX' or the value stored in the OCRx Register. The assignment is dependent on the mode of operation.

The main part of the Timer/Counter is the programmable bi-directional counter unit. Depending of the mode of operation used, the counter is cleared, incremented, or decremented at each timer clock (clk_{T0}). clk_{T0} can be generated from an external or internal clock source, selected by the Clock Select bits (CSxn). When no clock source is selected (CSxn:0 = 0) the timer is stopped. However, the TCNTx value can be accessed by the CPU, regardless of whether clkT0 is present or not. A CPU write overrides (has priority over) all counter clear or count operations. The counting sequence is determined by the setting of the WGMxn bits located in the Timer/Counter Control Register (TCCRx). There are close connections between how the counter behaves (counts) and how waveforms are generated on the Output Compare output OCx.

The Timer/Counter Overflow (TOVx) Flag is set according to the mode of operation selected by the WGMxn bits. TOVx can be used for generating a CPU interrupt.

The comparator continuously compares TCNTx with the Output Compare Register (OCRx). Whenever TCNTx equals OCRx, the comparator signals a match. A match will set the Output Compare Flag (OCFx) at the next timer clock cycle. If enabled (OCIEx = 1 and Global Interrupt Flag in SREG is set), the Output Compare Flag generates an output compare interrupt. The OCFx Flag is automatically cleared when the interrupt is executed. Alternatively, the OCFx Flag can be cleared by software by writing a logical one to its I/O bit location. The waveform generator uses the match signal to generate an output according to operating mode set by the WGMxn bits and Compare Output mode (COMxn) bits. The MAX and BOTTOM signals are used by the waveform generator for handling the special cases of the extreme values in some modes of operation.

Summary of Registers

Timer/Counter Control Register

– TCCR0 (0x33)
– TCCR2 (0x25)

Bit	7	6	5	4	3	2	1	0	
Timer 0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Timer 2	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	

Timer/Counter Register
– TCNT0 (0x32)
– TCNT2 (0x24)

Bit	7	6	5	4	3	2	1	0	
Timer 0	TCNT0[7:0]								TCNT0
Timer 2	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	

Output Compare Register
– OCR0 (0x3C)
– OCR2 (0x23)

Bit	7	6	5	4	3	2	1	0	
Timer 0	OCR0[7:0]								OCR0
Timer 2	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	

Timer/Counter Interrupt Mask
– TIMSK (0x39)

Bit	7	6	5	4	3	2	1	0	
Timer 0							OCIE0	TOIE0	
Timer 1			TICIE1	OCIE1A	OCIE1B	TOIE1			TIMSK
Timer 2	OCIE2	TOIE2							
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	

Timer/Counter Interrupt Flag
– TIFR (0x38)

Bit	7	6	5	4	3	2	1	0	
Timer 0							OCF0	TOV0	
Timer 1			ICF1	OCF1A	OCF1B	TOV1			TIMSK
Timer 2	OCF2	TOV2							
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	

Timer/Counter1 Control Register A
– TCCR1A (0x2F)

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	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Timer/Counter1 Control Register B
– TCCR1B (0x2E)

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
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	

Timer/Counter1
– TCNT1H and TCNT1L (0x2D and 0x2C)

Bit	7	6	5	4	3	2	1	0	
	TCNT1[15:8]								TCNT1H

	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Output Compare Register 1 A

– OCR1AH and OCR1AL (0x2B and 0x2A)

Bit	7	6	5	4	3	2	1	0	
	OCR1A[15:8]								OCR1AH
	OCR1A[7:0]								OCR1AL
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	

Output Compare Register 1 B

– OCR1BH and OCR1BL (0x29 and 0x28)

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

Input Capture Register 1

– ICR1H and ICR1L (0x27 and 0x26)

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

The Register summary is provided here to check Bits Name in a Byte, Read / Write access to each bit, Initial Value of Each Bit, etc. For a details description of Bits in these Registers, please refer to the DataSheet.

Mode	WGM01 WGM21	WGM00 WGM20	Timer / Counter Mode of operation	TOP	Update of OCR0 / OCR2	TOV0 / TOV2 Flag Set on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0 / OCR2	Immediate	MAX
3	1	1	Fast PWM	0xFF	TOP	MAX

Waveform Generation Mode Bit Description for Timer 0 and Timer 2

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

13	1	1	0	1	Reserved	–	–	–
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

Waveform Generation Mode Bit Description for Timer 1

COM01/COM1A1/ COM1B1/COM21	**COM00/COM1A0/ COM1B0/COM20**	**Description**
0	0	Normal port Operation, OC0 / OC1A / OC1B / OC2 Disconnected
0	1	Toggle OC0 / OC1A / OC1B / OC2 on Compare Match
1	0	Clear OC0 / OC1A / OC1B / OC2 on Compare Match
1	1	Set OC0 / OC1A / OC1B / OC2 on Compare Match

CS00 CS10 CS20	CS01 CS11 CS21	CS02 CS12 CS22	Counter 0	Counter 1	Counter 2
0	0	0	No Clock Source (Counter Stopped)		
0	0	1	clk_{IO}	clk_{IO}	Clk_{T2S}
0	1	0	clk_{IO} / 8	clk_{IO} / 8	Clk_{T2S} / 8
0	1	1	clk_{IO} / 64	clk_{IO} / 64	Clk_{T2S} / 32
1	0	0	clk_{IO}/ 256	clk_{IO}/ 256	Clk_{T2S} / 64
1	0	1	clk_{IO}/1024	clk_{IO}/1024	Clk_{T2S} / 128
1	1	0	External Clock Source on T0 pin. Clock on Falling Edge	External Clock Source on T1 pin. Clock on Falling Edge	Clk_{T2S} / 256
1	1	1	External Clock Source on T0 pin. Clock on Rising Edge	External Clock Source on T1 pin. Clock on Rising Edge	Clk_{T2S} / 1024

Clock Select Bit Description

Project 2.a, Blinking Light

We have seen the light blinking in Project 1.a, but the Timing is calculated through a delay loop, here we will use a Hardware Timer of the ATMEGA16 to calculate the timings. In this method the Timer Register is polled and compared with a pre calculated value and when it reaches to that value, the time elapsed is exactly as required. The function exits after adjusting the Timer Register to restart timing. This method of calculating time has some disadvantages too, but it has clear advantages over the previous method. The disadvantage of this method is that it is still a polling method that requires continuous polling of the timer register. So the time for which the polling is done no other task can be performed. The advantage over the previous method is that it is more accurate than previous method in all respects, as the timing unit is in hardware. So it has a better timeout accuracy (even if some interrupts are running) and it has better repeat accuracy. The repeat accuracy is particularly useful in cases where timing is to be maintained for a long period of time like in digital clock.

The schematic for the project is same as project 1.a.

The code for polling the timer to calculate the timing is given below. The code can be found in the CD at the location <AVR BOOK>\CH2_Timer\Blink

```
#include <avr/io.h>
int main (void)
{
    TCCR1B = (1 << CS10) | (1 << CS11);        //Set the clock divisor as 64
    While(1){
        PORTC &= 0xFE;      //make Bit 0 of PORTB Low to turn LED on
        Delay1s();
        PORTC |= 0x01;      //make Bit 0 of PORTB High to turn LED off
        Delay1s();
    }
}
```

Function Delay1s is defined as

```
void Delay1s(void)
{
    While(TCNT1 < 15625);
    TCNT1 -= 15625;
}
```

The Short Comings of the previous project can be removed by using the Timer Interrupt (which may be essential for the bigger projects), This method is not discussed here but left to the reader.

Project 2.b, Clock

The Previous projects may be trivial for most of the readers, but this project is bit complex than previously discussed ones. So it is divided into following steps.

Step 1. Display a Digit

To display a digit the port pins connected to the segments of 7 segment LEDs are to be configured in way that the segments that are required to glow are programmed as logic high. And the port bit corresponding to the common wire of the 7 segment LED that is required to glow is programmed as logic low. Function 'disp' defined in the code for this project is used to set the port pin as desired. This function will be used throughout this book whenever 7 segment LEDs are used in any project.

Step 2. Displaying a 4 digit Hex Number

To 4 a 4 digit Number requires periodically updating each 7 segment LED at rate faster than the human eye could see. This makes the display appear to the user as all 4 digits of the Display are glowing simultaneously.

Step 3. Displaying a 4 digit Decimal Number

The microprocessor keeps each number in the memory in binary (or hexadecimal) format. To display it in decimal format (as it is more understandable to humans) requires some calculations. Function power is used to generate the powers of 10, that is required to convert the hex numbers to decimal.

Step 4. Displaying a Digital Clock

A digital clock requires a time base of 1 sec on which a variable for the seconds is incremented. After the seconds reach to 60, minute variable is incremented and second variable is reset to zero. After the minutes reach to 60, hour variable is incremented and minute variable is reset to zero. The number to display on the 7 segments is made by multiplying hours by 100 and added to the minute. This number is displayed the same way as in the previous project.

1. Display a Digit

Following is the code to display a number at one of the 7 segment LED. Position of the Number and Number it self are the parameters of the function **disp**. The code can be found in the CD at the location <AVR BOOK>\CH2_Timer\Clock1

```
#include <avr/io.h>

void disp(unsigned char col, unsigned char num);

/*Following is the code to display a number (stored in variable num) at position '0'*/

unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};

int main (void)
{
```

```
    DDRA = 0xFF;
    DDRC = 0xC3;
    disp(1, 7);
    while(1);
}

void disp(unsigned char col, unsigned char num)
{
PORTC = 0xFF;
PORTA = image[num];
if(col == 0) PORTC = ~(1 << 0);
if(col == 1) PORTC = ~(1 << 1);
if(col == 2) PORTC = ~(1 << 6);
if(col == 3) PORTC = ~(1 << 7);
}
```

2. Displaying a 4 digit Hex Number

Following is the code to display a 4 digit Hex number in the 7 segment LED. The code can be found in the CD at the location <AVR BOOK>\CH2_Timer\Clock2

```
#include <avr/io.h>

void disp(unsigned char col, unsigned char num);

unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};

int main (void)

{
unsigned char ind;
unsigned int num;

    DDRA = 0xFF;
    DDRC = 0xC3;
    num = 0x89AB;

    while(1){
        disp(ind,  (num / (0x0001 << (ind * 4))) & 0x0F);
        ind++;
        ind &= 3;
    }
}
```

Function ‘disp’ is defined as above

3. Displaying a 4 digit Decimal Number

Following is the code to display a 4 digit decimal number in the 7 segment LEDs. The code can be found in the CD at the location <AVR BOOK>\CH2_Timer\Clock3

```
#include <avr/io.h>
#include <math.h>

void disp(unsigned char col, unsigned char num);
unsigned long power(unsigned long mentissa, unsigned char pow);

unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};

int main (void)
```

```
{
unsigned char ind;
unsigned int num;

      DDRA = 0xFF;
      DDRC = 0xC3;
      num = 1324;
      while(1){
            disp(ind,  (num / power(10, ind) % 10);
            ind++;
            ind &= 3;
      }
}

unsigned long power(unsigned long mentissa, unsigned char pow)
{
      if(pow == 0) return 1;
      if(pow == 1) return mentissa;
      if(pow == 2) return mentissa * mentissa;
      if(pow == 3) return mentissa * mentissa * mentissa;
return 0;
}
```

4. Displaying a Digital Clock

So far a timer is not used but we are going to use it now. Here one Timer is used to periodically refresh the LEDs and another timer is used to maintain timing. It is evident in the code that through the use of the timers, the code is quite simplified.

The code can be found in the CD at the location <AVR BOOK>\CH2_Timer\Clock4

```
#include <avr/io.h>
#include <avr/interrupt.h>

#define BV(n) (1 << (n))

void disp(unsigned char col, unsigned char num);
unsigned long power(unsigned long mentissa, unsigned char pow);

unsigned int num;
/*Timer will be used to periodically refresh the LEDs and the timing will also be calculated through timers.*/
/*Following is the code to display a number (stored in variable num) at position '0'*/

unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};

int main (void)
{
unsigned char sec, min, hr;

      TCCR0 = BV(CS00);
      TCCR1B = BV(CS10) | BV(CS11);
      DDRC = 0xC3;                  //make Bits of PORTC as output
      DDRA = 0xFF;                  //make All Bits of PORTA as output
      TIMSK = BV(TOIE0);
      sei();

      sec = min = 0;
      hr = 12;

      while(1){
            TCNT1 -= (15625 * 2);
            while(TCNT1 < (15625 * 2));
```

```
            sec++;
            if (sec >= 60) {sec = 0; min++;}
            if (min >= 60) {min = 0; hr++;}
            if (hr > 12) hr = 1;
            num = hr * 100 + min;
        }
}

unsigned char ind;

SIGNAL (SIG_OVERFLOW0)
{
        disp(ind,  (unsigned int)(num / power(10, ind)) % 10);
        ind++;
        ind &= 3;
}
```

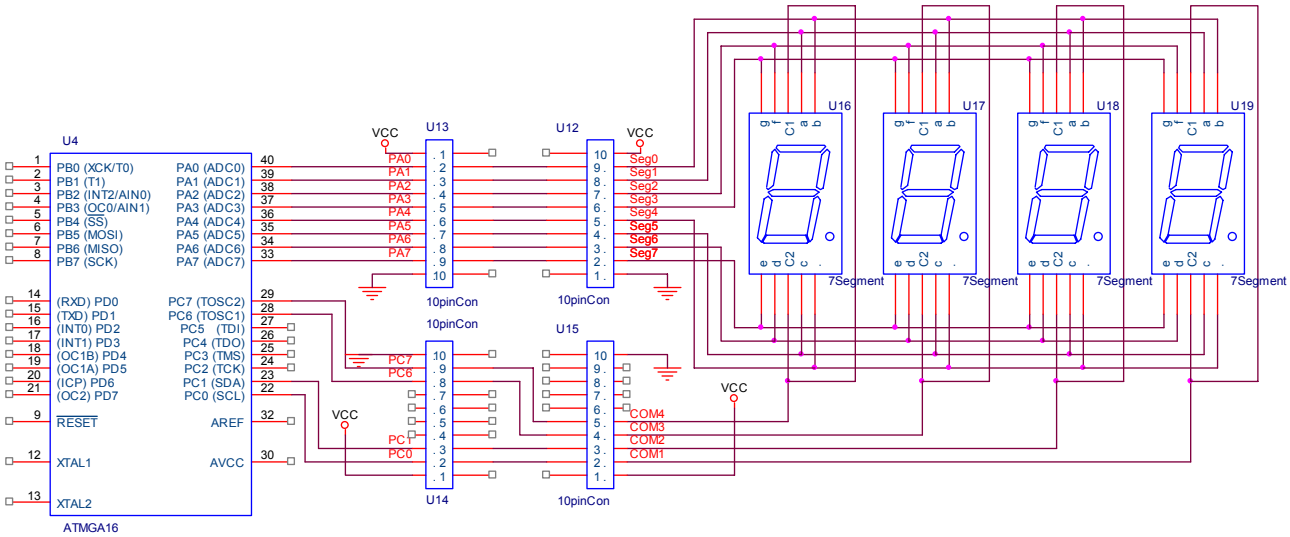


Figure IV-1: Schematic for Digital Clock

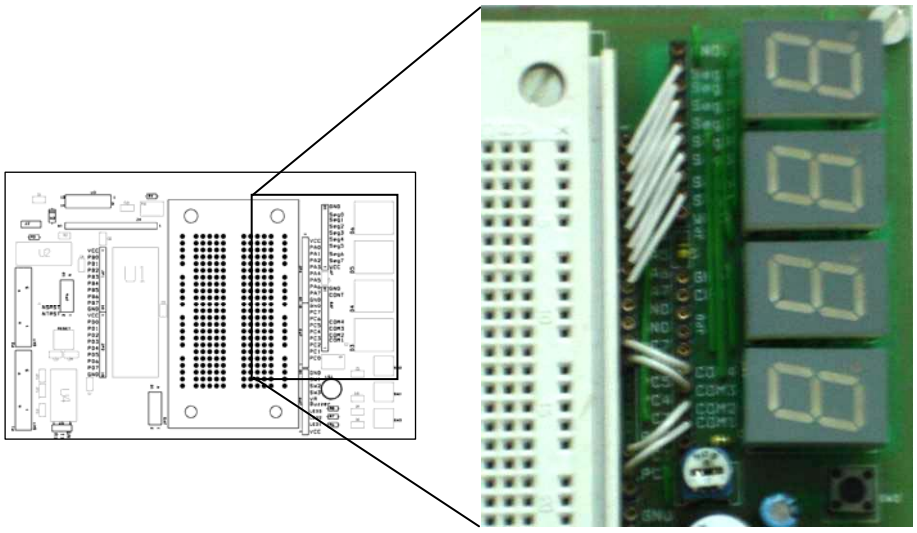


Figure IV-2: View of the wiring on Project Board for Digital Clock

Something to try

1. Moving party Lights.
   Three LEDs can be made moving if they glow in sequence that creates a movement impression. This project is to done using timers.

2. Up / down counter
   Using the 7 segment LEDs, an up / down counter can be built. One switch is used to up count and another for down count.

3. Alarm Clock
   A good project is to modify the clock software to make an Alarm Clock.

4. Display of a fraction Number
   A fraction Number is one that has a decimal point in it. Make a program that shows a decimal Number on 7 Segment LEDs.

Say, "He is the One who initiated you, and granted you the hearing, the eyes, and the brains. Rarely are you appreciative." Say, "He is the One who placed you on earth, and before Him you will be summoned." (Al-Mulk: 23-24)

3. USART

The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a highly flexible serial communication device. The main features of AVR USART 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 Over-Run 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**

The USART Baud Rate Register (UBRR) and the down-counter connected to it function as a programmable prescaler or baud rate generator. The down-counter, running at system clock (fosc), is loaded with the UBRR value each time the counter has counted down to zero or when the UBRRL Register is written. A clock is generated each time the counter reaches zero. This clock is the baud rate generator clock output (= fosc/(UBRR+1)). The Transmitter divides the baud rate generator clock output by 2, 8 or 16 depending on mode. The baud rate generator output is used directly by the receiver's clock and data recovery units. However, the recovery units use a state machine that uses 2, 8 or 16 states depending on mode set by the state of the UMSEL, U2X and DDR_XCK bits.

Equations for Calculating Baud Rate Register Setting

Operating Mode	Equation for Calculating Baud Rate[1]	Equation for Calculating UBRR Value
Asynchronous Normal Mode (U2X = 0)	$BAUD = \frac{f_{OSC}}{16(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{16BAUD} - 1$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \frac{f_{OSC}}{8(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{8BAUD} - 1$
Synchronous Master Mode	$BAUD = \frac{f_{OSC}}{2(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{2BAUD} - 1$

Note: 1. The baud rate is defined to be the transfer rate in bit per second (bps).
BAUD Baud rate (in bits per second, bps)
f_{OSC} System Oscillator clock frequency
UBRR Contents of the UBRRH and UBRRL Registers, (0 - 4095)

Summary of Registers

USART I/O Data Register

– UDR (0x0C)

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	

USART Control and Status Register A

– USCRA (0x0B)

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	

USART Control and Status Register B

– USCRB (0x0A)

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	

USART Control and Status Register C

– USCRC (0x20)

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	

USART Baud Rate Register

– UBRRL and UBRRH (0x09 and 0x20)

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	

The Register summary is provided here to check Bits Name in a Byte, Read / Write access to each bit, Initial Value of Each Bit, etc. For a details description of Bits in these Registers, please refer to the DataSheet.

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

UPM Bits Setting

USBS	Stop Bits
0	1 – Stop Bit
1	2 – Stop Bits

USBS Bit Setting

UCSZ2	UCSZ1	UCSZ0	Description
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

UCSZ Bits Setting

Project 3.a, Transmit through Serial Port

In this project some arbitrary text is transmitted through Serial Port in Buffered and non-buffered mode. The level of difficulty in writing a buffered Serial port is evident from the code, but the buffered serial port gives a lot of efficiency otherwise.

The schematic of the project is given below. No wires connections are required to do the project as all the connections are on PCB.

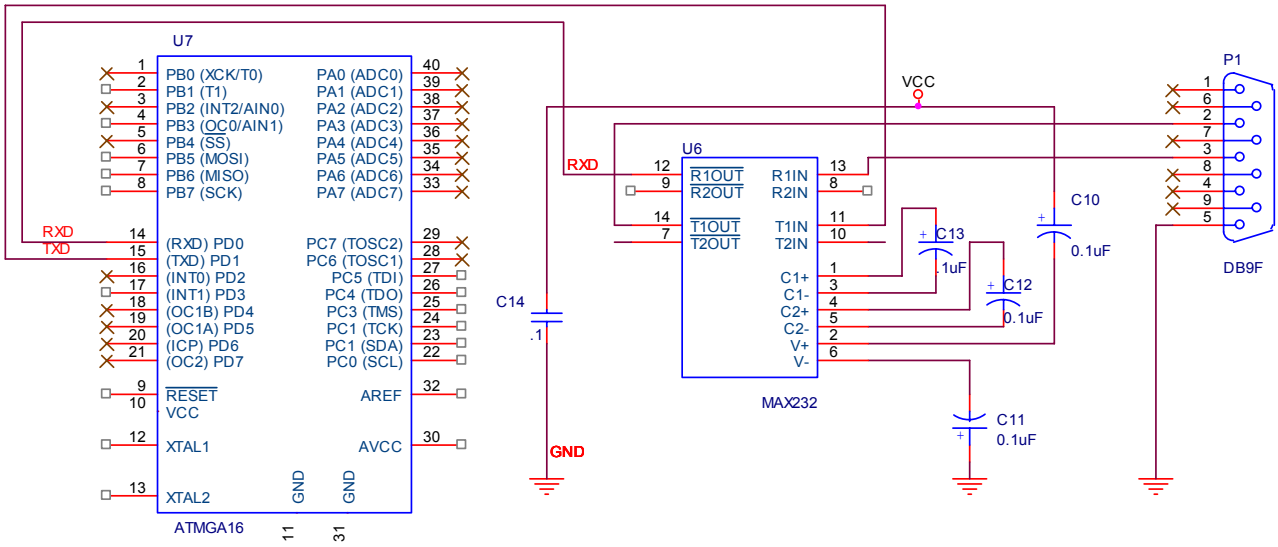


Figure V-1: Schematic for Serial Port Transmit

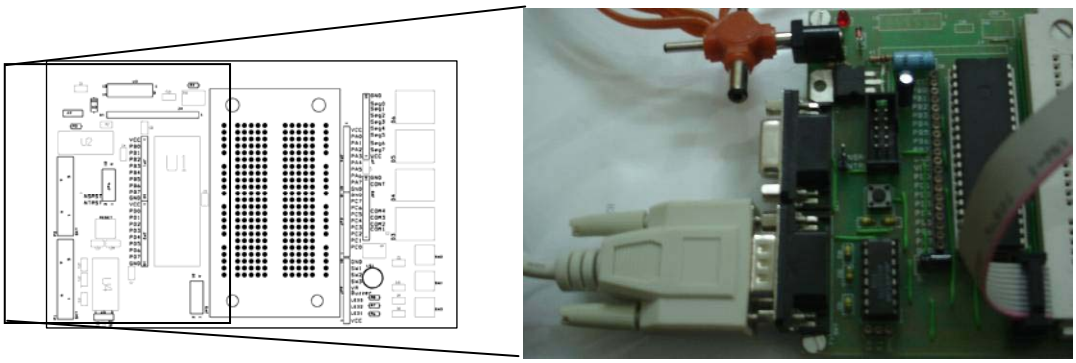


Figure V-2: A view of Serial cable connected

Following is the code that will transmit the text in non-buffered mode. The code can be found in the CD at the location <AVR BOOK>\CH3_UART\Trans

```
#include "avr/io.h"

void USART_Transmit( unsigned char data );

int main (void)
{
unsigned char i, Message[] = "Hello World";

      UCSRA = 0;
      UCSRB = 1<<TXEN; //Transmitter Enable
      UCSRC = 1<<URSEL | 1<<UCSZ1 | 1<<UCSZ0;        //8 Data bit 1 stop bit and No parity
      UBRRH = 0;
      UBRRL = 25; // 9600 Baude @ 4 Mhz

      for(i=0; Message[i]; i++)
                  USART_Transmit(Message[i]);
      while(1);
}


void USART_Transmit( unsigned char data )
{
      /* Wait for transmit buffer to be empty*/
      while ( !( UCSRA & (1<<UDRE))) ;
      /* Put data into buffer, sends the data */
      UDR = data;
}
```

Code Explanation

The initialization consists of setting up USART Register for asynchronous serial port with 8 data no parity and one stop bit. After which the baud rate is programmed for 9600 Baud for the processor running at 4MHz.

Then the function USART_Transmit is called for each byte in the message. The USART_Transmit function takes one byte of data. It first wait until the Transmit buffer is empty and then places the given byte into the transmit register.

When all of the bytes has been transmitted, the while(1) loop waits indefinitely till the controller is restarted.

The code of the buffered Transmit is not presented here but included in the CD and can be found at the location <AVR BOOK>\CH3_UART\Tr_Buff

Project 3.b, Receive Through Serial Port

In this project, characters are received through serial port and their ASCII Code is displayed on the 7 segment Display. This project is done using buffered (Interrupt based) and non-buffered (poling method) mode.

The schematic of the project is given below. No wires connections are required to do the project as all the connections are already on PCB.

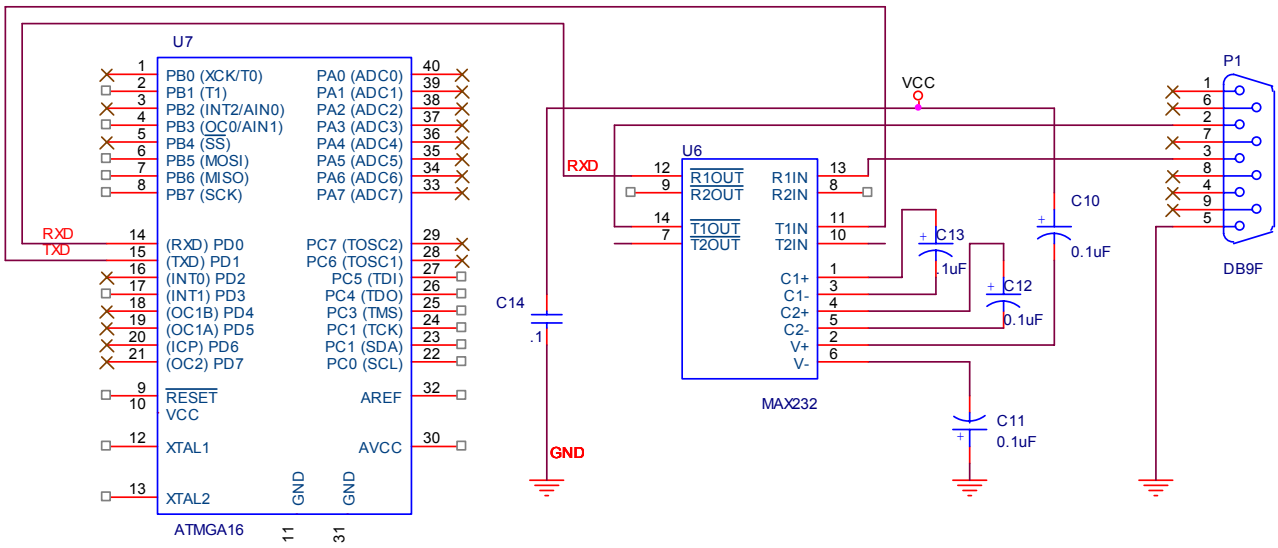


Figure V-3: Schematic for serial port Receive

Following is the code of the Serial Receive through polling. The code can be found in the CD at the location <AVR BOOK>\CH3_UART\Receive

```
#include "avr/io.h"
#include "avr/interrupt.h"

#define BitVal(x) (1 << (x))

#define FAILURE       0
#define SUCCESS       (~FAILURE)

#define SERIALPORTDIR       DDRD
#define SERIALPORT PORTD
#define DD_TXD 1
#define DD_RXD 0

#define DATA_BITS_5           3
#define DATA_BITS_6           1
#define DATA_BITS_7           2
#define DATA_BITS_8           0
#define DATA_BITS_9           4
#define DATA_BITS_MASK    (DATA_BITS_5 | DATA_BITS_6 | DATA_BITS_7 | DATA_BITS_8 | DATA_BITS_9)

#define NO_PARITY   0
#define EVEN_PARITY           8
#define ODD_PARITY0x10
#define PARITY_BITS_MASK  (NO_PARITY | EVEN_PARITY | ODD_PARITY)

#define STOP_BIT_1  0
#define STOP_BIT_2  0x20
#define STOP_BITS_MASK    (STOP_BIT_1 | STOP_BIT_2)

#define X2_MODE_MASK       0x40
#define X2MODE       0x40


unsigned long num;
unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};
```

```
void disp(unsigned char col, unsigned char num);
unsigned int power(unsigned char number, unsigned char pow);
unsigned char InitUART (unsigned int BaudRate, unsigned char Stop_Parity_NumBits);
unsigned char UART_Receive(unsigned char * ch );


int main (void)
{
unsigned char Buf;

      DDRC = 0xC3;                        //make Bit 0, 1, 6, 7 of PORTC as output
      DDRA = 0xFF;                        //make All Bits of PORTA as output
      TCCR0 = BitVal(CS00);
      TIMSK = BitVal(TOIE0);              //Set Timer 0 Interrupt

      InitUART (9600, DATA_BITS_8 | NO_PARITY | STOP_BIT_1);

      sei();

      while(1) {
            if(UART_Receive(&Buf))
               num = Buf;
      }
}


unsigned char UART_Receive(unsigned char * ch )
{
      if (UCSRA & BitVal(RXC)) {
            * ch = UDR;
            return SUCCESS;
      } else {
            return FAILURE;                                          //return 0 if no data is received
      }
}

unsigned char InitUART (unsigned int BaudRate, unsigned char Stop_Parity_NumBits)
{
unsigned int CV, BR;
unsigned int ERR8, ERR16;

      CV = ((F_CPU / 16 + BaudRate / 2) / BaudRate) - 1;
      BR = (F_CPU/((CV+1)*16));
      ERR16 = BaudRate > BR ? BaudRate - BR : BR - BaudRate;
      CV = ((F_CPU / 8 + BaudRate / 2) / BaudRate) - 1;
      BR = (F_CPU/((BR+1)*8));
      ERR8 = BaudRate > BR ? BaudRate - BR : BR - BaudRate;

      if(ERR16 < ERR8){
      CV = ((F_CPU / 16 + BaudRate / 2) / BaudRate) - 1;
      } else {
            Stop_Parity_NumBits |= X2MODE;
      }

      SERIALPORTDIR &= ~BitVal(DD_RXD);
      SERIALPORT |= BitVal(DD_RXD); //Disable pull up on receive
      UBRRH = (unsigned char) CV >> 8; // Set the baud Rate
      UBRRL = (unsigned char) CV; // Set the baud Rate
      //Enable UART receiver, Transmitter and receive interrupt
      UCSRB = BitVal(RXCIE) | BitVal(RXEN);

      if((Stop_Parity_NumBits & DATA_BITS_MASK) == DATA_BITS_8){
            //Asynchronous serial 8 Data N parity 1 stop bits
            UCSRC = BitVal(URSEL) | BitVal(UCSZ1) | BitVal(UCSZ0);
```

```
    }
    if((Stop_Parity_NumBits & X2_MODE_MASK) == X2MODE){
        UCSRA |= U2X;
    }
    return SUCCESS;
}

unsigned char ind;
SIGNAL (SIG_OVERFLOW0)
{
    disp(ind,  (unsigned int)(num / power(10, ind)) % 10);
    ind++;
    ind &= 3;
}
```

Code Explanation

The initialization consists of setting up Timers and Port Registers. Serial port is initialized through 'InitUART'. The InitUART function is a very comprehensive function for UART Initialization as it takes care of different speed modes of controller, current clock frequency etc. and decide for a best possible baud rate generator register value. It takes the data bit size, parity and stop bit values through parameters and sets the USART Registers accordingly..

After initialization the system continuously wait for a byte to be received from UART and then displays its Value. To get a byte from USART, UART_Receive function is called which checks the current port state and return SUCCES if a byte has been received or failure if the receive register is empty.

The received byte is displayed in the same way as discussed in the digital clock section.

The code of the buffered Receive is not printed here but included in the CD and can be found at the location <AVR BOOK>\CH3_UART\Rec_Buff

Also included in the CD is the project that shows how to receive and transmit at the same time using buffers. It can be found at the location <AVR BOOK>\CH3_UART\Tr_Rec

Something to try

1. Sending of Custom message on each key press
   The Transmit project may be changed to send a different message on each key press.

2. Receive and Transmit the same string
   The Receive / Transmit project is to be changed to transmit the complete string once a carriage return is detected on the receive line.

3. Table printing
   A good project is to make a table machine that accepts a number and sends out a formatted table of that number through the serial port that can be shown properly on a terminal program.

He (ALLÂH) is the One who created the heavens and the earth in six days, then assumed all authority. He knows everything that enters into the earth, and everything that comes out of it, and everything that comes down from the sky, and everything that climbs into it. He is with you wherever you may be. ALLÂH is Seer of everything you do. (Al-Hadeed:4)

4. External Interrupt

The External Interrupts are triggered by the INT0, INT1, and INT2 pins. Observe that, if enabled, the interrupts will trigger even if the INT0..2 pins are configured as outputs. This feature provides a way of generating a software interrupt. The external interrupts can be triggered by a falling or rising edge or a low level (INT2 is only an edge triggered interrupt). This is set up as indicated in the specification for the MCU Control Register – MCUCR – and MCU Control and Status Register – MCUCSR. When the external interrupt is enabled and is configured as level triggered (only INT0/INT1), the interrupt will trigger as long as the pin is held low. Note that recognition of falling or rising edge interrupts on INT0 and INT1 requires the presence of an I/O clock, described in "Clock Systems and their Distribution" in ATMEGA16 Data Sheet. Low level interrupts on INT0/INT1 and the edge interrupt on INT2 are detected asynchronously. This implies that these interrupts can be used for waking the part also from sleep modes other than Idle mode. The I/O clock is halted in all sleep modes except in Idle mode.

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. This makes the MCU less sensitive to noise. The changed level is sampled twice by the Watchdog Oscillator clock. The period of the Watchdog Oscillator is 1 µs (nominal) at 5.0V and 25°C. The MCU will wake up if the input has the required level during this sampling or if it is held until the end of the start-up time. The start-up time is defined by the SUT fuses as described in "System Clock and Clock Options" in ATMEGA16 Data Sheet. If the level is sampled twice by the Watchdog Oscillator clock but disappears before the end of the start-up time, the MCU will still wake up, but no interrupt will be generated. The required level must be held long enough for the MCU to complete the wake up to trigger the level interrupt.

Summary of Registers

MCU Control Register — The MCU Control Register contains control bits interrupt sense control and general MCU functions

– MCUCR (0x35)

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
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	

MCU Control and Status Register

– MCUCSR (0x34)

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					

General Interrupt Control Register

– GICR (0x3B)

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	-	-	-	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

General Interrupt Flag Register

– GIFR (0x3A)

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	-	-	-	-	-	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

ICS01 ICS11	ICS00 ICS10	Description
0	0	The Low Level of INT0 / INT1 generates an interrupt request
0	1	Any Logical Change on INT0 / INT1 generates an interrupt request
1	0	The Falling Edge of INT0 / INT1 generates an interrupt request
1	1	The Rising Edge of INT0 / INT1 generates an interrupt request

Interrupt Sense Control for INT 0 and INT 1

ICS2	Description
0	The Falling Edge of INT2 generates an interrupt request
1	The Rising Edge of INT2 generates an interrupt request

Interrupt Sense Control for INT 2

Project 4.a, Elapsed Time

Period measurement of a signal may be of interest in many situations, like accurate measurement of low frequencies, phase shift measurement etc.

Here in this project we measure a person's quickness by measuring the time between two consecutive presses of key. The time period measured will be displayed on the 7 segment LEDs.

To measure the cycle time, a free running counter count time between two cycles.

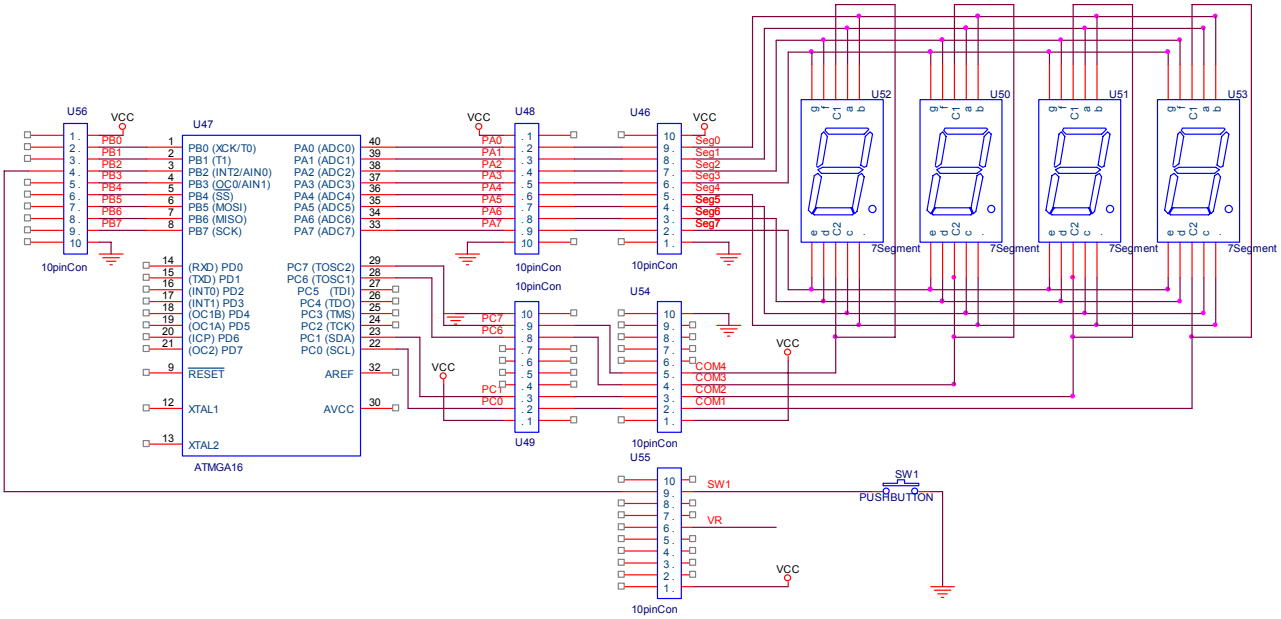


Figure VI-1: Schematic for Period Measurement

Circuit of Project 2.a. will be used to display the Key Response Time

```
//************************************************************
//This program calclates and display time elapsed between two
//consequitive key press
//************************************************************

#include <avr/io.h>
#include <avr/interrupt.h>

unsigned char Flag;
unsigned long num;

void disp(unsigned char col, unsigned char num);
unsigned int power(unsigned char number, unsigned char pow);

unsigned long num;
unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};


int main (void)
{

        TCCR0 = 0x01;
        TCCR1B = 0x04;
        GICR = 0x20;
        sei();

        while(1){
        }
}

unsigned char ind;

SIGNAL (SIG_OVERFLOW0)
{
        disp(ind,  (num / power(10, ind) % 10));
        ind++;
        ind &= 3;
}

SIGNAL (SIG_INTERRUPT2)
{
        if(Flag == 0) {
              TCNT1 = 0;
              Flag++;
        } else {
              num = TCNT1 * 256 / 1000;
              Flag = 0;
        }
}
```

Function ‘disp’ and ‘power’ are defined in previous Projects

Code Explanation

The initialization consists of setting up Timers and Port Registers. After that the system just waits and the software actually run in background (i.e. Intterupts)

When the key is pressed, the ISR for the External interrupt is activated. It checks if it’s the first time the key is pressed. In this case it resets the Timer 1. If the key is pressed second time, it updates the display variable based on the time elapsed between the key presses.

Project 4.b, Software Serial Port

Although ATMEGA16 has an on-chip serial port, but in many situations it can be desired to have one or two more serial ports. Addition of the serial ports can be achieved through software serial ports. The Software Over head to accomplish the task is usually large in case the receive pin is to be polled. Using external interrupt, the overhead can be reduced drastically.

The Schematic of the system is given below.

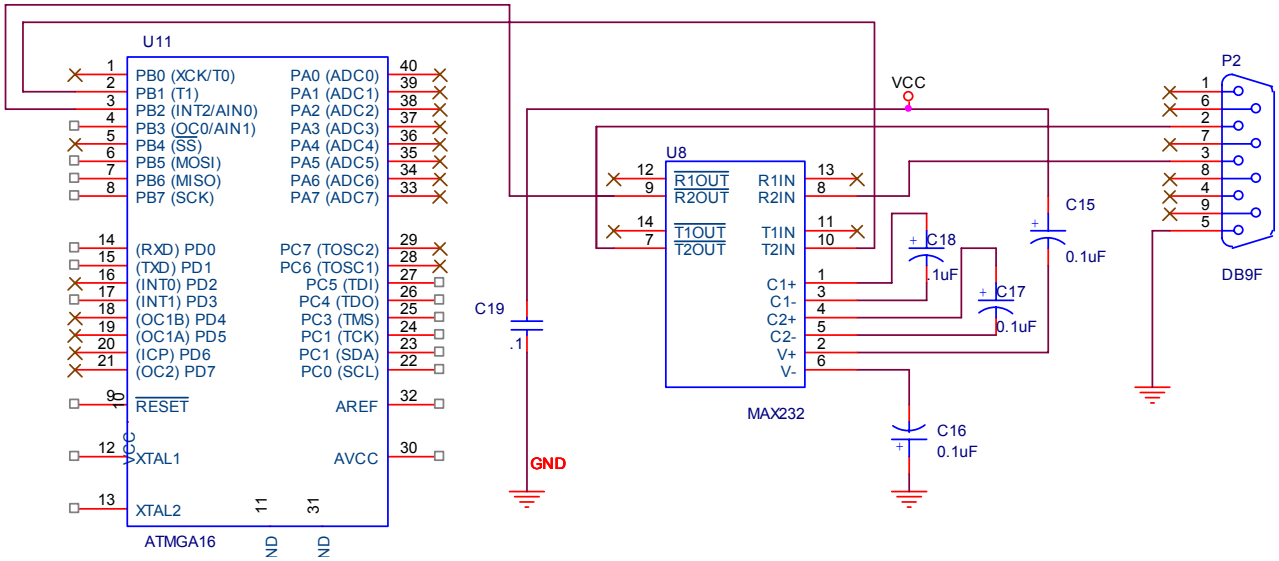


Figure VI-2: Schematic for Software Serial Port

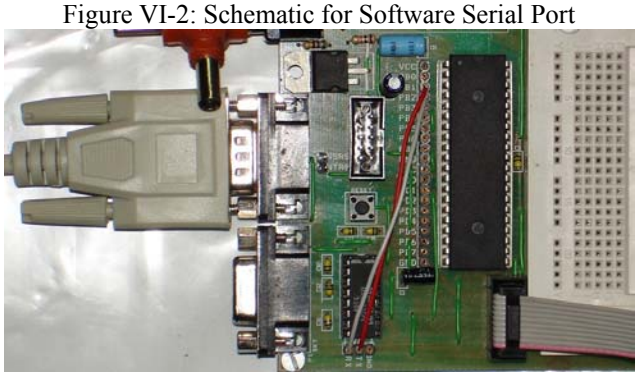


Figure VI-3: Picture showing wiring of Software Serial port

```
#include "avr/io.h"
#include "avr/interrupt.h"

void uart_transmit(char data);
void uart_init(void);


volatile unsigned char   N =   208;    // 19200@4Mhz
volatile unsigned char u_status, u_reload, u_bit_cnt, u_buffer;


#define   RDR   0
#define   TD    6
#define   BUSY  7

#define TX 1
```

```
#define RX 2

unsigned char Message[] = "Hello World\n\r";

void main (void)
{
unsigned char i;

        uart_init();
        sei();
        for(i = 0; Message[i]; i++)
                uart_transmit(Message[i]);
        while(1);

}


//***************************************************************************
//*
//* EXT_INT2 - External Interrupt Routine 2
//*
//*
//* DESCRIPTION
//* This routine is executed when a negative edge on the incoming serial
//* signal is detected. It disables further external interrupts and enables
//* timer interrupts (bit-timer) because the UART must now receive the
//* incoming data.

SIGNAL (SIG_INTERRUPT2)
{
  u_status =  (1<<BUSY);
  TCNT0 = (256-(N+N/2)+(29));
  TIFR = (1<<TOIE0);
  TIMSK = (1<<TOIE0);
  u_bit_cnt = 0;
  u_reload  = 256-N;
}
//***************************************************************************
//* TIM0_OVF - Timer/Counter 0 Overflow Interrupt
//*
//* DESCRIPTION
//* This routine coordinates the transmission and reception of bits. This
//* routine is automatically executed at a rate equal to the baud-rate. When
//* transmitting, this routine shifts the bits and sends it. When receiving,
//* it samples the bit and shifts it into the buffer.
//*
//* The serial routines uses a status register (u_status): READ-ONLY.
//*    BUSY  This bit indicates whenever the UART is busy
//*    TD  Transmit Data. Set when the UART is transmitting
//*    RDR  Receive Data Ready. Set when new data has arrived
//*      and it is ready for reading.
//*
//* When the RDR flag is set, the (new) data can be read from u_buffer.
//*
//* This routine also sets bits in TIMSK.

SIGNAL (SIG_OVERFLOW0)
{
        TCNT0 = u_reload + TCNT0;
        u_bit_cnt++;
        if((u_status & (1<<TD)))    // Transmit section
        {
                if(u_bit_cnt <= 7)
                {
                        if(u_buffer & 0x01)
```

```
                        {
                                PORTB|=(1<<TX);
                        } else {
                                PORTB &=(~(1<<TX));
                        }
                u_buffer = u_buffer>>1;
                } else {
                        PORTB|=(1<<TX);
                }
                if((u_bit_cnt == 8))
                {
                        u_status &= (~(1<<BUSY | 1<<TD));
                TIMSK = 0;
                }
        } else          // Receive routine
        {
                if(!(u_bit_cnt== 9))
                {
                        u_buffer = (u_buffer>>1);
                        if(PINB&(1<<RX))
                        {
                        u_buffer |= 0x80;
                        }
                }
                        else
                {
                        u_status &= (~(1<<RDR));
                }
        }
}

//*************************************************************************
//* uart_init - Subroutine for UART initialization
//*
//* DESCRIPTION
//* This routine initializes the UART. It sets the timer and enables the
//* external interrupt (receiving). To enable the UART the global interrupt
//* flag must be set (with SEI).

void uart_init(void)
{
  TCCR0 = 1<<CS00;        // Start Timer 0
  MCUCSR &= ~(1<<ISC2);   // Interrupt on falling edge
  GICR = (1<<INT2);       // Enable Timer 2 Interrupt
  DDRB |= (1<<TX);                  // Make transmit port pin as output
  PORTB |= (1<<TX);                 // default Transmit pin status is high
  PORTB |= (1<<RX);                 // Enable Pullup on the Receive Pin
}

//*************************************************************************
//* uart_transmit - Subroutine for UART transmittal
//*
//* DESCRIPTION
//* This routine initialize the UART to transmit data. The data to be sent
//* must be located in u_transmit.
//*

void uart_transmit(char data)
{
  while (u_status & (1<<BUSY));
  u_status = (1<<BUSY)|(1<<TD);
  GICR &= ~(1<<INT2);                       // Disable external interrupt
  u_bit_cnt = 0xFF;
  u_buffer = data;
  TIFR = (1<<TOIE0);
```

```
  TIMSK = (1<<TOIE0);          // Enable timer interrupt
  u_reload = 256-N;                  // Set reload value
  TCNT0 = (256-N+(14));       //Set 1st timer reload value
  PORTB &= ~(1<<TX);           //Clear PB1 to generate start bit
}
```

Code Explanation

This project is based on the ATMEL's Application Note AVR304. The Application note has a very detailed explanation of the project. The Application note is included in the CD.

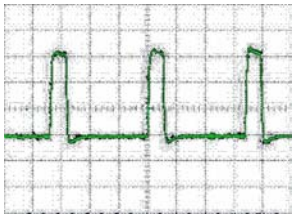
Something to try

1. Frequency Meter
   In this project, using external interrupts and the timer, a frequency meter is to be built.

2. Remote Control Decoder
   Using and IR receiver, a general purpose Remote control decoder can be built. More information on this project is to be searched and is specific to the encoder used in the Remote Control.

3. Switch de-bounce estimator
   A general purpose switch de-bounces for a while. The idea of this project is to estimate the time for which it de-bounce.

To ALLÂH belongs the sovereignty of the heavens and the earth. ALLÂH is Omnipotent. In the creation of the heavens and the earth, and the alternation of night and day, there are signs for those who possess intelligence. (Al-Imran: 189-190)



5. PWM

Pulse-Width Modulation (PWM) also called pulse-duration modulation (PDM) is a method of converting analog information using digital communication techniques. In PWM a train of pulses is produced. The width (duration) of each individual pulse varies according to the modulating analog signal. Normally, the pulse width increases as the instantaneous modulating-signal level increases (positive modulation). However, this can be reversed so that higher signal levels cause the pulse width to decrease (negative modulation).

All Timers of ATMEGA 16 has capability to produce PWM signals. The Timers of ATMEGA 16 has different PWM Modes, namely Fast PWM Mode and Phase Correct PWM Mode. These modes are discussed below

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 (OCn) is cleared on the compare match between TCNTn and OCRn, 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 uses 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 in Figure VII-1. The TCNTn 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 TCNT0 slopes represent compare matches between OCRn and TCNTn.

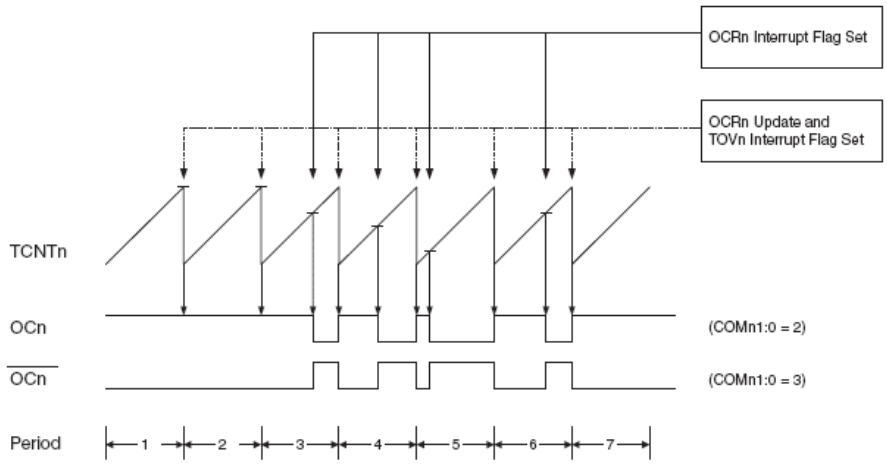


Figure VII-1: Fast PWM Mode, Timing Diagram

The Timer/Counter Overflow Flag (TOVn) 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 OCn pin. Setting the COMn1:0 bits to 2 will produce a non-inverted PWM and an inverted PWM output can be generated by setting the COMn1:0 to 3. The actual OCn 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 OCn Register at the compare match between OCRn and TCNTn, and clearing (or setting) the OCn Register at the timer clock cycle the counter is cleared (changes from MAX to BOTTOM).

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

$$f_{OCnxPWM} = \frac{f_{clk\_I/O}}{N \cdot (1 + TOP)}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024).

The extreme values for the OCRn Register represent special cases when generating a PWM waveform output in the fast PWM mode. If the OCRn is set equal to BOTTOM, the output will be a narrow spike for each MAX+1 timer clock cycle. Setting the OCRn equal to MAX will result in a constantly high or low output (depending on the polarity of the output set by the COMn1:0 bits.)

A frequency (with 50% duty cycle) waveform output in fast PWM mode can be achieved by setting OCn to toggle its logical level on each compare match (COMn1:0 = 1). The waveform generated will have a maximum frequency of fOCn = fclk_I/O/2 when OCRn is set to zero. This feature is similar to the OCn toggle in CTC mode, except the double buffer feature of the output compare unit is enabled in the fast PWM mode.

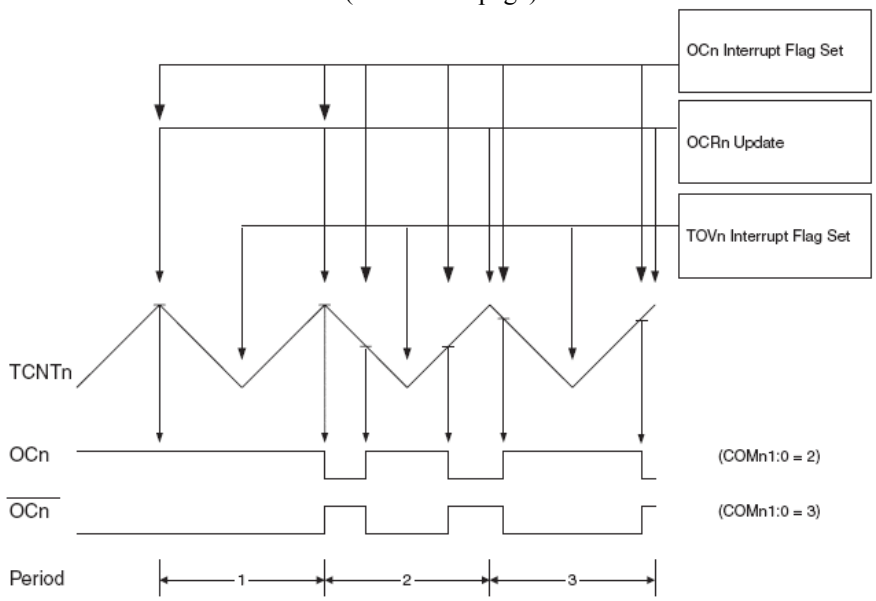
Phase Correct PWM Mode

The phase correct PWM mode (WGMn1:0 = 1) 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 non-inverting Compare Output mode, the Output Compare (OCn) is cleared on the compare match between TCNTn and OCRn while up counting, and set on the compare match while down counting. 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.

The PWM resolution for the phase correct PWM mode is fixed to eight bits. 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 TCNTn value will be equal to MAX for one timer clock cycle. The timing diagram for the phase correct PWM mode is

shown on Figure VII-2. The TCNTn 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 TCNTn slopes represent compare matches between OCRn and TCNTn.

Figure VII-2: Phase Correct PWM Mode, Timing Diagram
(on the next page)



The Timer/Counter Overflow Flag (TOVn) 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 OCn pin. Setting the COMn1:0 bits to 2 will produce a non-inverted PWM. An inverted PWM output can be generated by setting the COMn1:0 to 3. The actual OCn 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 clearing (or setting) the OCn Register at the compare match between OCRn and TCNTn when the counter increments, and setting (or clearing) the OCn Register at compare match between OCRn and TCNTn when the counter decrements. The PWM frequency for the output when using phase correct PWM can be calculated by the following equation:

$$f_{OCnxPCPWM} = \frac{f_{clk\_I/O}}{2 \cdot N \cdot TOP}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024).

The extreme values for the OCRn Register represent special cases when generating a PWM waveform output in the phase correct PWM mode. If the OCRn is set equal to BOTTOM, the output will be continuously low and if set equal to MAX the output will be continuously high for non-inverted PWM mode. For inverted PWM the output will have the opposite logic values.

At the very start of period 2 in Figure VII-2 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:

• OCRnA changes its value from MAX, like in Figure VII-2. When the OCRnA 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 OCn value at MAX must correspond to the result of an up-counting Compare Match.

• The timer starts counting from a value higher than the one in OCRnA, and for that reason misses the Compare Match and hence the OCn change that would have happened on the way up.

COM01 / COM1A1 / COM1B1 / COM21	COM00 / COM1A0 / COM1B0 / COM20	Description for Counter 0 and Counter 2	Description for Counter 1
0	0	Normal Operation, OC0 / OC1A / OC1B / OC2 Disconnected	
0	1	Reserved	WGM13:0 = 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM13:0 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC0 / OC1A / OC1B / OC2 at compare Match, Set OC0 / OC1A / OC1B / OC2 at TOP	
1	1	Set OC0 / OC1A / OC1B / OC2 at compare Match, Clear OC0 / OC1A / OC1B / OC2 at TOP	

Compare Match Output, Fast PWM Mode

COM01 / COM1A1 / COM1B1 / COM21	COM00 / COM1A0 / COM1B0 / COM20	Description for Counter 0 and Counter 2	Description for Counter 1
0	0	Normal Operation, OC0 / OC1A / OC1B / OC2 Disconnected	
0	1	Reserved	WGM13:0 = 9 or 14: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM13:0 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC0 / OC1A / OC1B / OC2 on compare Match when up counting, Set OC0 / OC1A / OC1B / OC2 on Compare Match when down counting	
1	1	Set OC0 / OC1A / OC1B / OC2 on compare Match when up counting, Clear OC0 / OC1A / OC1B / OC2 on Compare Match when down counting	

Compare Match Output, Phase Correct PWM Mode

Project 5.a, DAC from PWM

ATMEGA16 does not have a DAC inside. A DAC Chip can be added in the project if the Analog Conversion is required. However, adding a DAC chip has several disadvantages, like chip cost, extra board cost, use of I/O pins to interface the DAC chip, etc. In the cases where I/O pins are limited and Low Analog frequency is required, a cheap DAC can be built using the PWM mode of the on-chip Timer.

The timer unit generates a pulse width modulated signal whose duty cycle is based on the analog voltage required. A low pass filter is used to smooth the waveform to produce a stable analog signal. The filter components are decided according to the frequency of the PWM signal, so that the cutoff frequency of the filter is 5-10 times lower than the PWM Signal frequency. The PWM signal frequency is dependent on the Timer frequency and the sensitivity required on the analog signal.

So, on a 1MHz system that requires 6 bit (64 levels) sensitivity; the PWM Signal frequency will be 15.625 kHz. The cutoff frequency of the Low pass filter should be less than 4 kHz. So the analog system can work on a frequency of 0-4 kHz.

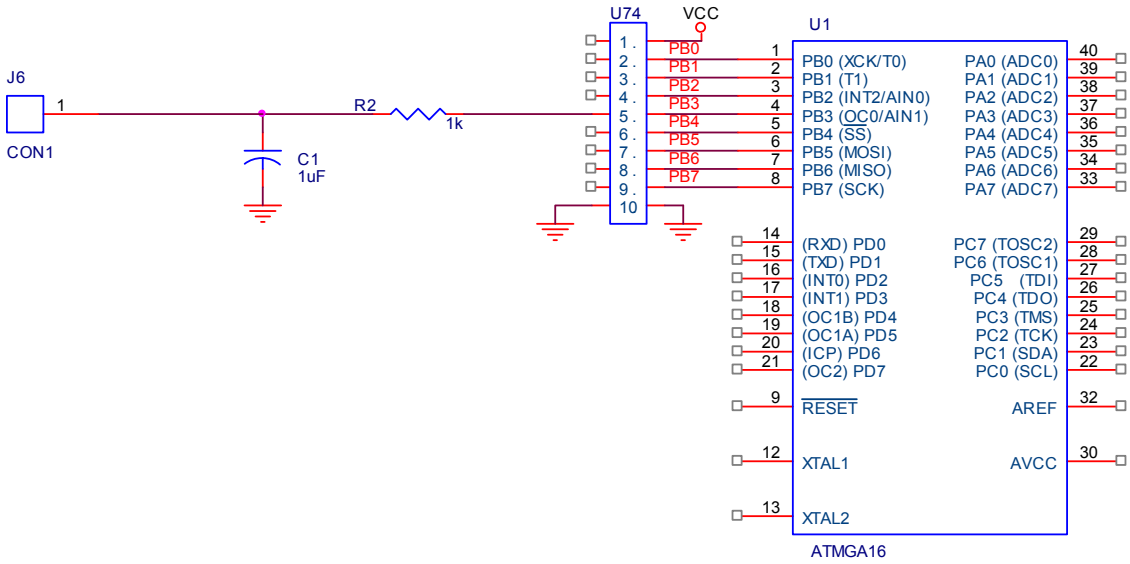


Figure VII-3: Schematic for Digital to Analog Conversion through PWM

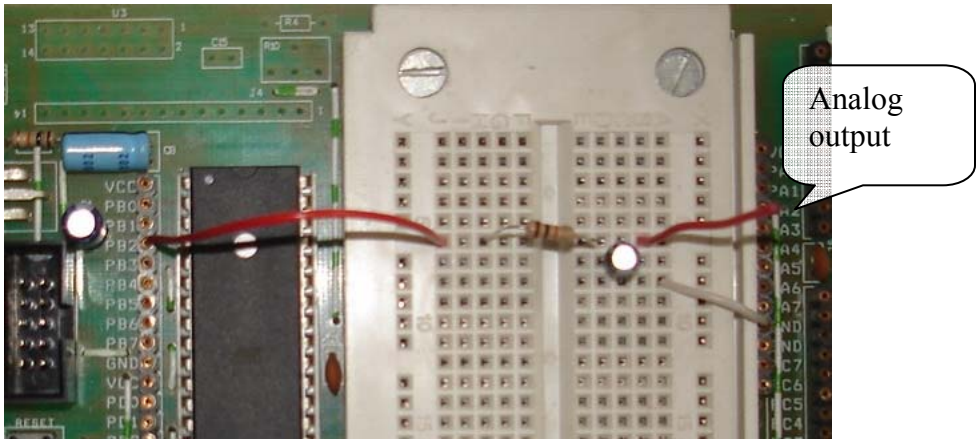


Figure VII-4: Picture showing Wiring of Digital to Analog Conversion through PWM

A Laboratory Voltmeter will be required to check the voltage produced

```
#include “avr/io.h”
#define BitVal(x) (1 << (x))

void main (void)
{
        TCCR0 = BitVal(WGM00) | BitVal(WGM01) | BitVal(CS00) | BitVal(COM00) | BitVal(COM01); // Fast PWM Mode
        OCR0 = 50;          // Output Voltage is directly dependent on the OCR0 Value
        while(1);
}
```

Code Explanation

The initialization part consists of setting up Timer 0 for producing a PWM signal for fixed frequency and variable duty cycle. OCR0 controls the duty cycle.

The main loop is not performing any functions as the frequency is produced by Timers (hardware).

Project 5.b, Voltage Converter through PWM

In some projects a negative voltage is required, like the driving voltage for an LCD. Use of a dual supply or a separate chip is not a feasible solution. Especially in the situation when a LCD driving voltage is required to produce where a variation of voltage is required and an unregulated converter can fulfill the requirement.

Unlike the Digital to analog conversion, the negative voltage converter is usually based on fixed on-time variable frequency PWM.

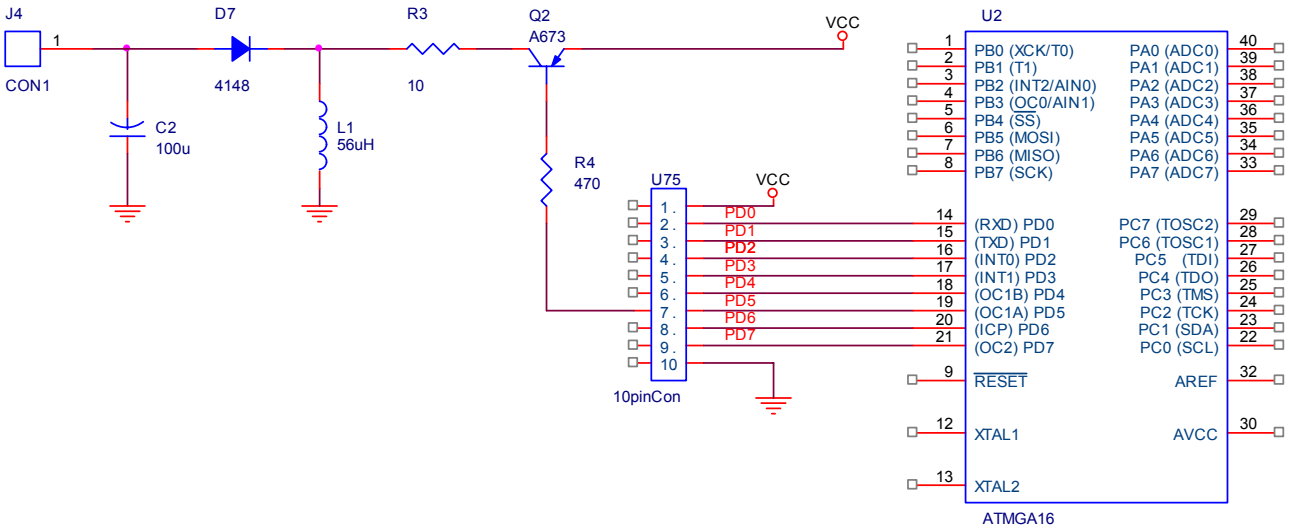


Figure VII-5: Schematic for Negative Voltage Generation

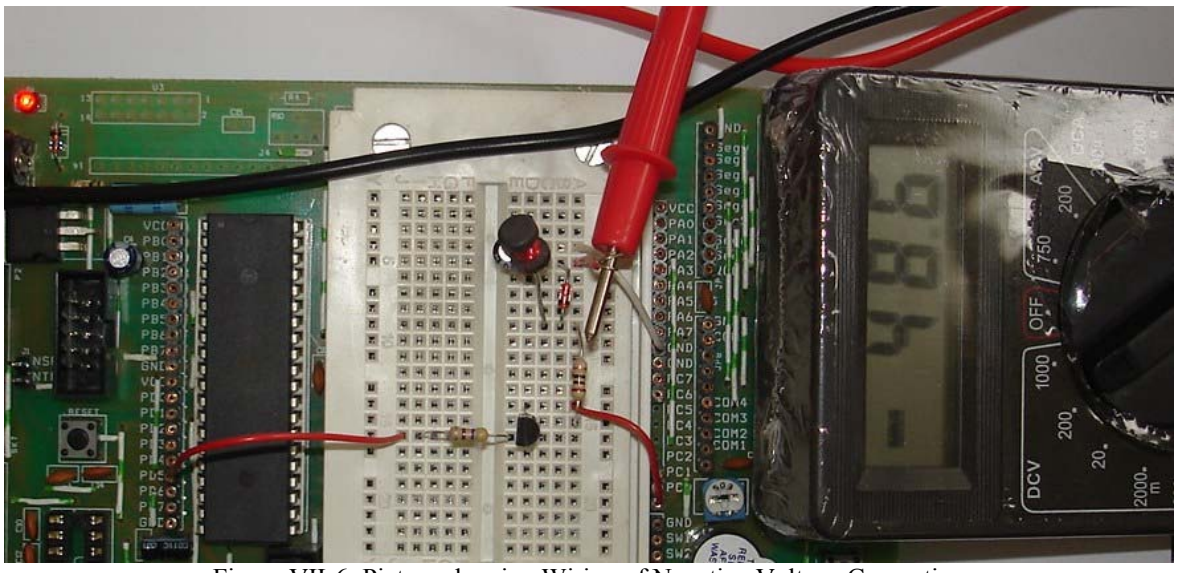


Figure VII-6: Picture showing Wiring of Negative Voltage Generation

```
#include "avr/io.h"

#define BitVal(x) (1 << (x))

void main (void)
{
        DDRD = BitVal(5);
        TCCR1A = BitVal(WGM11) |  BitVal(COM1A0) | BitVal(COM1A1); // Fast PWM Mode
        TCCR1B = BitVal(WGM12) | BitVal(WGM13) |  BitVal(CS10); // For generating Variable Frequency, fixed
                                                                 //On Time Pulse train
```

```
        OCR1A = 2;
        ICR1 = 50;          // Output Voltage is inversely dependent on the ICR1 Value
        while(1);
}
```

Code Explanation

The initialization part consists of setting up Timer 1 for producing a PWM signal for variable frequency and fixed on time. ICR1 controls the frequency, while OCR1 controls the ON-Time value.

The main loop is not performing any functions as the frequency is produced in hardware.

Something to try

1. Function generator
   A function generator that is able to produce sine, saw-tooth and square wave of any desired frequency is to be produced.

2. Voice playback
   This system should be able to play a voice that may be provided it through serial port.

3. Servo Motor Control
   Servo motors that are used in model aircrafts are controlled through a PWM signal. In this project a control mechanism for them is to be made through which they may be positioned to any desired direction.

> We created above you seven universes in layers, and we are never unaware of a single creature in them. We send down from the sky water, in exact measure, then we store it in the ground. Certainly, we can let it escape. With it, we produce for you orchards of date palms, grapes, all kinds of fruits, and various foods. Also, a tree native to Sinai produces oil, as well as relish for the eaters. (Al-Mu'minun: 17-20)

6. ADC

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.

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.

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.

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 conversions at fixed intervals. If the trigger signal still is set when the conversion completes, a new conversion will not be started. If another positive edge occurs on the trigger signal during conversion, 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.

Prescaling and Conversion Timing

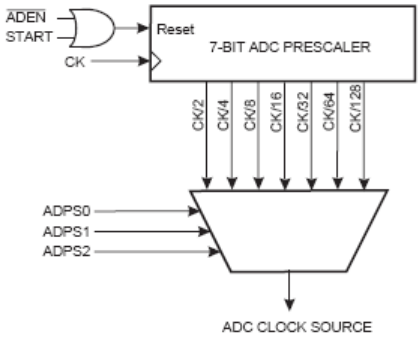


Figure VIII-1: ADC Prescaler

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.

Summary of Registers

ADC Multiplexer Selection
– ADMUX (0x07)

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	

ADC Control and Status Register

– ADCSRA (0x06)

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	

ADC Data Register

– ADCL and ADCH (0x05 and 0x04)

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	

Special Function IO Register

– SFIOR (0x30)

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR0	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	

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AVCC with external Capacitor on AREF pin
1	0	Reserved
1	1	Internal 2.56 Volt Reference with external capacitor at AREF pin

Voltage Reference Selection for 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

ADC prescalar Selection

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 Over Flow
1	0	1	Timer/Counter Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

ADC Auto Trigger Source Selection

Project 6.a, Voltmeter

The Most Basic use of an ADC is to input a slowly changing signal and convert it into appropriate Digital code. If a digital readout display is added to present the digitized voltage value, a voltmeter can be built.

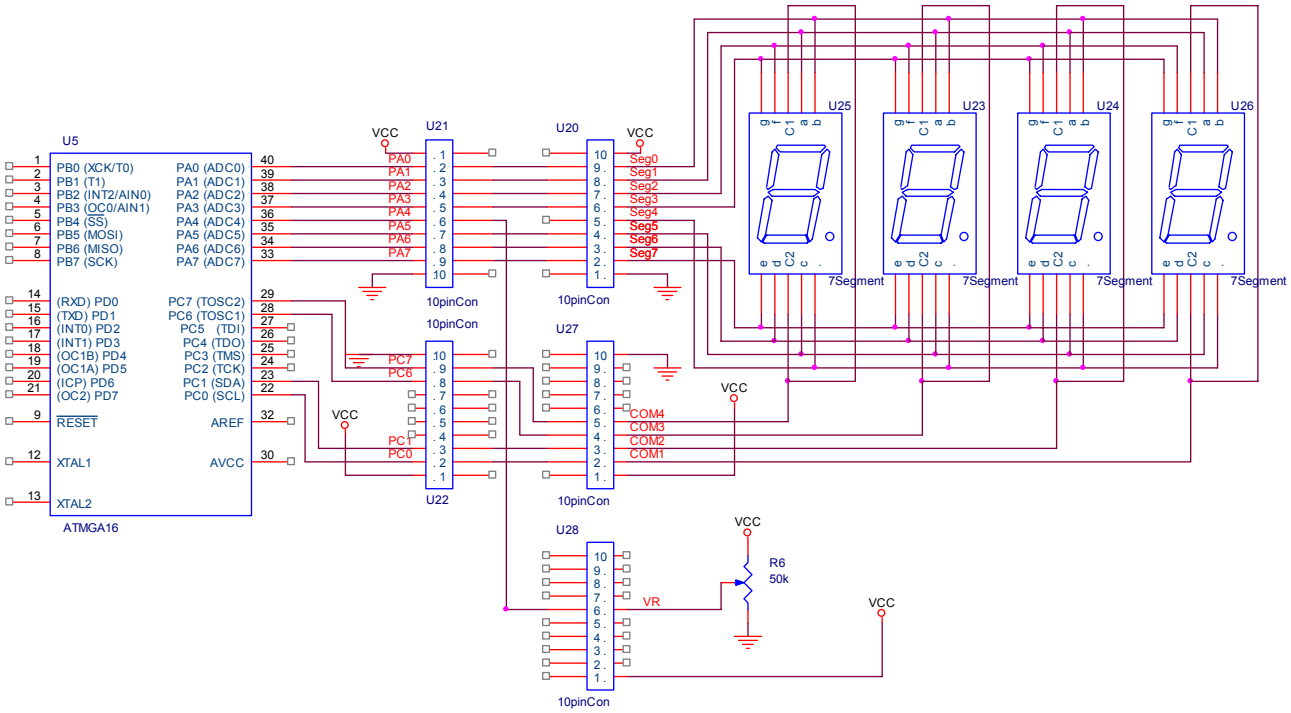


Figure VIII-2: Schematic for Voltmeter

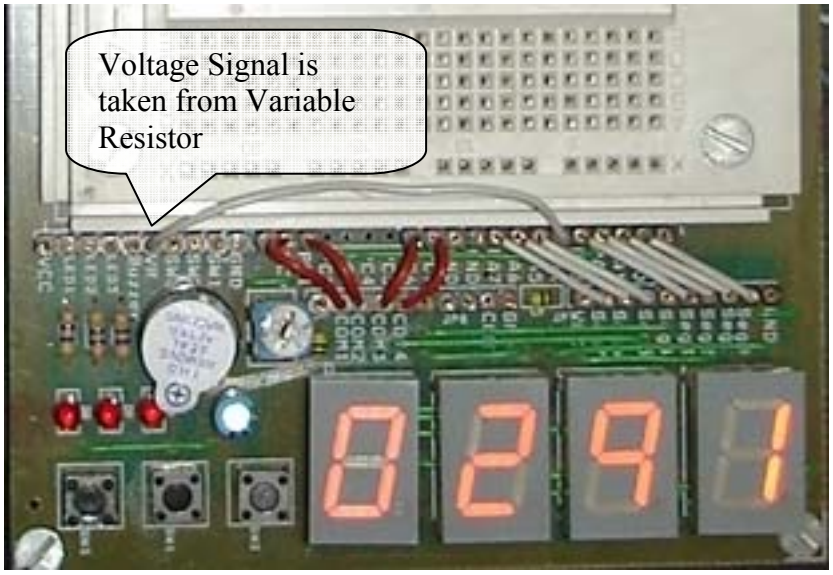


Figure VIII-3: picture showing Wiring of Voltmeter

```
#include "avr/io.h"
#include "avr/interrupt.h"

#define BitVal(n) (1 << (n))

void disp(unsigned char col, unsigned char num);

unsigned int power(unsigned char mentisa, unsigned char power);

/*Timer will be used to periodically
refresh the LEDs and the timing will
also be calculated through timers.*/
/*Following is the code to display a number (stored in variable num) at position '0'*/

unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};

unsigned long int num;

void main (void)
{
      TCCR0 = BitVal(CS00);
      TCCR1B = BitVal(CS10) | BitVal(CS11);
      DDRC = 0xC3;                        //make All Bits of PORTC as output
      DDRA = 0xEF;                        //make Bit 4 of PORTA to input
      TIMSK = BitVal(TOIE0);
      sei();
      ADMUX = BitVal(REFS0) | BitVal(MUX2);     //AVREF at 5V, Channel 4 is selected
      ADCSRA = BitVal(ADEN);
      while(1){
                  ADCSRA = BitVal(ADEN) | BitVal(ADSC) | BitVal(ADPS1);
                  while(ADCSRA & BitVal(ADSC));
                  ADCSRA |= BitVal(ADIF);
                  num = ((unsigned long)ADC*97)/200;
      }
}

unsigned char ind;

SIGNAL (SIG_OVERFLOW0)
{
      disp(ind,  (unsigned int)(num / power(10, ind)) % 10);
```

```
        ind++;
        ind &= 3;
}
```

Code Explanation

The initialization part consists of setting up Timer for servicing 7 segment LEDs, setting up Ports for driving 7 segment LEDs, Setting up ADC to use AVCC as reference, and work at ~250kHz. Timer interrupt and global flag is also set up.

The main loop consists of getting the conversion done and converting the ADC value to display the voltage value.

The Timer 0 is used to periodically refresh the 7 segment LEDs.

Project 6.b, Temperature Sensor

A temperature sensor is probably the most primitive sensor built. Usually they are based on Semiconductors (NTC Thermistor), Composites (PTC Thermistor), Metals (PT100), Bimetals (Thermocouple) or radiation based (pyrometeric sensors). Theses sensors differ in the range of temperature they work, price and signal conditioning required.

6.b.1: Using LM35

National Semiconductor has developed a chip that has the signal conditioning circuitry and the temperature sensing element on a small TO92 package. The chip gives quite accurate temperature reading for 0-100°C with an accuracy of 0.5°C. A 10 bit ADC is sufficient to digitize the output produced by LM34 with 0.5°C accuracy.

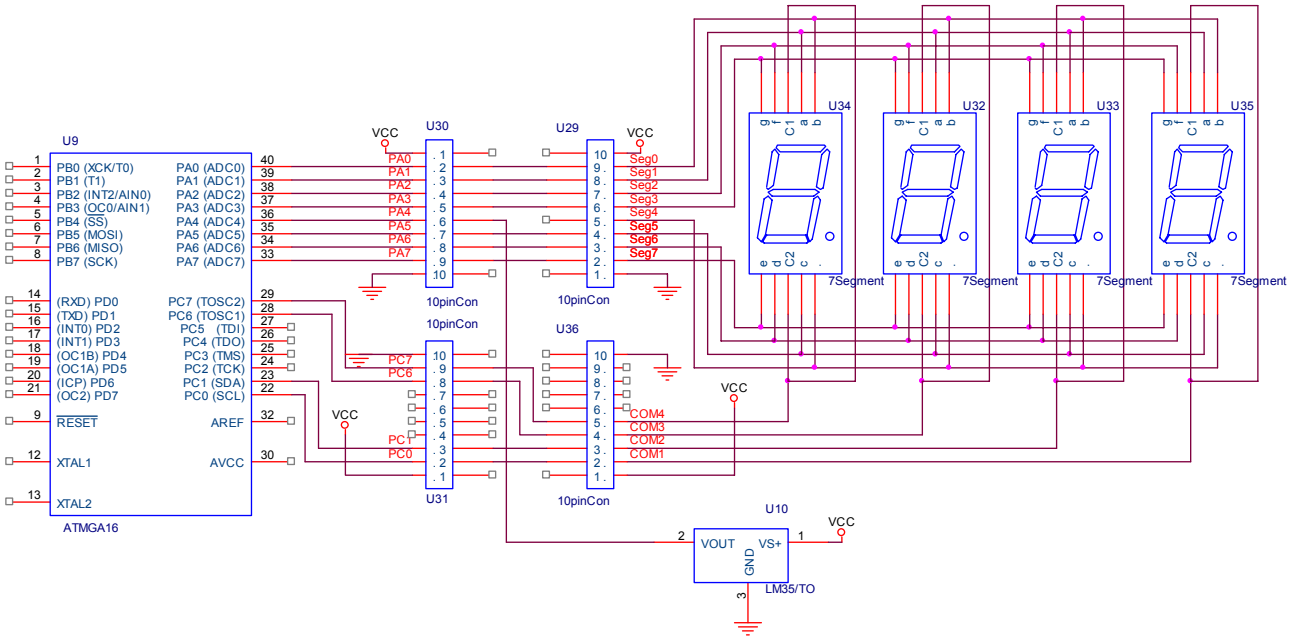


Figure VIII-4: Schematic for Temperature Sensor using LM35

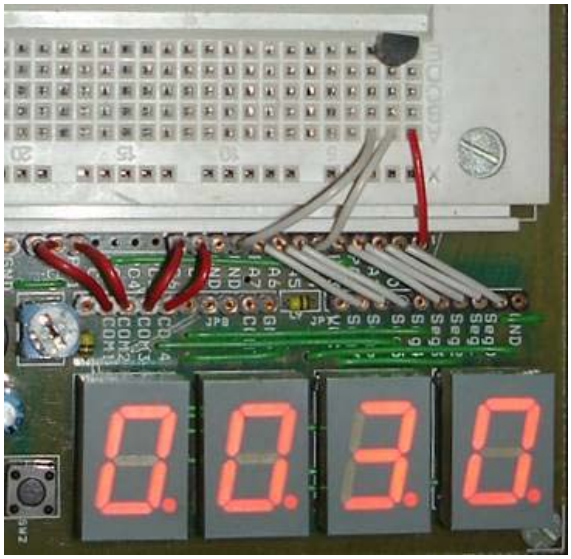


Figure VIII-5: Picture showing Wiring of Temperature Sense using LM35

Circuit of Project 2.a. will be used to display the Temperature Value

```
#include "avr/io.h"
#include "avr/interrupt.h"

#define BitVal(n) (1 << (n))

void disp(unsigned char col, unsigned char num);

unsigned int power(unsigned char mentisa, unsigned char power);

/*Timer will be used to periodically
refresh the LEDs and the timing will
also be calculated through timers.*/
/*Following is the code to display a number (stored in variable num) at position '0'*/

unsigned char image[] = {0xE7, 0x21, 0xCB, 0x6B, 0x2D, 0x6E, 0xEE, 0x23, 0xEF, 0x2F, 0xAF, 0xEC, 0xC6, 0xE9,
0xCE, 0x8E};

unsigned long int num;

void main (void)
{
	TCCR0 = BitVal(CS00);
	TCCR1B = BitVal(CS10) | BitVal(CS11);
	DDRC = 0xC3;				//make All Bits of PORTC as output
	DDRA = 0xEF;				//make Bit 4 of PORTA to input
	TIMSK = BitVal(TOIE0);
	sei();
	ADMUX = BitVal(REFS0) | BitVal(MUX2);	//AVREF at 5V, Channel 4 is selected
	ADCSRA = BitVal(ADEN);
	while(1){
		ADCSRA = BitVal(ADEN) | BitVal(ADSC) | BitVal(ADPS1);
		while(ADCSRA & BitVal(ADSC));
		ADCSRA |= BitVal(ADIF);
		num = ((unsigned long)ADC*500+512)/1024; //For LM35 10mV/C
	}
}

unsigned char ind;

SIGNAL (SIG_OVERFLOW0)
{
	disp(ind,  (unsigned int)(num / power(10, ind)) % 10);
```

```
        ind++;
        ind &= 3;
}
```

Code Explanation

The initialization part consists of setting up Timer for servicing 7 segment LEDs, setting up Ports for driving 7 segment LEDs, Setting up ADC to use AVCC as reference, and work at ~250kHz. Timer interrupt and global flag is also set up.

The main loop consists of getting the conversion done and converting the ADC value for the display.

The Timer 0 is used to periodically refresh the 7 segment LEDs.

6.b.2: Using a Thermistor

Although it is easy to interface a chip like LM34, a Thermistor may be required to employ if a broader temperature range is required.

Thermistors are nonlinear devices and are difficult to interface. The nonlinearity of Thermistor increases many folds if it is used in a voltage divider with a fixed value resistor. To overcome this problem, usually a constant current source is used to bias a Thermistor to get a more linear temperature value.

In the following schematic, Q_1, R_1, R_5, D_{23} and D_{24} form the current source. The current through Thermistor can be controlled by R_1 with the following relation

$$I = \frac{2 * V_{Diode} - V_{BE}}{R_1}$$

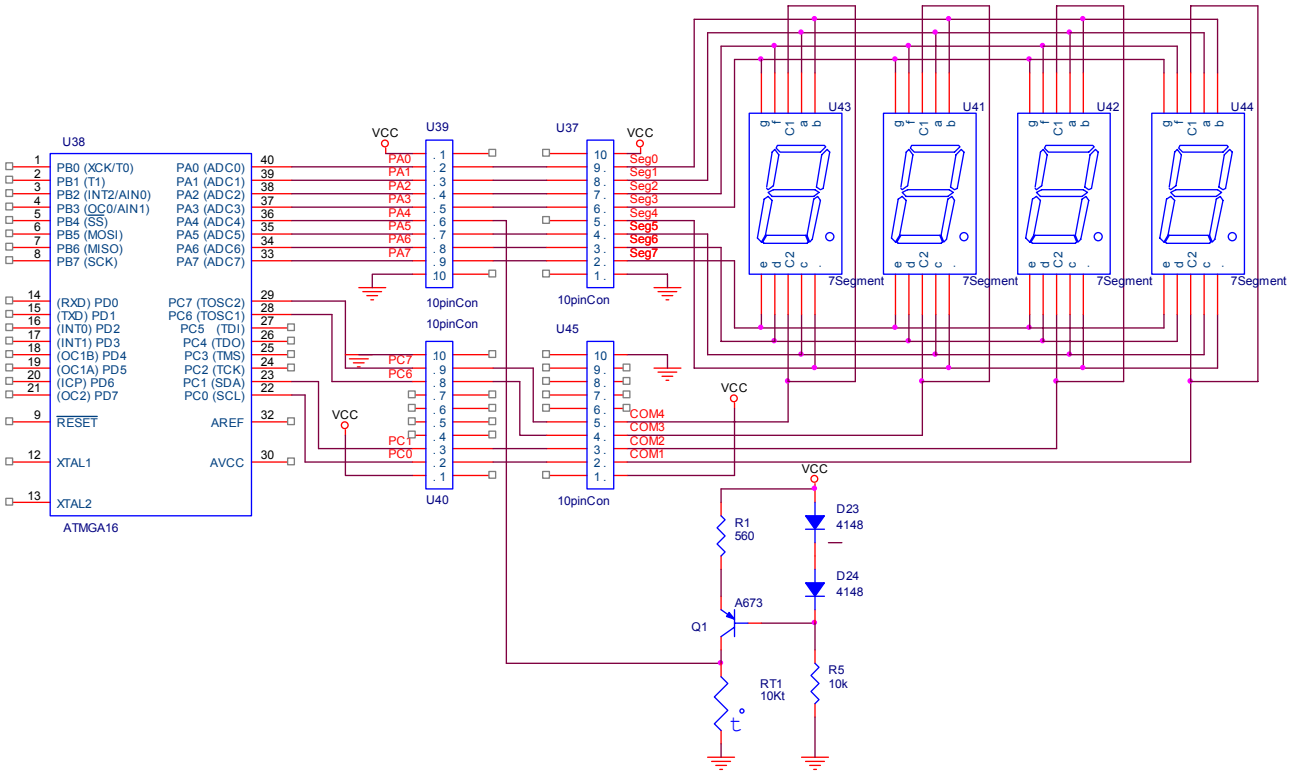


Figure VIII-6: Schematic for Temperature sensing through Thermistor

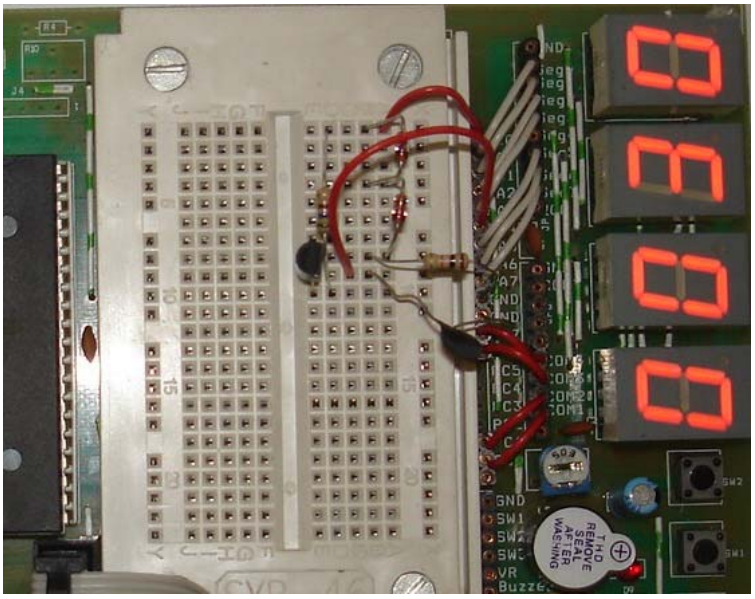


Figure VIII-7: Picture showing Wiring for Temperature sensing through Thermistor

Software Code of Project 2.a. will be used to display the Temperature Value, the only change to be done in the code is the conversion from digitized value to actual temperature value.

```
num = 2760/ADC;          //Thermistor
```

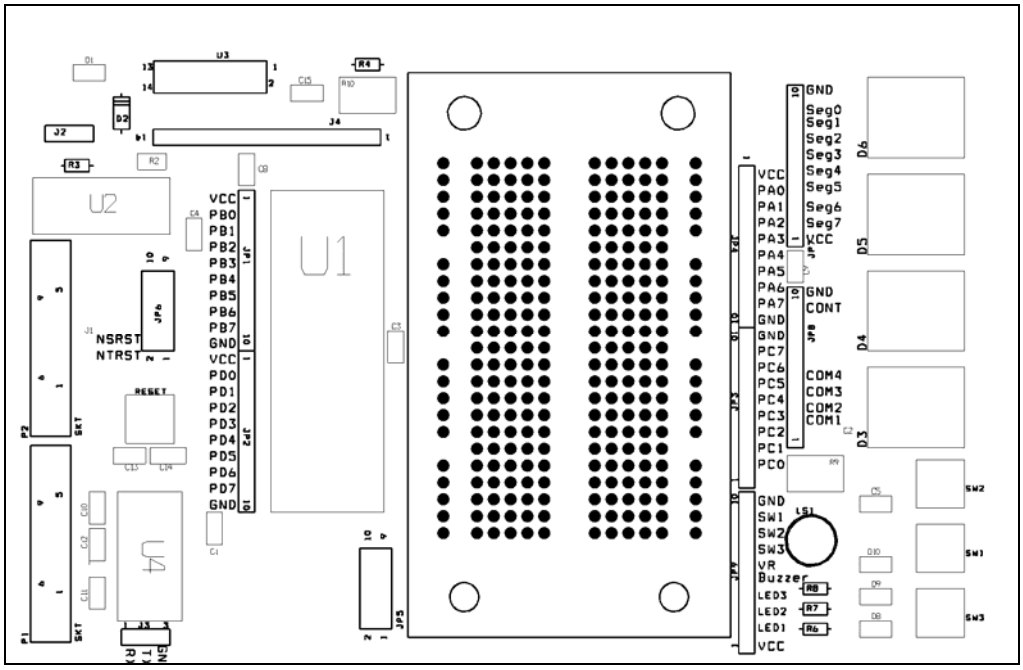
Something to try

1. Lux meter
   Using a LDR, Lux meter can be formed.

2. Variable Voltage Reference
   By Combing the Project 5a and 6a, a variable voltage reference can be made.

3. Tilt Sensor
   By attaching a stick to a potentiometer that has a mass on the other end, a tilt sensor can be made. This tilt sensor can be used to level articles like pictures, or it can be used in vehicles to estimate the inclination angles.

Appendix-A: Layouts and Schematics



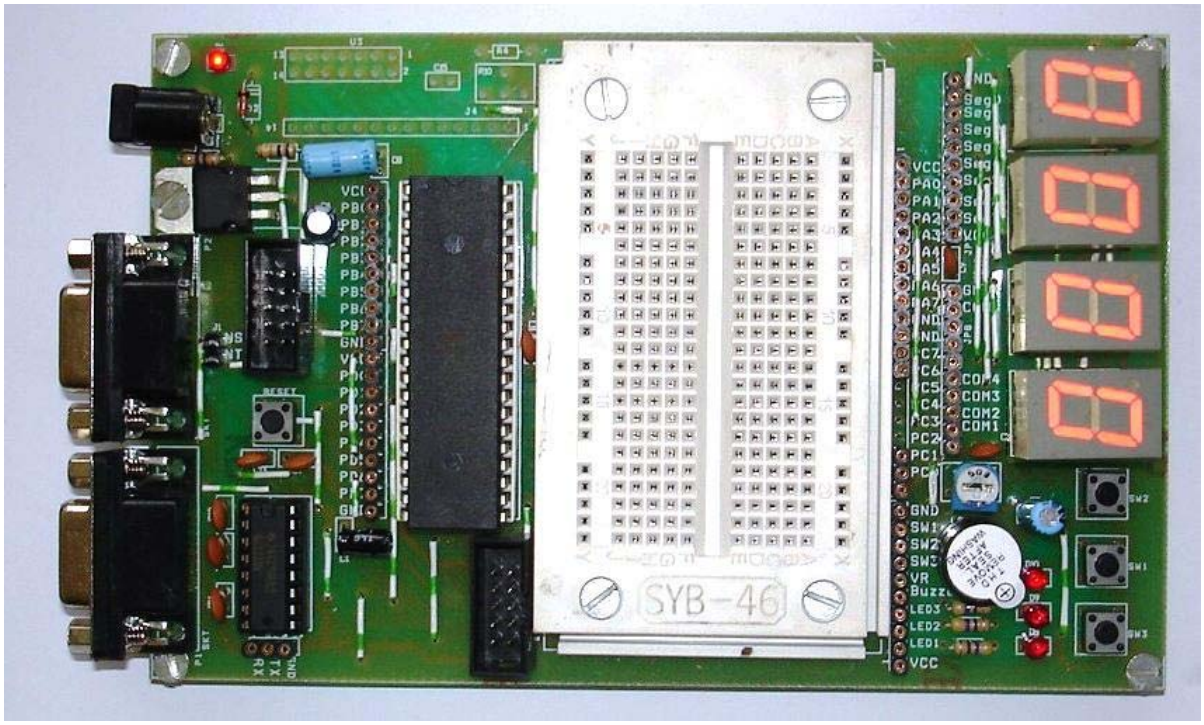
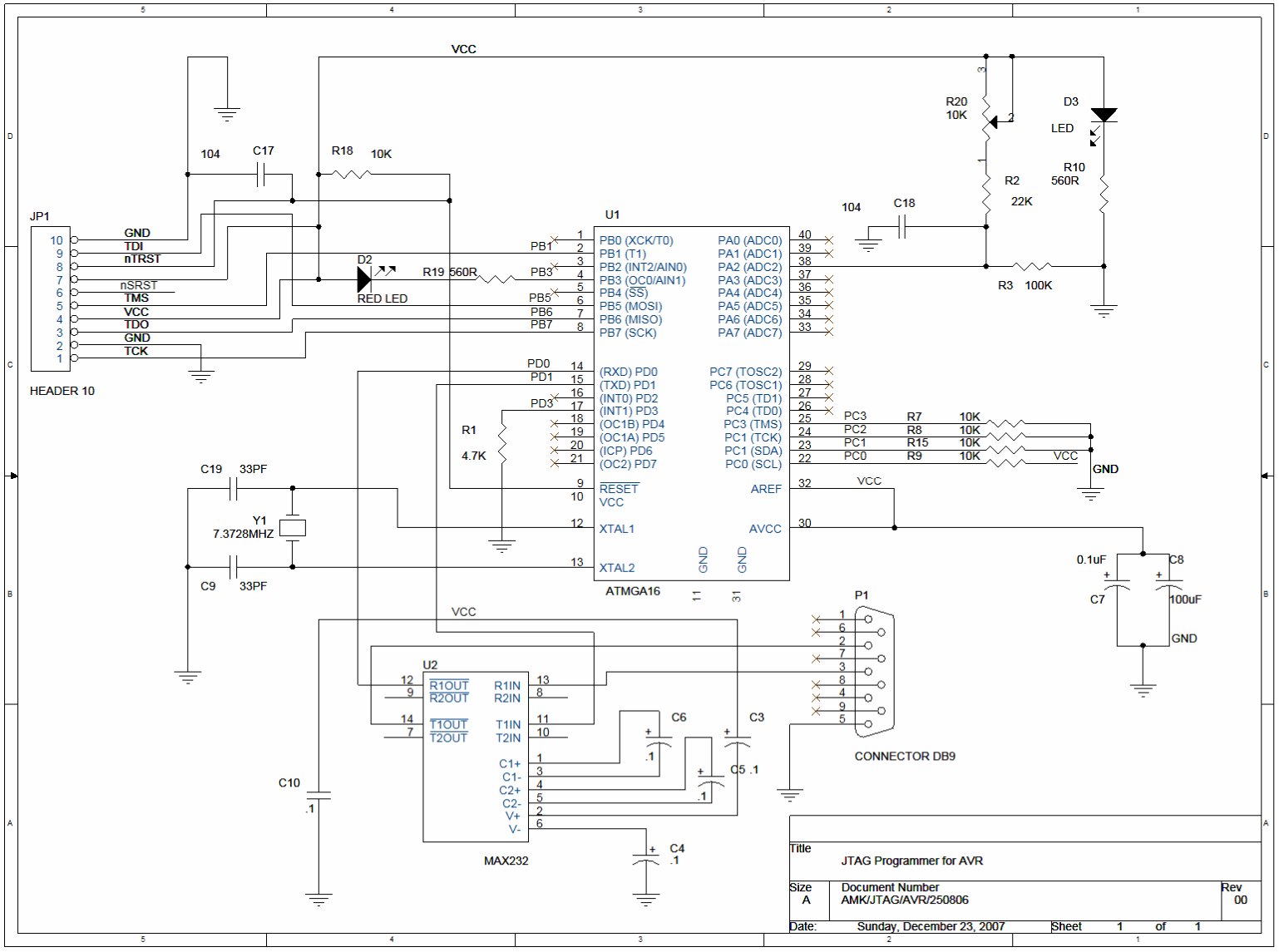
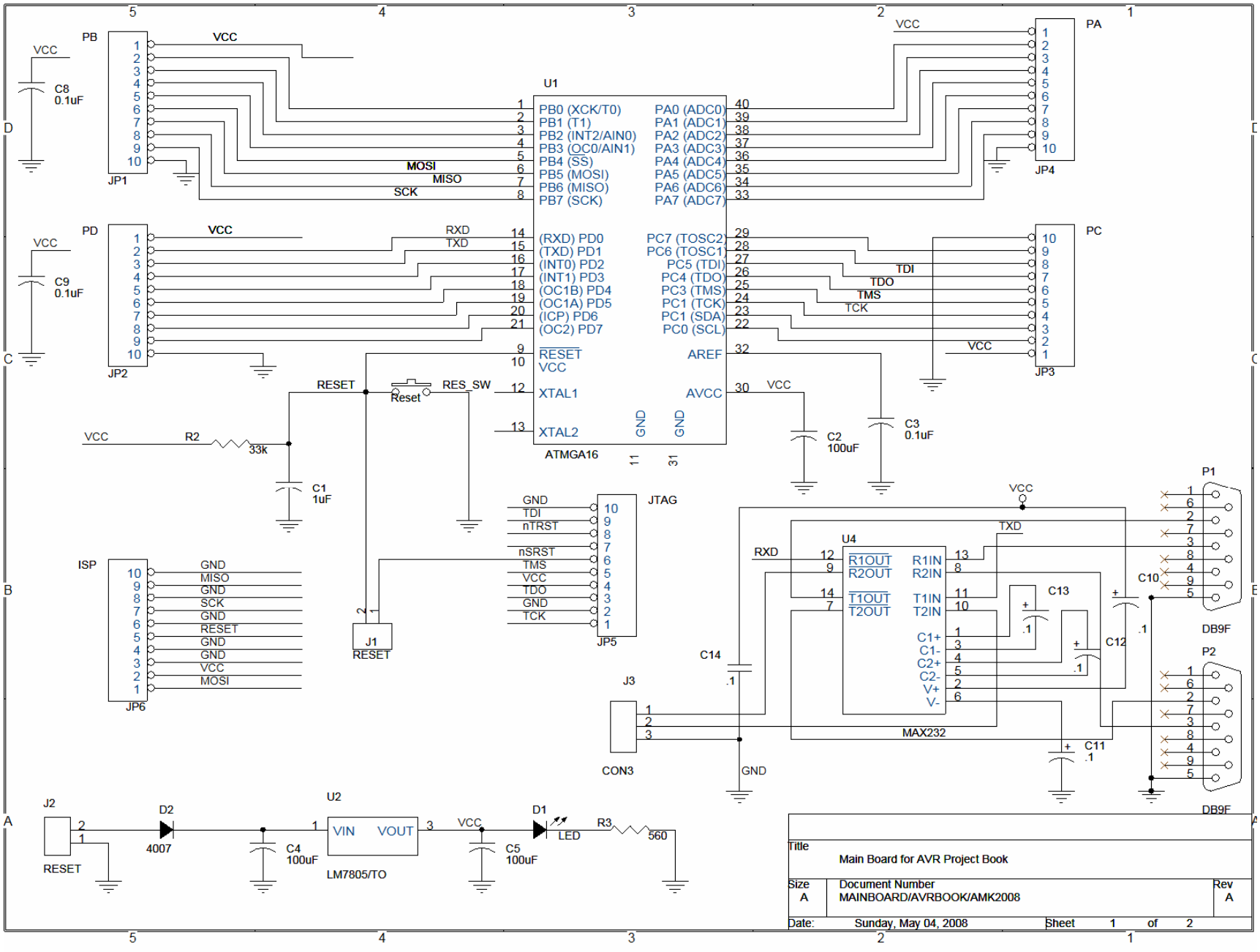


Figure IX-8: LAYOUTs of Project Board PCB

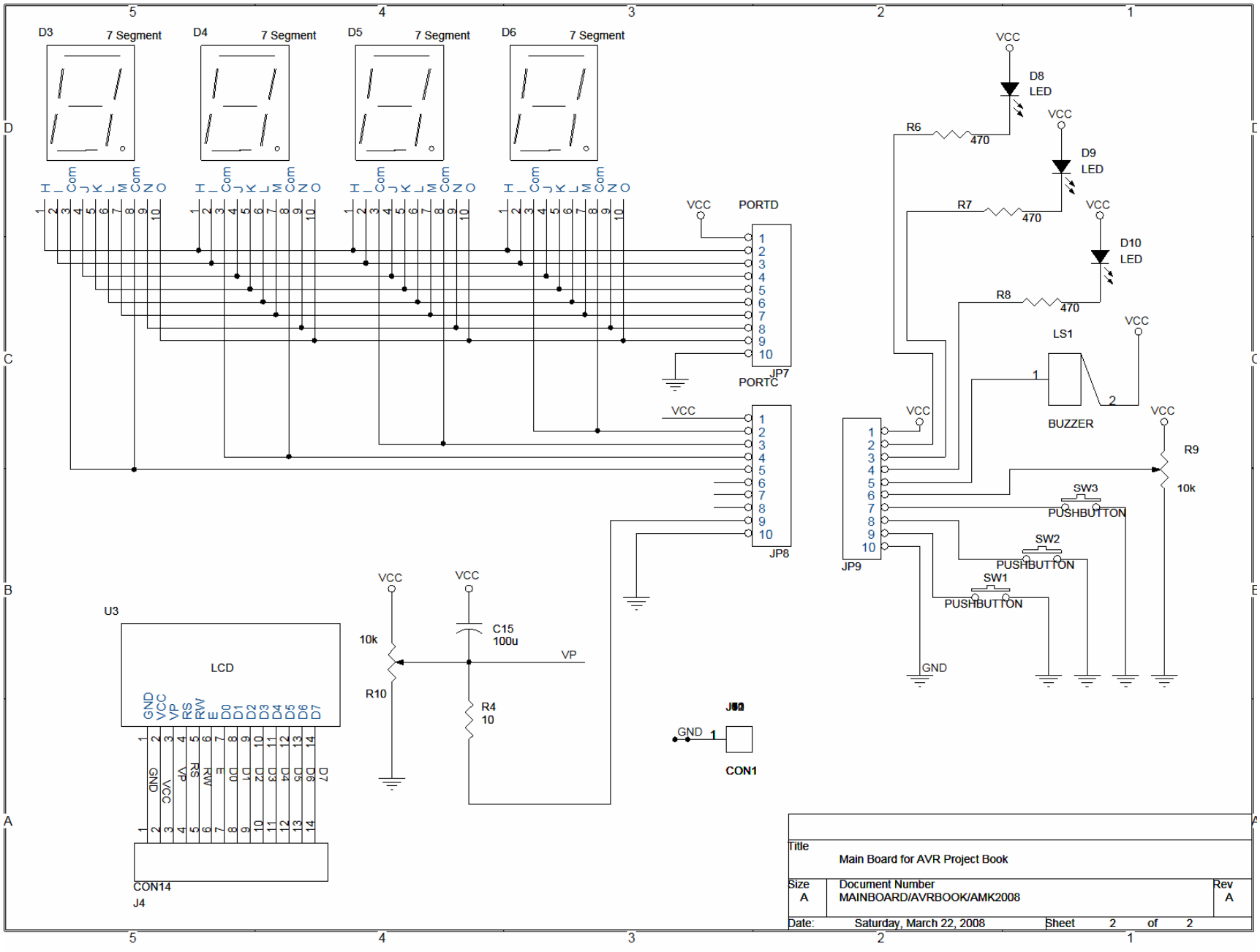
Schematics


VCC
JP1
HEADER 10
GND
TDI
nTRST
nSRST
TMS
VCC
TDO
GND
TCK
104
C17
R18
10K
D2
RED LED
R19 560R
U1
ATMGA16
PB0 (XCK/T0)
PB1 (T1)
PB2 (INT2/AIN0)
PB3 (OC0/AIN1)
PB4 (SS)
PB5 (MOSI)
PB6 (MISO)
PB7 (SCK)
PA0 (ADC0)
PA1 (ADC1)
PA2 (ADC2)
PA3 (ADC3)
PA4 (ADC4)
PA5 (ADC5)
PA6 (ADC6)
PA7 (ADC7)
(RXD) PD0
(TXD) PD1
(INT0) PD2
(INT1) PD3
(OC1B) PD4
(OC1A) PD5
(ICP) PD6
(OC2) PD7
PC7 (TOSC2)
PC6 (TOSC1)
PC5 (TD1)
PC4 (TD0)
PC3 (TMS)
PC1 (TCK)
PC1 (SDA)
PC0 (SCL)
RESET
VCC
XTAL1
XTAL2
AREF
AVCC
GND
R1
4.7K
R20
10K
D3
LED
R10
560R
R2
22K
R3 100K
104
C18
PC3 R7 10K
PC2 R8 10K
PC1 R15 10K
PC0 R9 10K
C19 33PF
Y1
7.3728MHZ
C9 33PF
0.1uF
C7
C8
100uF
U2
MAX232
R1OUT
R2OUT
T1OUT
T2OUT
R1IN
R2IN
T1IN
T2IN
C1+
C1-
C2+
C2-
V+
V-
C6
C3
C5 .1
C10
.1
C4
P1
CONNECTOR DB9
Title
JTAG Programmer for AVR
Size
A
Document Number
AMK/JTAG/AVR/250806
Rev
00
Date:
Sunday, December 23, 2007
Sheet 1 of 1

Appendix-A: Layouts and Schematics



Appendix-A: Layouts and Schematics



Package Content

The package of AVR Project Book contains the following items

1. AVR Project Board
2. AVR JTAG Board
3. 10 pin Ribbon Cable
4. Serial Cable for JTAG Board
5. Software CD
6. This manual
7. Components for the project
    a. Resistors 10, 470, 560, 10k. One ea.
    b. Capacitors 1uF, 10uF. One ea.
    c. Inductor 58uH. One pc.
    d. Diode 1N4148. Two pcs.
    e. Thermistor. One pc.
    f. Temperature Sensor LM35. One pc.
    g. Transistor A673. One pc.

Before starting work on the projects, check the content of the package to see if the content matches the list provided above.

About the Author

Abdul Maalik Khan has graduated from Institute of Industrial Electronics Engineering, NED University Karachi in 1995. After graduation, he has spent some time in industry, working as free lancer. In 1999 he has joined one of the Public Sector Research Institution. In 2002 he has secured Ministry of Science and Technology Scholarship for MS. He then joined EME College, NUST to

pursue for MS Degree. During studies he has selected for the split program under which he went to Georgia Institute of Technology and completed MS in 2004. His fields of interest are Embedded Systems Programming, Signal Processing and Image Processing. In 2005 he started some ground work on Core Technologies Development. The idea of Core Technology Development is to work in such fields of Embedded Systems that make its bases; like Firmware Development, Kernel Development, Interface Hardware Development, Driver Development, Library Development, etc. In 2008 he started to compile his work and wanted to publish it under name of DigiSoft.

During his studies and job tenure he has worked on Intel's 8051 series of Microcontrollers, ATMEL's AVR Series of Microcontrollers, Infineon 80C165, Super Hitachi Microcontroller, 80x86 series of Microprocessors, ARM series of Microcontrollers and Microchip's PIC Microcontrollers. He also has expertise of PCB Designing on OrCAD and mechanical drawings on AutoCAD. He also has worked on porting some popular Embedded OS on to Microcontrollers.

He also has advised some students to do their Research work for the fulfillment of degree requirement for MS in the field of Embedded System Programming.

Items required to perform projects

1. AVR Project Book with AVR Project Board, AVR-JTAG and supplied components and cables.
2. PC.
3. A general purpose 9V wall adapter.
4. Lab Multi-meter.
5. Soldering Iron and Wire.
6. Screw Drivers, pliers and wire stripper.

Further Information

The author can be reached through amk@digisoft.com.pk

For latest information about the book and new books, visit www.DigiSoft.com.pk

For discussion about the DigiSoft products join "DigiSoft_Support" at groups.yahoo.com

ATMEL Data Sheets and AVR studio can be viewed from www.atmel.com

Other Manufacturer's Data sheets can be viewed from www.alldatasheet.com

GCC and other gnu related topics can be taken from www.gnu.org

Appendix-B: Frequently Asked Questions

- How should I Run a project given in the book?
  First you should make sure that you have all the required items necessary, as given in **Items required to perform projects**
  Then setup the Environment for running projects, as given in under the heading of **Projects**
  Browse to the Folder where the project files of your intended project are present.
  Double click the project file and it should open in AVR Studio.
  Compile the project.
  Now you have the option to program the compiled project and run it, or debug it. Both options are discussed in DS-AVR-JTAG user guide.
  You may also need to connect wires to different connectors on the Project Board as required in the project.
  Be sure to register your Book by sending an email to the author. Also subscribe to the DigiSoft group to receive future updates of the Book.

- How can I setup a new project?
  To setup an AVR project that uses AVR GCC, Consult AVR studio user guide included in the CD.
  For a project that is based on Assembly, consult user guide provided by ATMEL.

- What other guides are included?
  The CD Contains following other user guides;
  1. AVR Project Book User Guide.
  2. AVR JTAG User Guide.
  3. AVR Studio User Guide.

  Help Files, user guides and Application notes from different sources are also included.

- Can I use any other controller in the AVR Project Board?
  ATMEGA 32 or ATMEGA 323 can also be used in AVR project Board.

- Why there is some un-stuffed area on AVR Project Board?
  The un-stuffed are on AVR Project Board contains a layout to interface a Text LCD. It is left to the user if he wishes to use it.

- Can I use external Crystal to clock the chip?
  Unfortunately the board does not support it, but you can still connect an external crystal, for that you have to solder it directly on the pins.

- I've just bought the book, but my Board is not working, what should I do?
  Email the author about your problem.

- Is there any warranty on this board?
  The boards are covered under limited warranty, Email the author if you found some defects in your board.

- I have an idea of a project that can be built on the Project Board, can I get help from somewhere to write code?
  You can email your idea to the author.

- I have found a bug in your code / I have found some wrong text in the book?
  Kindly email your finding to the author. Your help will be highly appreciated.

- The explanations for projects are not adequate?
  Future editions of the book will contain more explanations. If you have registered to the email group, you will receive a soft copy of the future editions of the book.

- I was using my board and it was working fine, but now it is not working, what should I do?
  First check if it is AVR Project Board that is faulty or the AVR-JTAG. If the AVR Studio is unable to connect to AVR-JTAG it is likely that JTAG is faulty. If your AVR-JTAG is not under warranty, you may need to buy a new AVR JTAG. If the AVR JTAG is unable to program the ATMEGA 16 on the AVR Project Board, you may need to replace the controller on the AVR Project Board.

- I have a question that is not answered here, what should I do?
  If your question is regarding the AVR Project Book, send your question to the DigiSoft mailing group. Include as much information as possible.
  If your question is regarding to the ATMEGA 16 or AVR GCC, send it to DigiSoft mailing group or AVR GCC mailing Group.

Appendix-C: Glossary of Terms

ADC (Analog to Digital Converter) is a Peripheral circuitry of ATMEGA16, that converts an Analog signal into its Digital Representation.

Absolute Section
A section with a fixed Access Memory Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register

Address
Value that identifies a location in memory.

Alphabetic Character
Alphabetic characters are those characters that are letters of the arabic alphabet

Alphanumeric
Alphanumeric characters are comprised of alphabetic characters and decimal digits

Anonymous Structure
An unnamed structure that is a member of a C union. The members of an anonymous structure may be accessed as if they were members of the enclosing union.

ANSI
American National Standards Institute is an organization responsible for formulating and approving standards in the United States.

Application
A set of software and hardware that may be controlled by a microcontroller.

Archive
A collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver to combine the object files into one library file. A library can be linked with object modules and other libraries to create executable code.

Archiver
A tool that creates and manipulates libraries.

ASCII
American Standard Code for Information Interchange is a character set encoding that uses 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembler
A language tool that translates assembly language source code into machine code.

Assembly Language
A programming language that describes binary machine code in a symbolic form.

Assigned Section
A section which has been assigned to a target memory block in the linker command file.

Asynchronously
Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution, usually manually.

Asynchronous Stimulus
Data generated to simulate external inputs to a simulator device.

Attribute
Characteristics of variables or functions in a C program which are used to describe machine-specific properties.

Attribute, Section
Characteristics of sections, such as “executable”, “readonly”, or “data” that can be specified as flags in the assembler **.section** directive.

Binary
The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then $2^2 = 4$, etc.

Breakpoint, Hardware
An event whose execution will cause a halt.

Breakpoint, Software
An address where execution of the firmware will halt. Usually achieved by a special break instruction.

Build
Compile and link all the source files for an application.

C
A general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators.

Calibration Memory
A special function register or registers used to hold values for calibration of a microcontroller on-board RC oscillator or other device peripherals.

Central Processing Unit
The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic

logic unit address bus, the data memory address bus, and accesses to the stack.

COFF
Common Object File Format. An object file of this format contains machine code, debugging and other information.

Command Line Interface
A means of communication between a program and its user based solely on textual input and output.

Compiler
A program that translates a source file written in a high-level language into machine code.

Conditional Assembly
Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

Conditional Compilation
The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

Configuration Bits
Special-purpose bits programmed to set microcontroller modes of operation. A Configuration bit may or may not be preprogrammed.

Control Directives
Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

CPU
See Central Processing Unit.

Cross Reference File
A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

Data Directives
Data directives are those that control the assembler's allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory
Internal Static Memory, where the temporary data is stored.

Deprecated Features
Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

Device Programmer
A tool used to program electrically programmable semiconductor devices such as microcontrollers.

Digital Signal Controller
A microcontroller device with digital signal processing capability.

Digital Signal Processing
The computer manipulation of digital signals, commonly analog signals been converted to digital form

Digital Signal Processor
A microprocessor that is designed for use in digital signal processing.

Directives
Statements in source code that provide control of the language tool's operation.

Download
Download is the process of sending data from a host to another device, such as an emulator, programmer or target board.

DSC
See Digital Signal Controller.

DSP
See Digital Signal Processor.

DWARF
Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

EEPROM
Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

ELF
Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in with DWARF. ELF/DWARF provides better debugging of optimized code than COFF.

Emulation
The process of executing software loaded into emulation memory as if it were firmware residing on a microcontroller device.

Emulation Memory
Program memory contained within the emulator.

Emulator
Hardware that performs emulation.

Endianess
The ordering of bytes in a multi-byte object.

Environment – IDE
The particular layout of the desktop for application development.

Epilogue
A portion of compiler-generated code that is responsible for de-allocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

EPROM
Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Error File
A file containing error messages and diagnostics generated by a language tool.

Errors
Errors report problems that make it impossible to continue processing your program. When possible, errors identify the source file name and line number where the problem is apparent.

Executable Code
Software that is ready to be loaded for execution.

Export
Send data out of the IDE in a standardized format.

Expressions
Combinations of constants and/or symbols separated by arithmetic or logical operators. literal offset addressing.

External Label
A label that has external linkage.

External Linkage
A function or variable has external linkage if it can be referenced from outside the module in which it is defined.

External Symbol
A symbol for an identifier which has external linkage. This may be a reference or a definition.

External Symbol Resolution
A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to resolve all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

External Input Line
An external input signal logic probe line

External RAM
Off-chip Read/Write memory.

Fatal Error
An error that will halt compilation immediately. No further messages will be produced.

File Registers
On-chip data memory, including General Purpose

Registers Filter
Determine by selection what data is included/excluded in a trace display or data file.

Flash
A type of EEPROM where data is written or erased in blocks instead of bytes.

Halt
A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

Heap
An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

Hex Code
Executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

Hex File
An ASCII file containing hexadecimal addresses and values device.

Hexadecimal
The base 16 numbering system that uses the digits 0-9 plus the letters A-F represent hexadecimal digits with values of $16^2 = 256$, etc.

High Level Language
A language for writing programs that is further removed from the processor than assembly.

ICSP
In-Circuit Serial Programming. A method of programming embedded devices using serial communication and a minimum number of device pins.

IDE
Integrated Development Environment.

Identifier
A function or variable name.

IEEE
Institute of Electrical and Electronics Engineers.

IIEE
Institute of Industrial Electronics Engineering.

Import
Bring data into the IDE from an outside source, such as from a hex file.

Initialized Data
Data which is defined with an initial value. In C, int myVar=5; defines a variable which will reside in an initialized data section.

Instruction Set
The collection of machine language instructions that a particular processor understands.

Instructions
A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

Internal Linkage
A function or variable has internal linkage if it can not be accessed from outside the module in which it is defined.

International Organization for Standardization
An organization that sets standards in many businesses and technologies, including computing and communications.

Interrupt
A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine execution of the application resumes.

Interrupt Handler
A routine that processes special code when an interrupt occurs.

Interrupt Request
An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

Interrupt Service Routine
A function that handles an interrupt.

Interrupt Service Routine
User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

Interrupt Vector
Address of an interrupt service routine or interrupt handler.

IRQ
See Interrupt Request.

ISO
See International Organization for Standardization.

ISR
See Interrupt Service Routine.

L-value
An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

Latency
The time between an event and its response.

Librarian
See Archiver.

Library
See Archive.

Linker
A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files
Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives
Listing directives are those directives that control the assembler listing file format. They allow the specification of titles, pagination and other listing control.

Listing File
A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, assembler directive, or macro encountered in a source file.

Little Endian
A data ordering scheme for multi-byte data whereby the least significant byte is stored at the lower addresses.

Local Label
A local label is one that is defined inside a macro with the LOCAL directive. These labels are particular to a given instance of a macro's instantiation. In other words, the symbols and labels that are declared as local are no longer accessible after the ENDM macro is encountered.

LVDS
Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed per second data transmission over wire.

Machine Code
The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions with data). The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language
A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

Macro
Macro instruction. An instruction that represents a sequence of instructions in abbreviated form.

Macro Directives
Directives that control the execution and data allocation within macro body definitions.

Make Project
A command that rebuilds an application, recompiling only those source files that have changed since the last complete compilation.

MCU
Microcontroller Unit. An abbreviation for microcontroller. Also uC.

Memory Models
A representation of the memory available to the application.

Memory Models
Versions of libraries and/or precompiled object files based on a device's memory structure.

Memory Model
A description that specifies the size of pointers that point to program memory.

Message
Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

Microcontroller
A highly integrated chip that contains a CPU, RAM, program memory, I/O ports and timers.

MRU
Most Recently Used. Refers to files and windows available to be selected from IDE main pull down menus.

Nesting Depth
The maximum level to which macros can include other macros.

Non-Volatile Storage
A storage device whose contents are preserved when its power is off.

NOP
No Operation. An instruction that has no effect when executed except to advance the program counter.

Object Code
The machine code generated by an assembler or compiler.

Object File
A file containing machine code and possibly debug information. It may be immediately executable or it may be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

Object File Directives
Directives that are used only when creating an object file.

Octal
The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2 = 64$, etc.

Opcodes
Operational Codes. *See* Mnemonics.

Operators
Symbols, like the plus sign '+' and the minus sign '-', that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

OTP
One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

PC
Personal Computer or Program Counter.

PC Host
Any PC running a supported Windows operating system.

Persistent Data
Data that is never cleared or initialized. Its intended use is so that an application can preserve data across a device reset.

Power-on-Reset Emulation
A software randomization process that writes random values in data RAM areas to simulate un-initialized values in RAM upon initial power application.

Pragma
A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information**.**

Precedence
Rules that define the order of evaluation in expressions.

Program Counter
The location that contains the address of the instruction that is currently executing.

Program Counter Unit
A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

Program Memory
The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

Program Memory
The memory area in a device where instructions are stored.

Project
A set of source files and instructions to build the object and executable code for an application.

Prologue
A portion of compiler-generated code that is responsible for allocating stack space, preserving registers and performing any other machine-specific requirement specified in the runtime model. This code executes before any user code for a given function.

Prototype System
A term referring to a user's target application, or target board.

PWM (Pulse Width Modulation)
PWM is a Digital Modulation Technique.

Radix
The number base, hex, or decimal, used in specifying an address.

RAM
Random Access Memory

Raw Data
The binary representation of code or data associated with a section.

Read Only Memory
Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

Real Time
When an in-circuit emulator or debugger is released from the halt state, the processor runs in Real Time mode and behaves exactly as the normal chip would behave. In Real Time mode, the real time trace buffer of an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.
In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Recursive Calls
A function that calls itself, either directly or indirectly.

Recursion
The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Reentrant
A function that may have multiple, simultaneously active instances. This may happen due to either direct or

indirect recursion or through execution during interrupt processing.

Relaxation
The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size.

Relocatable
An object whose address has not been assigned to a fixed location in memory.

Relocatable Section
A section whose address is not fixed through a process called relocation.

Relocation
A process performed by the linker in which absolute addresses are assigned to relocatable sections and all symbols in the relocatable sections are updated to their new addresses.

ROM
Read Only Memory

Run
The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

Run-time Model
Describes the use of target architecture resources.

Section
A portion of an application located at a specific address of memory.

Section Attribute
A characteristic ascribed to a section

SFR
See Special Function Registers.

Simulator
A software program that models the operation of devices.

Single Step
This command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution. You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew
The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid
When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code
The form in which a computer program is written by the programmer. Source code is written in a formal programming language which can be translated into machine code or executed by an interpreter.

Source File
An ASCII text file containing source code.

Special Function Registers
The portion of data memory timers or other modes or peripherals.

Stack, Hardware
Locations in some microcontrollers where the return address is stored when a function call is made or an interrupt occurs.

Stack, Software
Memory used by an application for storing return addresses, function parameters, and local variables. This memory is typically managed by the compiler when developing code in a high-level language.

Static RAM or SRAM
Static Random Access Memory. Program memory you can read/write on the target board that does not need refreshing frequently.

Status Bar
The Status Bar is located on the bottom of the IDE window and indicates such current information as cursor position, development mode and device, and active tool bar.

Step Into
This command is the same as Single Step. Step Into instruction into a subroutine.

Step Over
Step Over allows you to debug code without stepping into subroutines. When stepping over a CALL instruction, the next breakpoint will be set at the instruction after the CALL. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of CALL instructions.

Step Out
Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

Stimulus
Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous

Stopwatch
A counter for measuring execution cycles.

Storage Class
Determines the lifetime of the memory associated with the identified object.

Storage Qualifier
Indicates special properties of the objects being declared

Symbol
A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc. Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

Symbol, Absolute
Represents an immediate value such as a definition through the assembly .equ directive.

System Window Control
The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items "Minimize," "Maximize," and "Close."

Target
Refers to user hardware.

Target Application
Software residing on the target board.

Target Board
The circuitry and programmable device that makes up the target application.

Target Processor
The microcontroller device on the target application board.

Template
Lines of text that you build for inserting into your files at a later time.

Tool Bar
A row or column of icons that you can click on to execute MPLAB IDE functions.

Trace
An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE's trace window.

Trace Memory
Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

Trigger Output
Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs
Three-character sequences, all starting with ??, that are defined by ISO C as replacements for single characters.

Unassigned Section
A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data
Data which is defined without an initial value. In C,
int myVar; defines a variable which will reside in an uninitialized data section.

Upload
The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

USB
Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. Also referred to as high-speed USB, USB 2.0 supports data rates up to 480 Mbps.

Vector
The memory locations that an application will jump to when either a reset or interrupt occurs.

Warning
Warnings report conditions that may indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text 'warning:' to distinguish them from error messages.

Watch Variable
A variable that you may monitor during a debugging session in a Watch window.

Watch Window
Watch windows contain a list of watch variables that are updated at each breakpoint.

Watchdog Timer
A timer on a microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

WDT
See Watchdog Timer.

Table of Alphabetical Index