NED University of Engineering & Technology

Department of Electrical Engineering

LAB MANUAL

EMBEDDED SYSTEMS

(EE-354) For T.E.(EE)

Instructor name:	
Student name:	
Roll #	**Batch:**
Semester:	**Year:**

LAB MANUAL

For the course

EMBEDDED SYSTEMS

(EE-354) For T.E.(EE)

Developed by:

Mr. Hassan-ul-Haq, Mr. Hafiz Muhammad Furqan & Ms. Aiman

Approved By

The Board of Studies of Department of Electrical Engineering

_______________________ _______________________

_______________________ _______________________

_______________________ _______________________

Summary of Rubric Sheets for Data Entry in OBE MIS System [to be entered as they are on OBE portal with no scaling]

Lab #	Extent of achievement	Lab #	Extent of achievement	Lab #	Extent of achievement
	Total out of 32		Total out of 32		Total out of 32

Lab #	Extent of achievement	Lab #	Extent of achievement	Lab #	Extent of achievement
	Total out of 32		Total out of 32		Total out of 32

Add six total scores from above to get total Rubric score Out of 192	R=
Scaled to 25 marks (R/192)*25	S=
Lab Attendance percentage from portal:	A=
Attendance scaled to 5 marks: (A/100)*5	P=
For entry on NED exam portal Total lab sessionals out of 30: S + P	T=

PBL/OEL	Extent of achievement	Final	Extent of achievement
	Total out of 32		Total out of 32

Add two total scores from above to get total final Rubric Score Out of 64	Z=
For entry on NED exam portal Total Final Lab Marks scaled to 20 marks (Z/64)*20	F=

CONTENTS

Psychomotor: P3
CLO: ***Duplicate*** wiring connections for given circuit design while manipulating the embedded software with C/Assembly IDE in order to change system behavior.
PLO: Lifelong Learning- PLO 12

** RUBRIC based assessment*

LAB SESSION 01

OBJECTIVE:

To set-up the Code::Blocks IDE with AVR toolchain and test an AVR project on the ATmega328P microcontroller

LAB OUTCOMES:

By the end of the lab, you would be able to:

1) Install the Code::Blocks IDE with an AVR toolchain using (WinAVR)
2) Create an AVR project with the required compiler settings
3) Test the created project on ATmega328P microcontroller using AVRDUDE

BACKGROUND:

The ATmega328P is an 8-bit microcontroller based on the AVR RISC architecture. It is produced by Microchip Technology. The ATmega328P has 32 KB of flash memory, 2 KB of RAM, and 1 KB of EEPROM. It has 23 general-purpose I/O pins, a 16-bit timer/counter, a 8-bit timer/counter, a real-time clock, a pulse-width modulation (PWM) unit, a serial communication interface (USART), a two-wire interface (TWI), and an analog-to-digital converter (ADC).

The ATmega328P is a popular microcontroller for a variety of projects, including robotics, home automation, and wearables. It is also used in the Arduino Uno, Arduino Pro Mini, and Arduino Nano microcontroller development boards. The ATmega328P is a powerful and versatile microcontroller that can be used in a wide variety of applications. It is a popular choice for hobbyists and professionals alike.

We will be utilizing ATmega328P (Arduino UNO DIP R3) throughout Embedded Systems Lab work and will program it in C-language. To program the microcontroller, we will use the following:

- **Avrdude** is a command-line utility for programming AVR microcontrollers. It can be used to write firmware to the microcontroller's flash memory, erase the flash memory, and read the contents of the flash memory.
- **WinAVR** is a software package that includes avrdude, as well as a compiler, assembler, and a number of other tools for developing applications for AVR microcontrollers.
- **Code::Blocks** is an integrated development environment (IDE) that can be used to develop applications for AVR microcontrollers. It includes a graphical user interface for editing and compiling code, as well as a debugger for stepping through and debugging code.

Avrdude and Winavr are essential tools for developing applications for AVR microcontrollers. Code::Blocks is a powerful IDE that can make the development process easier.

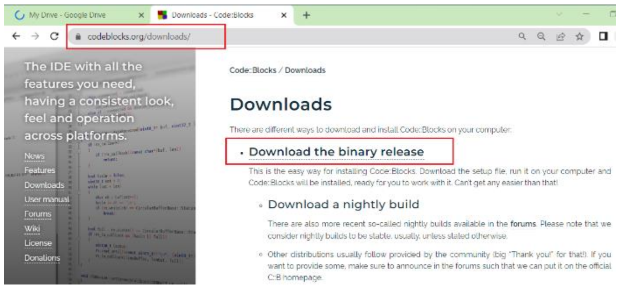
LAB TASKS

The first two lab tasks guide you to installation of the required IDE and toolchain. The setup files can be easily found and downloaded from the internet. However, these setup files are available on this shared folder too.

TASK 1: To install Code::Blocks IDE with AVR Toolchain

Follow the given step-by-step procedure to first install the Code::Blocks IDE on your system.

1) To download the required setup files from the internet, search CodeBlocks and go to https://www.codeblocks.org/downloads/. Select ***Download the binary releases***. Select the setup package depending upon your platform like Microsoft Windows in our case. From the given setup files, select ***codeblocks-20.03-setup.exe*** as shown in Figure 1**.** Click on one of the **Download From** options for example **Sourceforge.net**. For a 32-bit operating system type, you can select ***codeblocks-20.03-32bit-setup.exe.***
2) Once the setup file is downloaded, click on it to begin the installation process. Select the default option and follow the installation steps as suggested by the wizard. After a few minutes of decompressing and install files, click YES when prompted to start Code::Blocks. This should yield the IDE shown in Figure. For now, select OK if it fails to auto-detect the compiler.


My Drive - Google Drive
Downloads - Code::Blocks
codeblocks.org/downloads/
The IDE with all the features you need, having a consistent look, feel and operation across platforms.
News
Features
Downloads
User manual
Forums
Wiki
License
Donations

Code::Blocks / Downloads

Downloads

There are different ways to download and install Code::Blocks on your computer:

- Download the binary release

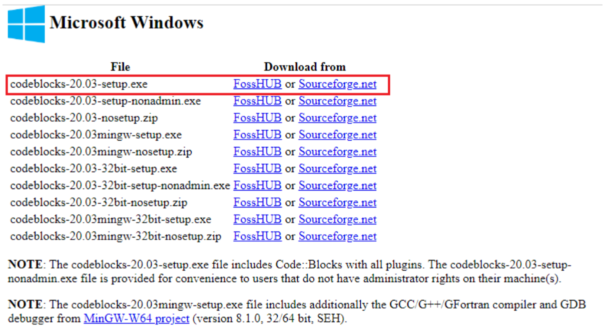
  This is the easy way for installing Code::Blocks. Download the setup file, run it on your computer and Code::Blocks will be installed, ready for you to work with it. Can't get any easier than that!

  - Download a nightly build

    There are also more recent so-called nightly builds available in the forums. Please note that we consider nightly builds to be stable, usually, unless stated otherwise.

  - Other distributions usually follow provided by the community (big "Thank you!" for that!). If you want to provide some, make sure to announce in the forums such that we can put it on the official C::B homepage.

Figure 1: Download Code::Blocks


Microsoft Windows

File	Download from
codeblocks-20.03-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-nosetup.zip	FossHUB or Sourceforge.net

NOTE: The codeblocks-20.03-setup.exe file includes Code::Blocks with all plugins. The codeblocks-20.03-setup-nonadmin.exe file is provided for convenience to users that do not have administrator rights on their machine(s).

NOTE: The codeblocks-20.03mingw-setup.exe file includes additionally the GCC/G++/GFortran compiler and GDB debugger from MinGW-W64 project (version 8.1.0, 32/64 bit, SEH).

Figure 2: Select Source and File Type

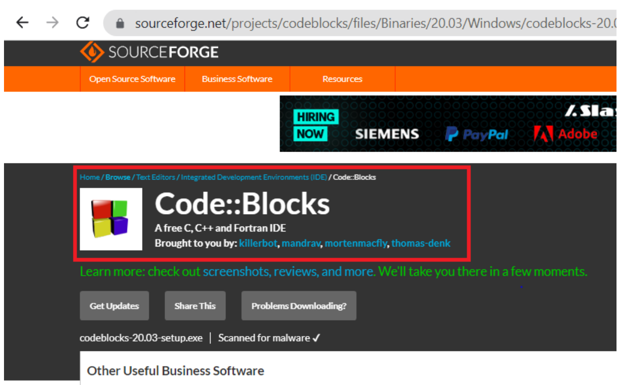
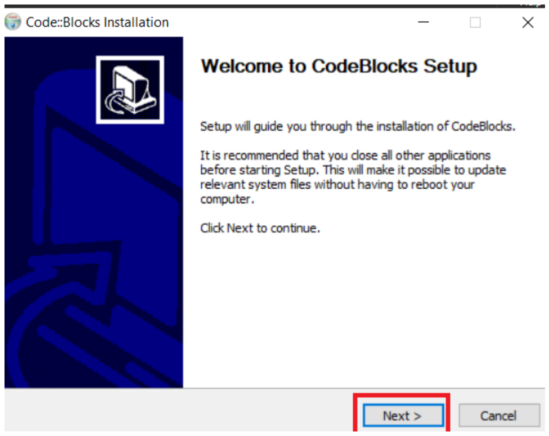

sourceforge.net/projects/codeblocks/files/Binaries/20.03/Windows/codeblocks-20.(
SOURCEFORGE
Open Source Software
Business Software
Resources
HIRING NOW
SIEMENS
PayPal
Adobe
Home / Browse / Text Editors / Integrated Development Environments (IDE) / Code::Blocks
Code::Blocks
A free C, C++ and Fortran IDE
Brought to you by: killerbot, mandrav, mortenmacfly, thomas-denk
Learn more: check out screenshots, reviews, and more. We'll take you there in a few moments.
Get Updates
Share This
Problems Downloading?
codeblocks-20.03-setup.exe | Scanned for malware ✓
Other Useful Business Software

Figure 3: Downlod Source



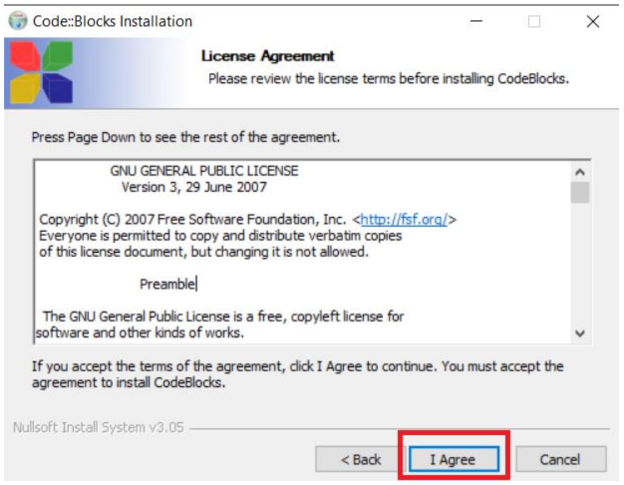
Welcome to CodeBlocks Setup

Setup will guide you through the installation of CodeBlocks.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Figure 4: Starting Installation Wizard



License Agreement
Please review the license terms before installing CodeBlocks.

Press Page Down to see the rest of the agreement.

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

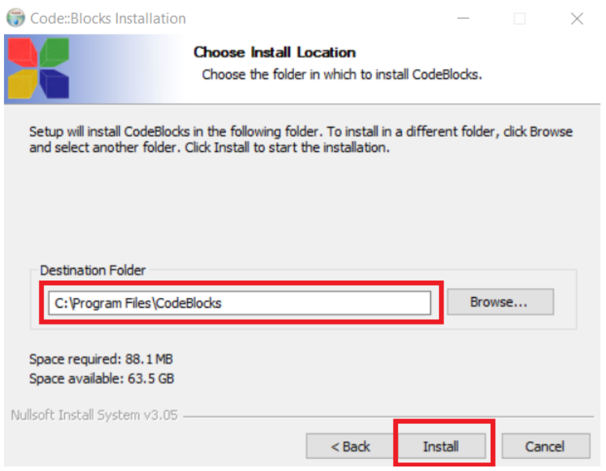
Preamble

The GNU General Public License is a free, copyleft license for
software and other kinds of works.

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install CodeBlocks.

Nullsoft Install System v3.05

Figure 5: License Agreement



Choose Install Location
Choose the folder in which to install CodeBlocks.

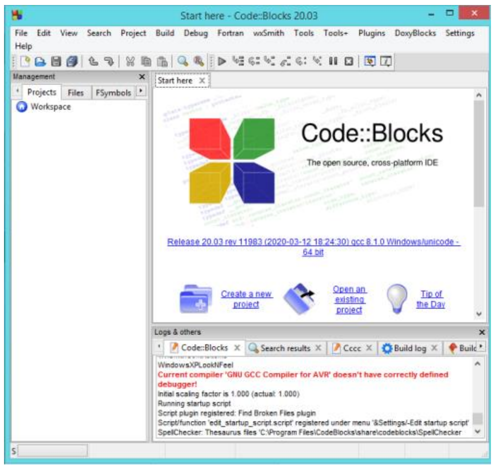
Setup will install CodeBlocks in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder
C:\Program Files\CodeBlocks

Space required: 88.1 MB
Space available: 63.5 GB

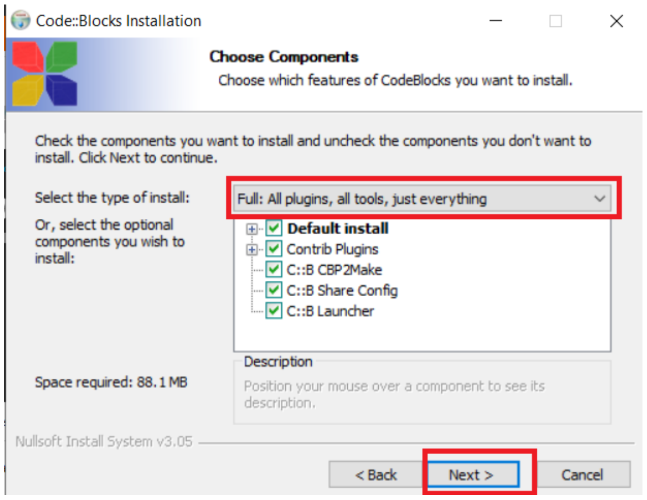
Nullsoft Install System v3.05

Figure 7: Destination Folder for Installation


Start here - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management
Projects Files FSymbols
Workspace
Start here
Code::Blocks
The open source, cross-platform IDE
Release 20.03 rev 11983 (2020-03-12 18:24:30) gcc 8.1.0 Windows/unicode - 64 bit
Create a new project
Open an existing project
Tip of the Day
Logs & others
Code::Blocks
Search results
Cccc
Build log

```
WindowsXPLookNFeel
Current compiler 'GNU GCC Compiler for AVR' doesn't have correctly defined debugger!
Initial scaling factor is 1.000 (actual: 1.000)
Running startup script
Script plugin registered: Find Broken Files plugin
Script/function 'edit_startup_script.script' registered under menu '&Settings/-Edit startup script'
SpellChecker: Thesaurus files 'C:\Program Files\CodeBlocks\share\codeblocks\SpellChecker
```

Figure 9: Code::Blocks IDE



Choose Components
Choose which features of CodeBlocks you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Full: All plugins, all tools, just everything

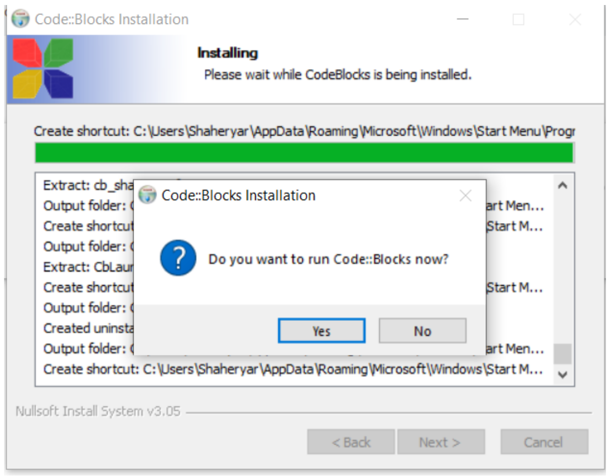
Or, select the optional components you wish to install:
- ✓ **Default install**
- ✓ Contrib Plugins
- ✓ C::B CBP2Make
- ✓ C::B Share Config
- ✓ C::B Launcher

Space required: 88.1 MB

Description
Position your mouse over a component to see its description.

Nullsoft Install System v3.05

Figure 6: Plug-in Selection



Installing
Please wait while CodeBlocks is being installed.

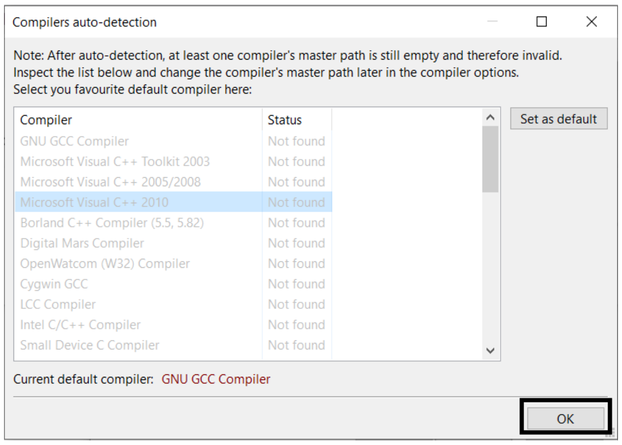
Create shortcut: C:\Users\Shaheryar\AppData\Roaming\Microsoft\Windows\Start Menu\Progr

```
Extract: cb_sha                                  
Output folder:                                   art Men...
Create shortcut                                  Start M...
Output folder:
Extract: CbLaun
Create shortcut                                  Start M...
Output folder:
Created uninsta
Output folder:                                   art Men...
Create shortcut: C:\Users\Shaheryar\AppData\Roaming\Microsoft\Windows\Start M...
```

Do you want to run Code::Blocks now?

Nullsoft Install System v3.05

Figure 8: Installation Completion



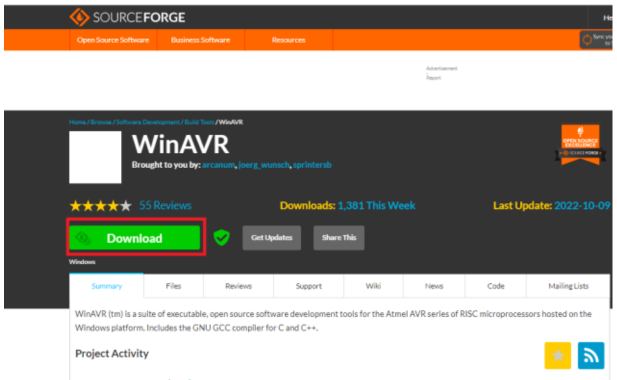
Note: After auto-detection, at least one compiler's master path is still empty and therefore invalid.
Inspect the list below and change the compiler's master path later in the compiler options.
Select you favourite default compiler here:

Compiler	Status
GNU GCC Compiler	Not found
Microsoft Visual C++ Toolkit 2003	Not found
Microsoft Visual C++ 2005/2008	Not found
Microsoft Visual C++ 2010	Not found
Borland C++ Compiler (5.5, 5.82)	Not found
Digital Mars Compiler	Not found
OpenWatcom (W32) Compiler	Not found
Cygwin GCC	Not found
LCC Compiler	Not found
Intel C/C++ Compiler	Not found
Small Device C Compiler	Not found

Current default compiler: GNU GCC Compiler

Figure 10: Ignore Compiler Status

3) The Code::Blocks IDE is now ready and would allow you to create projects. Since our aim is to develop AVR Projects, therefore, we first need to install AVR Toolchain. For this we will make use of WinAVR.
4) Search for WinAVR and go to https://sourceforge.net/projects/winavr/files/latest/download. Click on **Download**. It will start downloading the latest WinAVR package.
5) Click on the downloaded file to start the setup wizard. Follow the steps shown in figures and select the correct features to be installed. The WinAVR package has the required **avrdude** as well as GNU GCC Compilers.



WinAVR

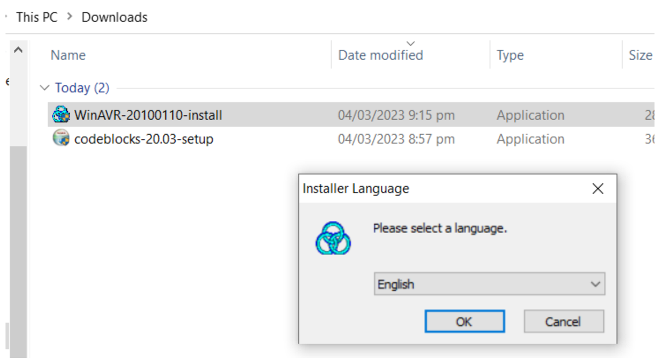
Brought to you by: arcanum, joerg_wunsch, sprintersb

55 Reviews Downloads: 1,381 This Week Last Update: 2022-10-09

WinAVR (tm) is a suite of executable, open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. Includes the GNU GCC compiler for C and C++.

Project Activity

Figure 11: Source to Download WinAVR



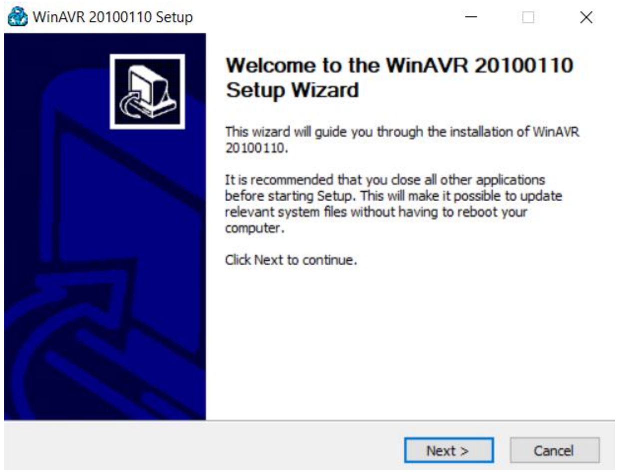
Name	Date modified	Type
WinAVR-20100110-install	04/03/2023 9:15 pm	Application
codeblocks-20.03-setup	04/03/2023 8:57 pm	Application

Installer Language

Please select a language.

English

Figure 12: Run Dwonloaded Application File



WinAVR 20100110 Setup

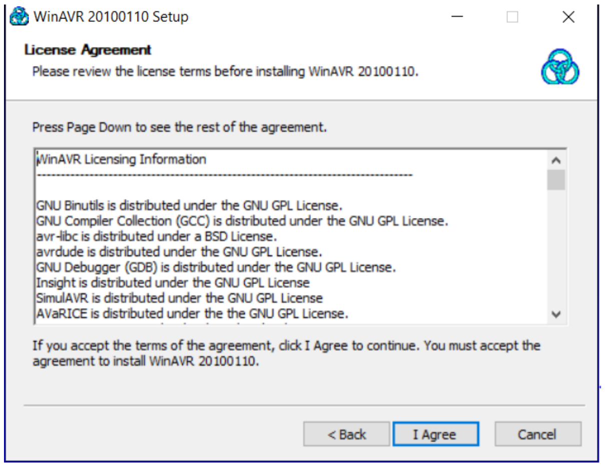
Welcome to the WinAVR 20100110 Setup Wizard

This wizard will guide you through the installation of WinAVR 20100110.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Figure 13: Setup Wizard



WinAVR 20100110 Setup

License Agreement

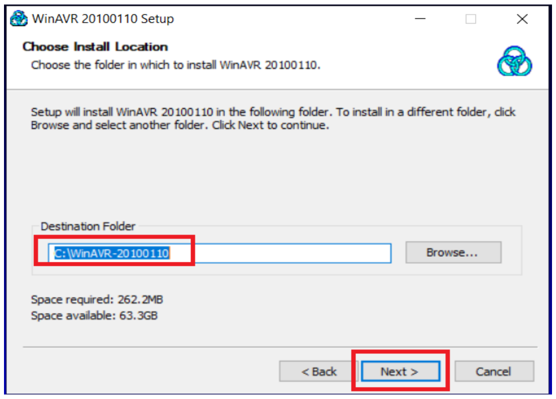
Please review the license terms before installing WinAVR 20100110.

Press Page Down to see the rest of the agreement.

```
WinAVR Licensing Information
----------------------------------------------------------------------------

GNU Binutils is distributed under the GNU GPL License.
GNU Compiler Collection (GCC) is distributed under the GNU GPL License.
avr-libc is distributed under a BSD License.
avrdude is distributed under the GNU GPL License.
GNU Debugger (GDB) is distributed under the GNU GPL License.
Insight is distributed under the GNU GPL License
SimulAVR is distributed under the GNU GPL License
AVaRICE is distributed under the the GNU GPL License.
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install WinAVR 20100110.

Figure 14: License Argeement



WinAVR 20100110 Setup

Choose Install Location

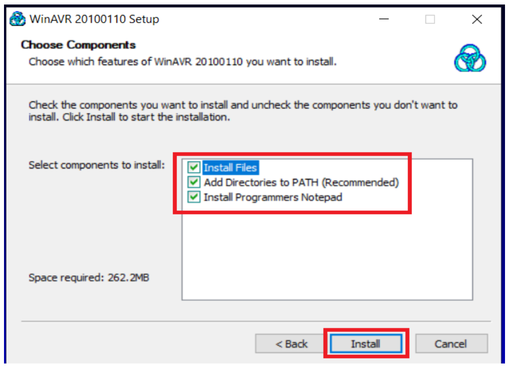
Choose the folder in which to install WinAVR 20100110.

Setup will install WinAVR 20100110 in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder: C:\WinAVR-20100110

Space required: 262.2MB
Space available: 63.3GB

Figure 15: Destination Folder



WinAVR 20100110 Setup

Choose Components

Choose which features of WinAVR 20100110 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Install to start the installation.

Select components to install:
- [✓] Install Files
- [✓] Add Directories to PATH (Recommended)
- [✓] Install Programmers Notepad

Space required: 262.2MB

Figure 16: Choose Components for Installation

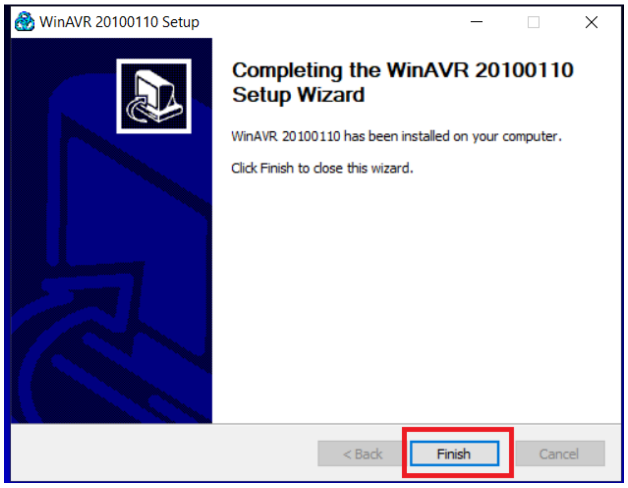
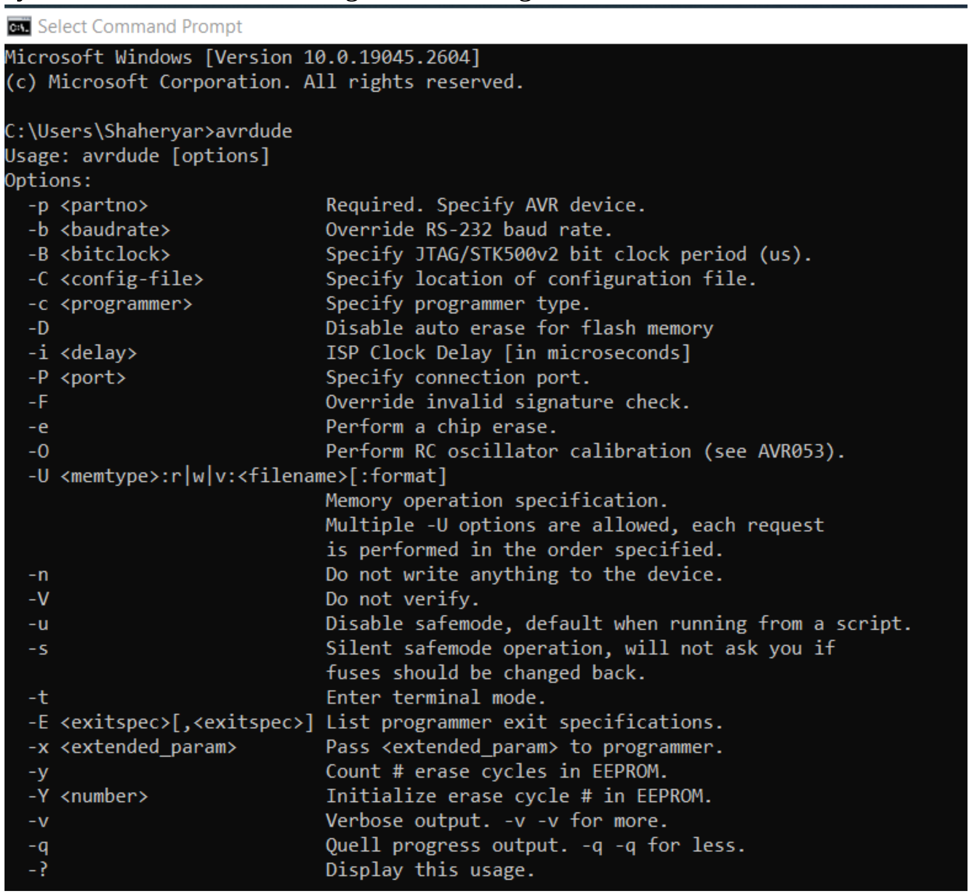

WinAVR 20100110 Setup
Completing the WinAVR 20100110 Setup Wizard
WinAVR 20100110 has been installed on your computer.
Click Finish to close this wizard.
< Back
Finish
Cancel

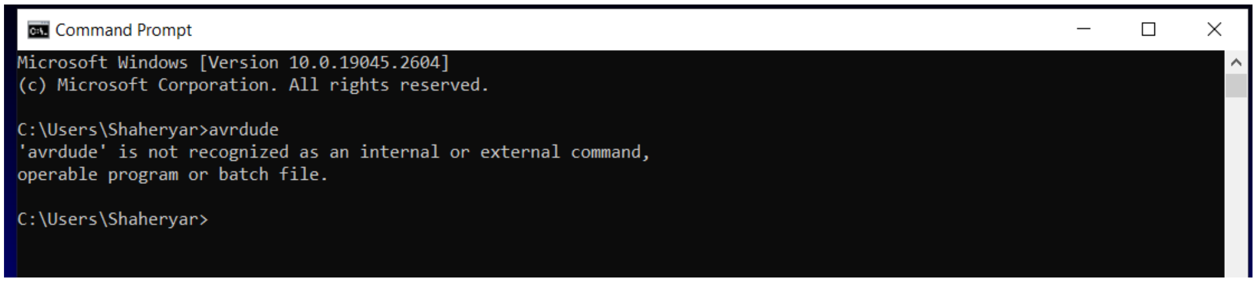
Figure 17: Finish Installation

6) Before you create and start working on a project, verify that directories are correctly added to the system path. To do this, open the command prompt by searching **cmd** in the Windows search option. In the command prompt, simply write **avrdude.** You will see the message shown in Figure if it is correctly added else you will see the error message shown in Figure.



```
Microsoft Windows [Version 10.0.19045.2604]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Shaheryar>avrdude
Usage: avrdude [options]
Options:
  -p <partno>                Required. Specify AVR device.
  -b <baudrate>              Override RS-232 baud rate.
  -B <bitclock>              Specify JTAG/STK500v2 bit clock period (us).
  -C <config-file>           Specify location of configuration file.
  -c <programmer>            Specify programmer type.
  -D                         Disable auto erase for flash memory
  -i <delay>                 ISP Clock Delay [in microseconds]
  -P <port>                  Specify connection port.
  -F                         Override invalid signature check.
  -e                         Perform a chip erase.
  -O                         Perform RC oscillator calibration (see AVR053).
  -U <memtype>:r|w|v:<filename>[:format]
                             Memory operation specification.
                             Multiple -U options are allowed, each request
                             is performed in the order specified.
  -n                         Do not write anything to the device.
  -V                         Do not verify.
  -u                         Disable safemode, default when running from a script.
  -s                         Silent safemode operation, will not ask you if
                             fuses should be changed back.
  -t                         Enter terminal mode.
  -E <exitspec>[,<exitspec>] List programmer exit specifications.
  -x <extended_param>        Pass <extended_param> to programmer.
  -y                         Count # erase cycles in EEPROM.
  -Y <number>                Initialize erase cycle # in EEPROM.
  -v                         Verbose output. -v -v for more.
  -q                         Quell progress output. -q -q for less.
  -?                         Display this usage.
```

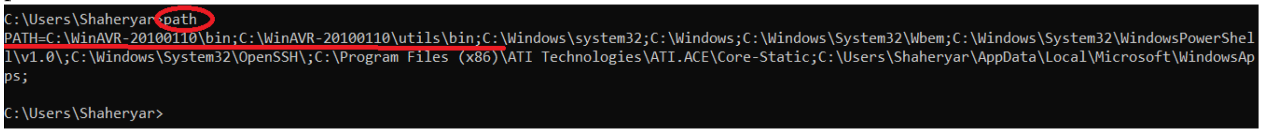
Figure 18: Testing **avrdude** through Command Prompt



```
Command Prompt
Microsoft Windows [Version 10.0.19045.2604]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Shaheryar>avrdude
'avrdude' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Shaheryar>
```

Figure 19: Error-**avrdude** is not recognized as internal command

7) After this write **path** to verify that the required directories shown in the figure are added to the system path.



```
C:\Users\Shaheryar>path
PATH=C:\WinAVR-20100110\bin;C:\WinAVR-20100110\utils\bin;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShel
l\v1.0\;C:\Windows\System32\OpenSSH\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static;C:\Users\Shaheryar\AppData\Local\Microsoft\WindowsAp
ps;

C:\Users\Shaheryar>
```

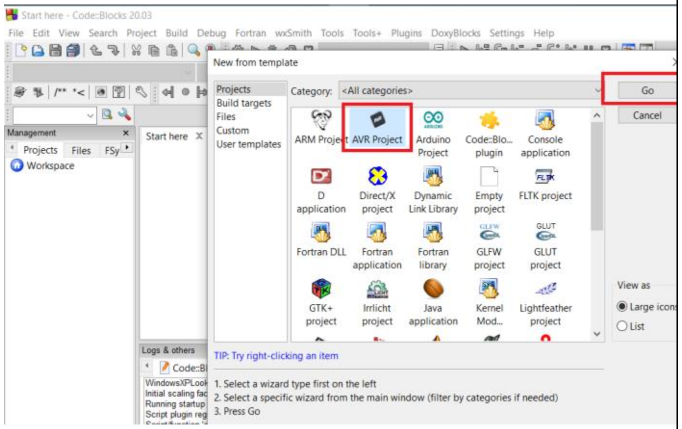
Figure 20: Verifying Directories Added in System Path

TASK 2: To create an AVR Project with the required compiler settings

Now you are ready to create and build an AVR project through the Code::Blocks IDE. The following steps guide you to create the project and set the compiler settings required to build the project. Follow the steps given below.

a) Creating AVR Project:

1) Open the Code::Blocks. Go to ***File> Create >Project….*** Select AVR Project and click ***Go***. Select ***Next.***
2) At this step, you are asked to give project title and select folder to create project files. Give any suitable name to your project for example here, we have called it ***Project1.*** A new folder **ES Labs** is created on the desktop and is chosen as project folder. After this, click ***Next.***
3) Now, make sure the selected compiler is **GNU GCC for AVR** and keep the remaining settings as shown in the figure. Then click Next.
4) The processor we have selected for ES labs is ATmega328P. Select it from the dropdown menu carefully and keep the rest of the settings as shown in the Figure. Then click ***Finish***
5) You will see your created project ***Project1*** folder created in the Projects tab workspace. Click on the Project Name > Sources. This folder will show 2 files; **main.c** and **fuse.c**, double click on the main.c file to open it. It is a blank file template created for the AVR project. This is where you will write a code.

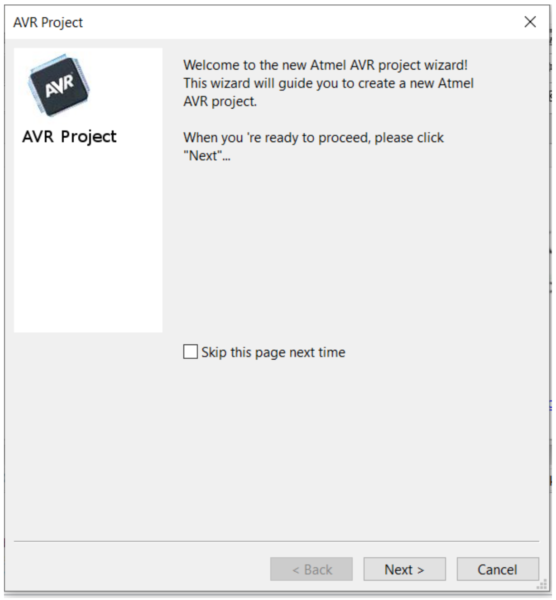


Category: <All categories>

TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Figure 21: Creating AVR Project

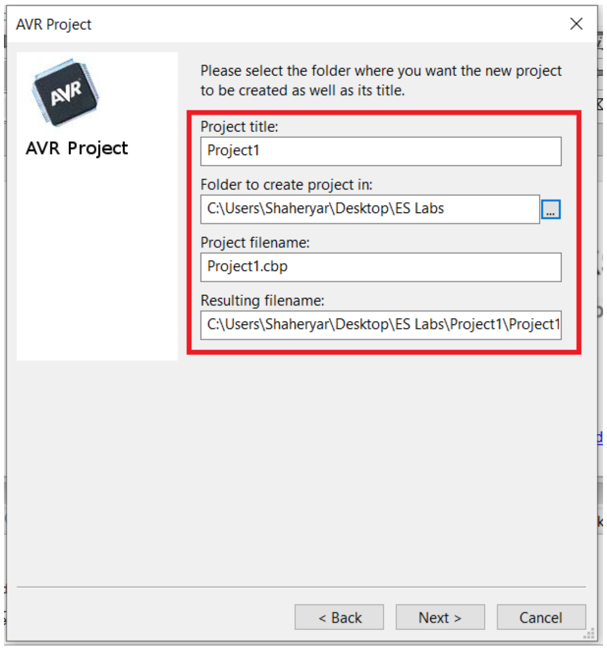


Welcome to the new Atmel AVR project wizard!
This wizard will guide you to create a new Atmel AVR project.

When you 're ready to proceed, please click "Next"...

☐ Skip this page next time

Figure 22: AVR Project Wizard



Please select the folder where you want the new project to be created as well as its title.

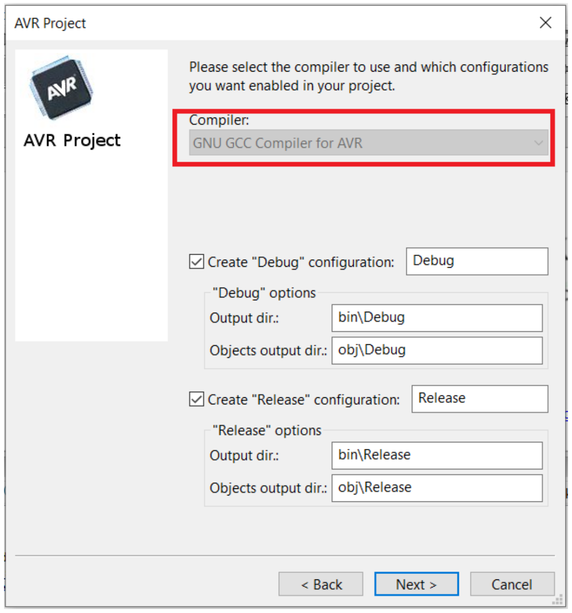
Project title: Project1

Folder to create project in: C:\Users\Shaheryar\Desktop\ES Labs

Project filename: Project1.cbp

Resulting filename: C:\Users\Shaheryar\Desktop\ES Labs\Project1\Project1

Figure 23: Project Details



Please select the compiler to use and which configurations you want enabled in your project.

Compiler: GNU GCC Compiler for AVR

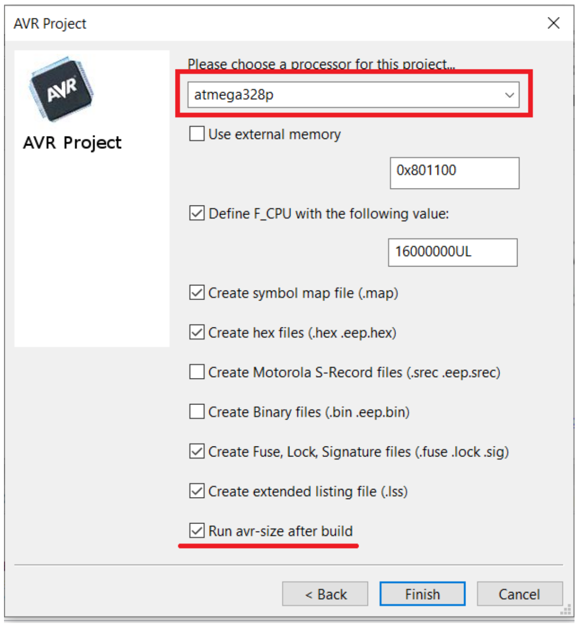
☑ Create "Debug" configuration: Debug

"Debug" options
Output dir.: bin\Debug
Objects output dir.: obj\Debug

☑ Create "Release" configuration: Release

"Release" options
Output dir.: bin\Release
Objects output dir.: obj\Release

Figure 24: Compiler Selection


AVR Project

Please choose a processor for this project...

atmega328p

☐ Use external memory

0x801100

☑ Define F_CPU with the following value:

16000000UL

☑ Create symbol map file (.map)

☑ Create hex files (.hex .eep.hex)

☐ Create Motorola S-Record files (.srec .eep.srec)

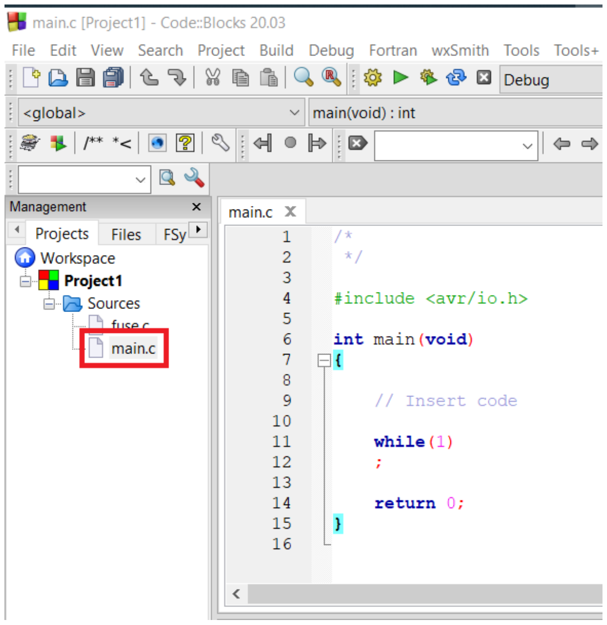
☐ Create Binary files (.bin .eep.bin)

☑ Create Fuse, Lock, Signature files (.fuse .lock .sig)

☑ Create extended listing file (.lss)

☑ Run avr-size after build

Figure 25: Choosing Processor for AVR Project


main.c [Project1] - Code::Blocks 20.03

```
/*
 */

#include <avr/io.h>

int main(void)
{

    // Insert code

    while(1)
    ;

    return 0;
}
```

Figure 26: Created Empty Project with main.c and fuse.c files

b) <u>Compiler Settings:</u>

Before we write instructions for our first test project, let's first complete the required compiler settings that will be needed for all AVR projects.

6) Now, go to ***Settings*** > ***Compiler.***

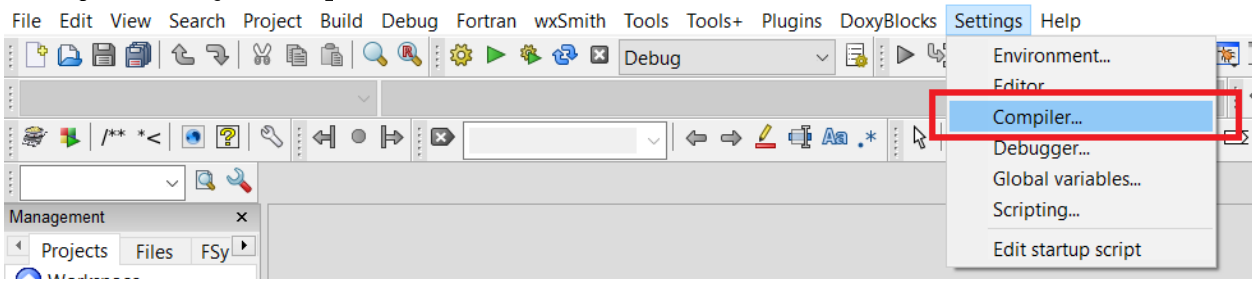
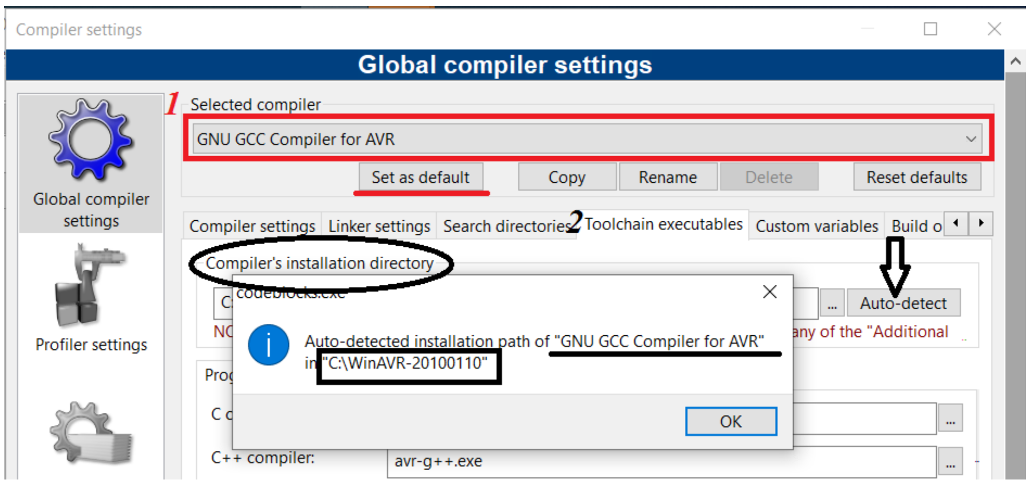


Figure 27: Accessing Compiler Settings Option

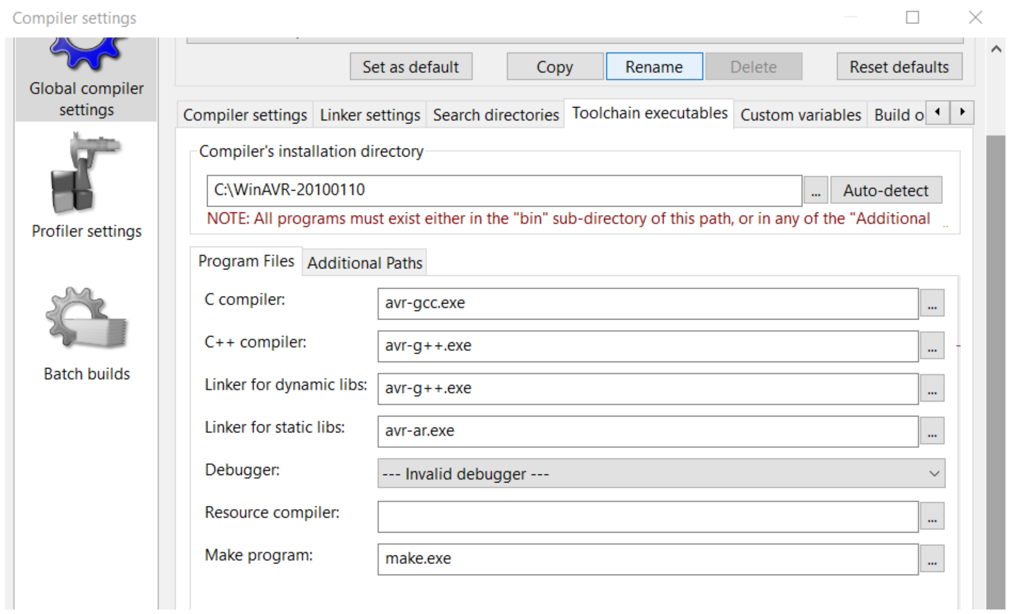
7) Under the **Selected Compiler** dropdown menu, select **GNU GCC Compiler for AVR** and click **Set as default.** From the different tabs right below this, select **Toolchain executables.** For **Compiler's installation directory**, click **Auto detect.** It should show the auto detected installation path as the one where WinAVR destination folder was selected earlier. If it fails to do so, you can manually select the folder by browsing through **…** option beside Auto-detect. Click OK.



Auto-detected installation path of "GNU GCC Compiler for AVR" in "C:\WinAVR-20100110"

Figure 28: Specifying the Compiler and Installation Directory in Toolchain Executables

This will update a number of things for these settings. Verify each as shown in the following figures.



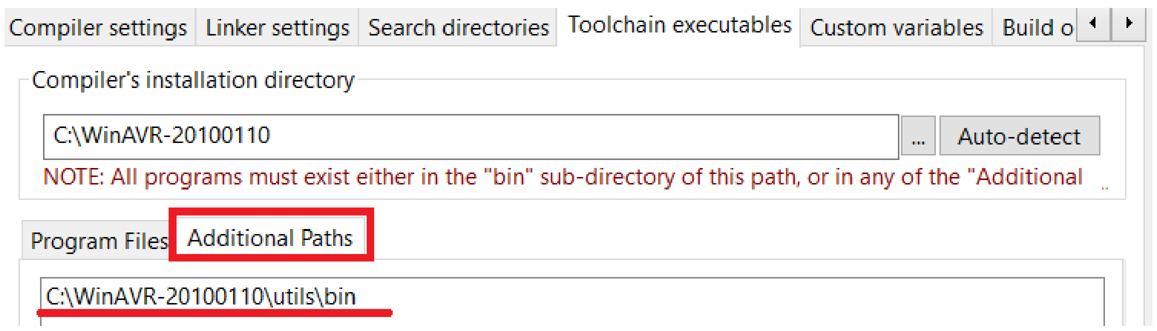
Compiler's installation directory: C:\WinAVR-20100110

NOTE: All programs must exist either in the "bin" sub-directory of this path, or in any of the "Additional

C compiler: avr-gcc.exe
C++ compiler: avr-g++.exe
Linker for dynamic libs: avr-g++.exe
Linker for static libs: avr-ar.exe
Debugger: --- Invalid debugger ---
Resource compiler:
Make program: make.exe

Figure 29: Specifying the Program Files in the Compiler's Installation Directory

The **Program Files** and **Additional Paths** tabs under the Toolchain executables will be updated as shown.



Compiler's installation directory: C:\WinAVR-20100110

NOTE: All programs must exist either in the "bin" sub-directory of this path, or in any of the "Additional

C:\WinAVR-20100110\utils\bin

Figure 30: Additional Paths in Toolchain Executables

Go to **Search directories** tab and check that C:WinAVR\avr\include is added in the Compiler tab.

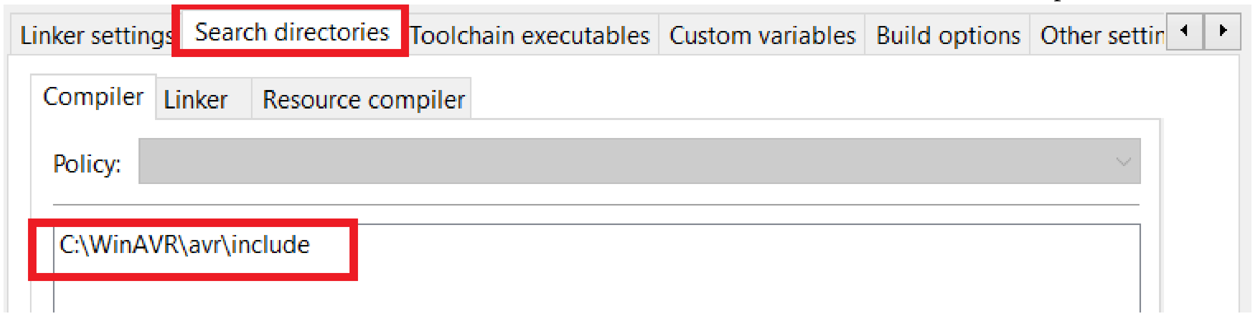
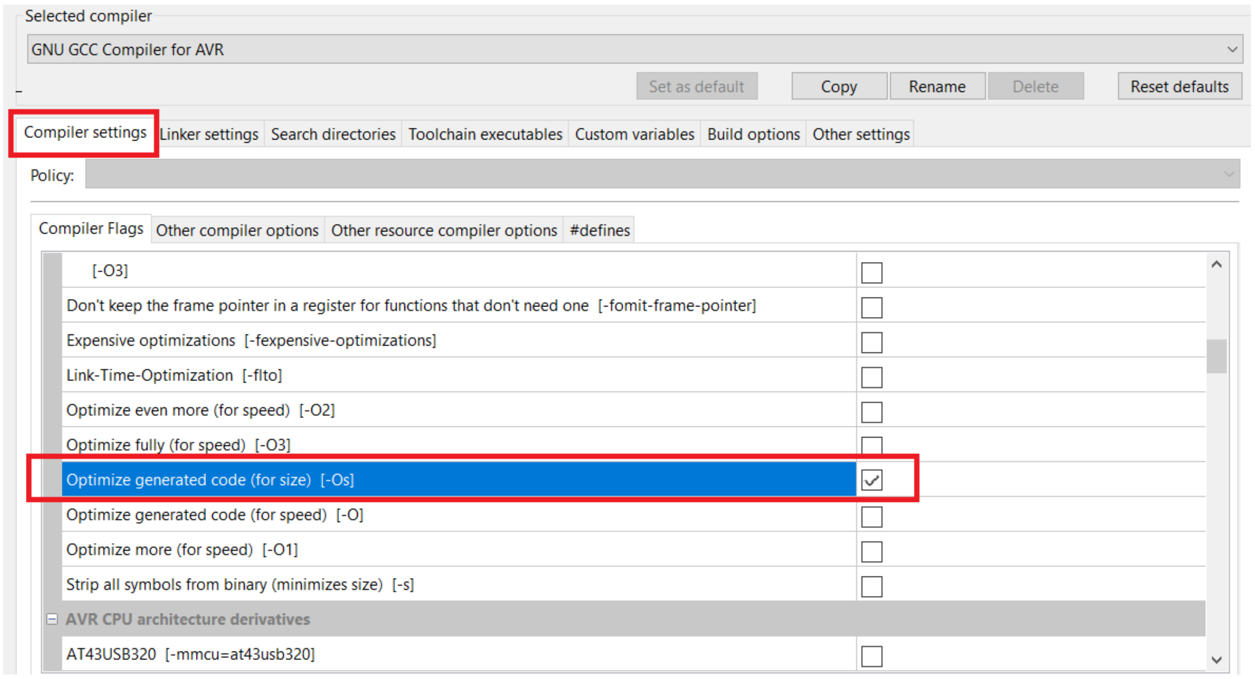


Figure 31: Verifying the Search Directories

Select **Compiler settings** tab at the left-most. Check the **Optimize generated code (for size) [-Os]** option.



[-O3]	☐
Don't keep the frame pointer in a register for functions that don't need one [-fomit-frame-pointer]	☐
Expensive optimizations [-fexpensive-optimizations]	☐
Link-Time-Optimization [-flto]	☐
Optimize even more (for speed) [-O2]	☐
Optimize fully (for speed) [-O3]	☐
Optimize generated code (for size) [-Os]	✓
Optimize generated code (for speed) [-O]	☐
Optimize more (for speed) [-O1]	☐
Strip all symbols from binary (minimizes size) [-s]	☐
AVR CPU architecture derivatives	
AT43USB320 [-mmcu=at43usb320]	☐

Figure 32: Optimization Settings for Compiler

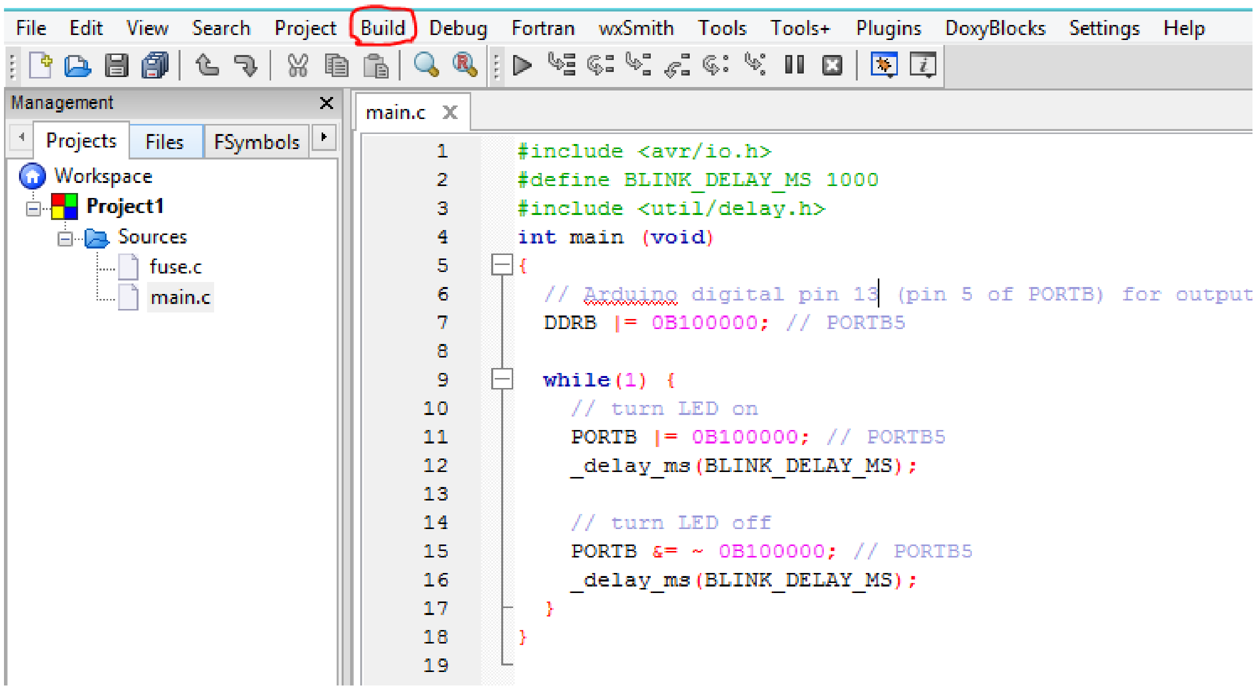
8) Click OK to close the window. The GNU GCC Compiler is set as default with the required compiler settings needed for now.

c) <u>Add Code and Build Project:</u>

1) Update the main.c file opened in the Editor window with the code given in Figure and save the file (Ctrl+S). This is a test code that causes the LED on board blink with a delay of 1000msec (1sec).

```
#include <avr/io.h>
#define BLINK_DELAY_MS 1000
#include <util/delay.h>
int main (void)
{
 // Arduino digital pin 13 (pin 5 of PORTB) for output
 DDRB |= 0B100000; // PORTB5
 while(1) {
  // turn LED on
  PORTB |= 0B100000; // PORTB5
  _delay_ms(BLINK_DELAY_MS);
  // turn LED off
  PORTB &= ~ 0B100000; // PORTB5
  _delay_ms(BLINK_DELAY_MS);
 }
}
```

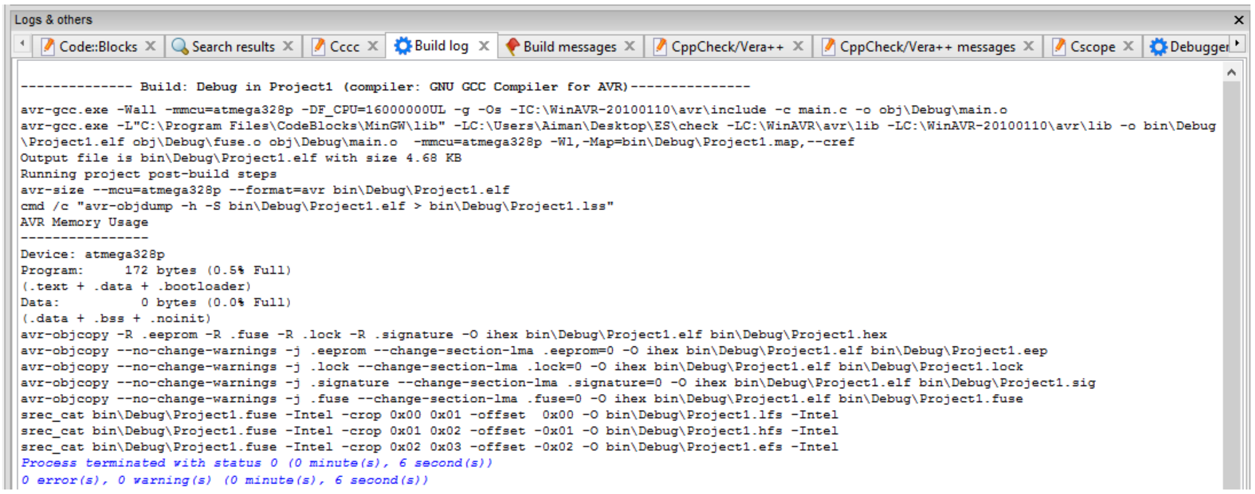
Figure 33: Test Code - LED Blink



```
#include <avr/io.h>
#define BLINK_DELAY_MS 1000
#include <util/delay.h>
int main (void)
{
 // Arduino digital pin 13 (pin 5 of PORTB) for output
 DDRB |= 0B100000; // PORTB5

 while(1) {
  // turn LED on
  PORTB |= 0B100000; // PORTB5
  _delay_ms(BLINK_DELAY_MS);

  // turn LED off
  PORTB &= ~ 0B100000; // PORTB5
  _delay_ms(BLINK_DELAY_MS);
 }
}
```

Figure 34: Updating main.c File with Test Code

2) Now select **Build** > **Build.** For a successful build, the Build Log below will show the following message with 0 errors and 0 warnings.



```
-------------- Build: Debug in Project1 (compiler: GNU GCC Compiler for AVR)---------------
avr-gcc.exe -Wall -mmcu=atmega328p -DF_CPU=16000000UL -g -Os -IC:\WinAVR-20100110\avr\include -c main.c -o obj\Debug\main.o
avr-gcc.exe -L"C:\Program Files\CodeBlocks\MinGW\lib" -LC:\Users\Aiman\Desktop\ES\check -LC:\WinAVR\avr\lib -LC:\WinAVR-20100110\avr\lib -o bin\Debug
\Project1.elf obj\Debug\fuse.o obj\Debug\main.o  -mmcu=atmega328p -Wl,-Map=bin\Debug\Project1.map,--cref
Output file is bin\Debug\Project1.elf with size 4.68 KB
Running project post-build steps
avr-size --mcu=atmega328p --format=avr bin\Debug\Project1.elf
cmd /c "avr-objdump -h -S bin\Debug\Project1.elf > bin\Debug\Project1.lss"
AVR Memory Usage
----------------
Device: atmega328p
Program:     172 bytes (0.5% Full)
(.text + .data + .bootloader)
Data:          0 bytes (0.0% Full)
(.data + .bss + .noinit)
avr-objcopy -R .eeprom -R .fuse -R .lock -R .signature -O ihex bin\Debug\Project1.elf bin\Debug\Project1.hex
avr-objcopy --no-change-warnings -j .eeprom --change-section-lma .eeprom=0 -O ihex bin\Debug\Project1.elf bin\Debug\Project1.eep
avr-objcopy --no-change-warnings -j .lock --change-section-lma .lock=0 -O ihex bin\Debug\Project1.elf bin\Debug\Project1.lock
avr-objcopy --no-change-warnings -j .signature --change-section-lma .signature=0 -O ihex bin\Debug\Project1.elf bin\Debug\Project1.sig
avr-objcopy --no-change-warnings -j .fuse --change-section-lma .fuse=0 -O ihex bin\Debug\Project1.elf bin\Debug\Project1.fuse
srec_cat bin\Debug\Project1.fuse -Intel -crop 0x00 0x01 -offset  0x00 -O bin\Debug\Project1.lfs -Intel
srec_cat bin\Debug\Project1.fuse -Intel -crop 0x01 0x02 -offset -0x01 -O bin\Debug\Project1.hfs -Intel
srec_cat bin\Debug\Project1.fuse -Intel -crop 0x02 0x03 -offset -0x02 -O bin\Debug\Project1.efs -Intel
Process terminated with status 0 (0 minute(s), 6 second(s))
0 error(s), 0 warning(s) (0 minute(s), 6 second(s))
```

Figure 35: Build Log (Successful Build)

3) Upon successful build, you will see some new folders and files added to the project folder. Go to bin and verify the addition of **.hex** file as shown in the figure.

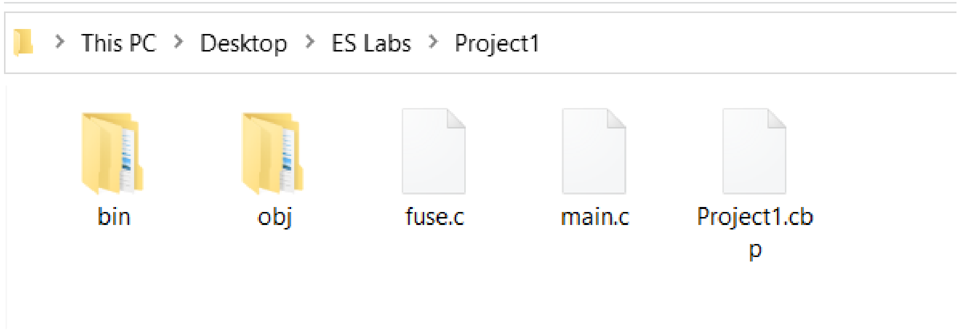
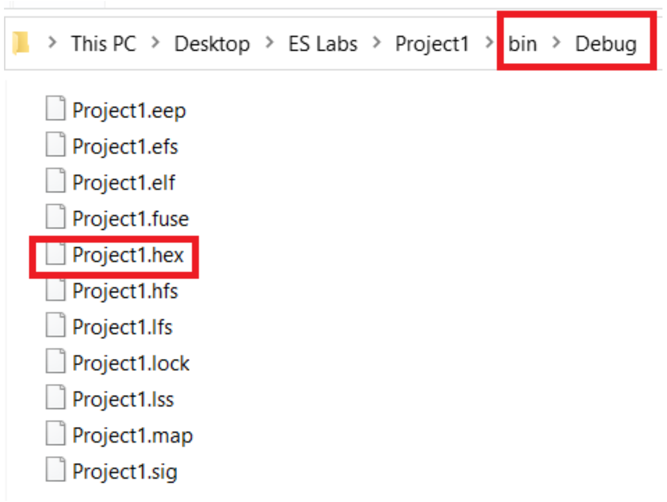


Figure 36: Project Folder Updated after Building the Project



This PC > Desktop > ES Labs > Project1 > bin > Debug

Project1.eep
Project1.efs
Project1.elf
Project1.fuse
Project1.hex
Project1.hfs
Project1.lfs
Project1.lock
Project1.lss
Project1.map
Project1.sig

Figure 37: Generated Files

Congratulations! You have generated the output file for our code written in the C language. This .hex file is the one that will be uploaded to the flash memory of microcontroller ATmega328P for execution of the code.

TASK 3: To test the project on ATmega328P microcontroller

Arduino UNO R3 DIP uses ATmega328P microcontroller IC. Therefore, we will be using the Arduino UNO board to program the ATmega328P.

1) Connect Arduino UNO USB cable with your system port. The LEDs on your board should light-up to verify that connection is made. The system might start installing the required drivers. To verify the correct connection, go to Device Manager of your system, and check **Ports (COM & LPT).** If the device is detected as Arduino UNO or USB Serial Device, note the port it is connected to. Here, we can see **Arduino UNO (COM3)** so COM3 is the port**.** In case the system fails to identify the device, you need to install Arduino Drivers. For this download the drivers from the internet or access through the shared drive. Extract the folder contents. Then, right click on the device under Ports in the Device Manager, and select Update Driver Software… and follow the Wizard. You will have to specify the path of downloaded driver files.

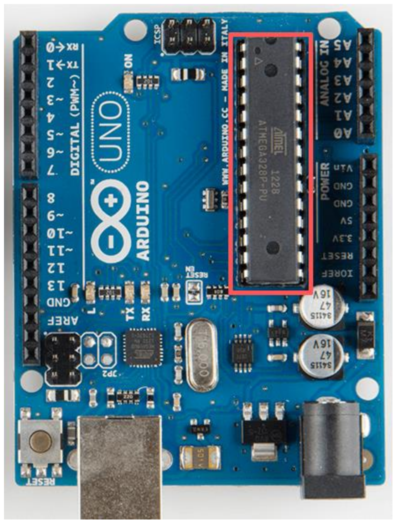
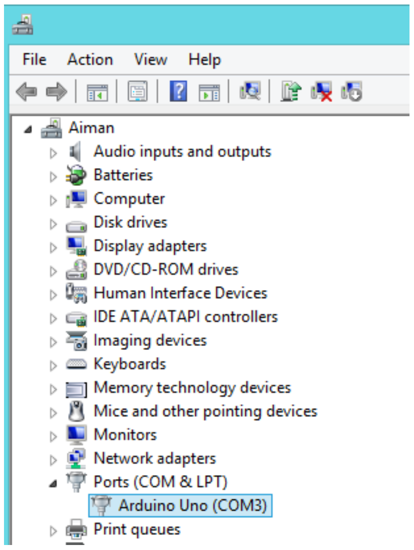


Figure 38: Arduino UNO R3 DIP (ATmega328P)



- Aiman
  - Audio inputs and outputs
  - Batteries
  - Computer
  - Disk drives
  - Display adapters
  - DVD/CD-ROM drives
  - Human Interface Devices
  - IDE ATA/ATAPI controllers
  - Imaging devices
  - Keyboards
  - Memory technology devices
  - Mice and other pointing devices
  - Monitors
  - Network adapters
  - Ports (COM & LPT)
    - Arduino Uno (COM3)
  - Print queues

Figure 39: Arduino UNO Connection Port

There are 2 approaches to upload the generated code on ATmega328P flash memory. First we will use **AVRDUDE** through **Command Prompt** instructions. Through this, you will be able to understand the working of AVRDUDE for uploading the file to our microcontroller. Later, we will integrate this tool to our Code::Blocks to eliminate the manual Command Prompt steps for our ease. Let's begin the interesting part of this lab.

a) Uploading code to ATmega328P microcontroller using AVRDUDE through Command Prompt

1) Open the Command Prompt by typing **cmd** in the Windows search.
2) First change the current directory to the folder path where **Project1.hex** file is saved.
   For example: **C:\Users\Aiman\Desktop\ES Labs\Project1\bin\Debug**
   To do this, use **cd command**

```
cd C:\Users\Aiman\Desktop\ES Labs\Project1\bin\Debug
```

3) Now, type **avrdude** to verify that it is recognized (as has already been done). It display a list of options available for usage with avrdude command. Read the description written with each.
4) For now, we will use only the required ones and the details of which are given below. Note that these are case sensitive.

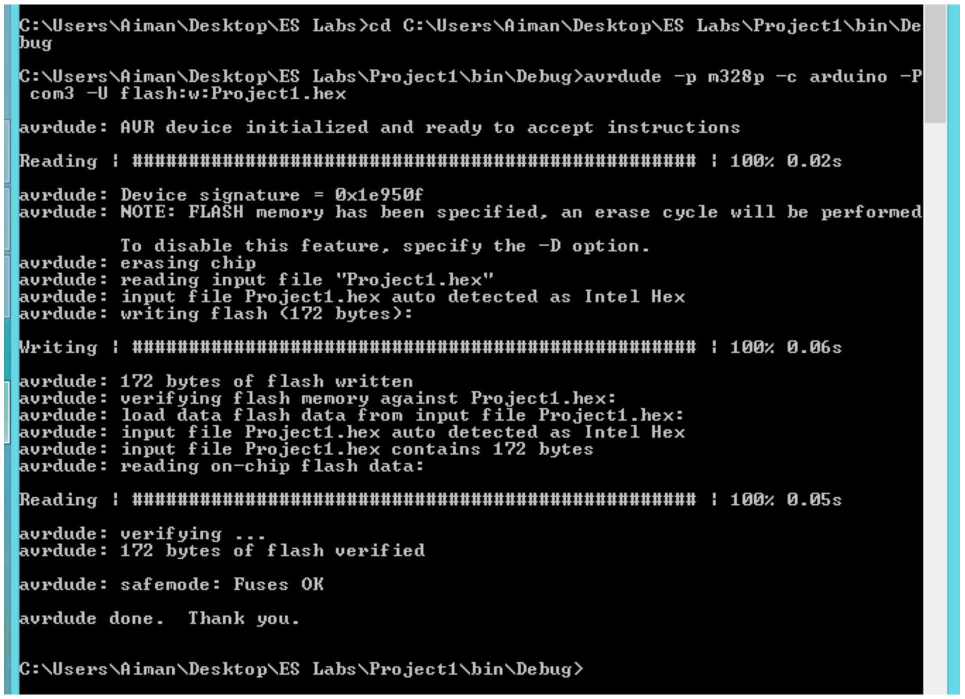
Table 1: ***avrdude*** Usage Options and Description

Options	**Description**	**Example as applicable to the test case**
-p	Part No. (To specify the AVR device)	In our case, it is **m328p** (Atmega 328p)
-P	Port (To specify the Connection port)	In our case, it is COM3
-c	Programmer (To specify Programmer Type)	In our case, it is Arduino
-U	Memory operation specification. Required format is: **<memory type>:w:<file name>** Where, w shows read the specified file and write it to the specified device memory	In our case, **flash** is the memory type where we want to write the code saved in the generated **Project1.hex** file

5) Based on the above, write **(type, don't copy-paste)** the following command in the Command Prompt.

```
avrdude –p m328p –P com3 –c arduino –U flash:w:Project1.hex
```

It will display some messages shown in Figure below. For successful upload of the hex file, you will see the LED blinking at the specified rate.



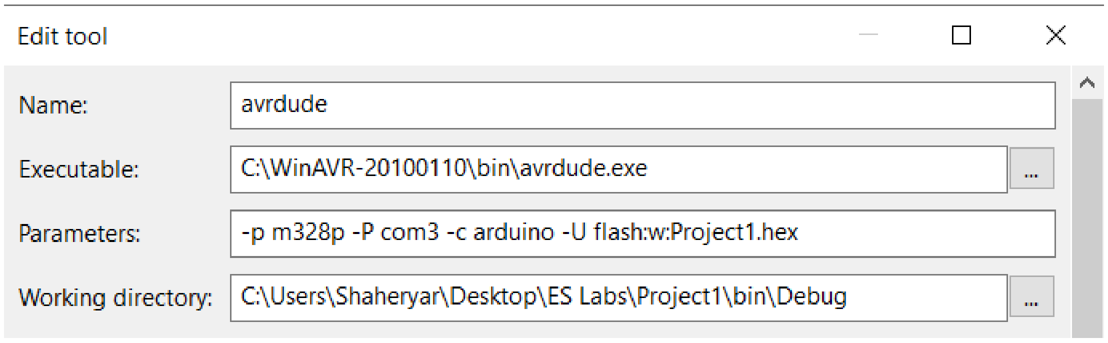
```
C:\Users\Aiman\Desktop\ES Labs>cd C:\Users\Aiman\Desktop\ES Labs\Project1\bin\De
bug

C:\Users\Aiman\Desktop\ES Labs\Project1\bin\Debug>avrdude -p m328p -c arduino -P
 com3 -U flash:w:Project1.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.02s

avrdude: Device signature = 0x1e950f
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed

         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "Project1.hex"
avrdude: input file Project1.hex auto detected as Intel Hex
avrdude: writing flash (172 bytes):

Writing | ################################################## | 100% 0.06s

avrdude: 172 bytes of flash written
avrdude: verifying flash memory against Project1.hex:
avrdude: load data flash data from input file Project1.hex:
avrdude: input file Project1.hex auto detected as Intel Hex
avrdude: input file Project1.hex contains 172 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.05s

avrdude: verifying ...
avrdude: 172 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done.  Thank you.


C:\Users\Aiman\Desktop\ES Labs\Project1\bin\Debug>
```

Figure 40: Uploading the **.hex** File to ATmega328P through Command Prompt **avrdude**

b) Uploading code to ATmega328P microcontroller using Code::Blocks tool

To avoid the time-consuming and intimidating Command Prompt interface, we will now add the AVRDUDE tool to Code::Blocks. Follow the following steps:

1) In your Code:Blocks IDE, click on **Tools > Configure Tools > Add.** In the **Edit Tool** window, set the Name, Executable, Parameters and Working Directory as shown in Figure. To set the executable browse the specified path of WinAVR folder, and select avrdude.exe. Note, that in the parameters option, we have written the same avrdude command used earlier. The working directory is set to be the one containing the .hex file. Click OK and close the Tools Window.



Name: avrdude
Executable: C:\WinAVR-20100110\bin\avrdude.exe
Parameters: -p m328p -P com3 -c arduino -U flash:w:Project1.hex
Working directory: C:\Users\Shaheryar\Desktop\ES Labs\Project1\bin\Debug

Figure 41: Required Tool Settings

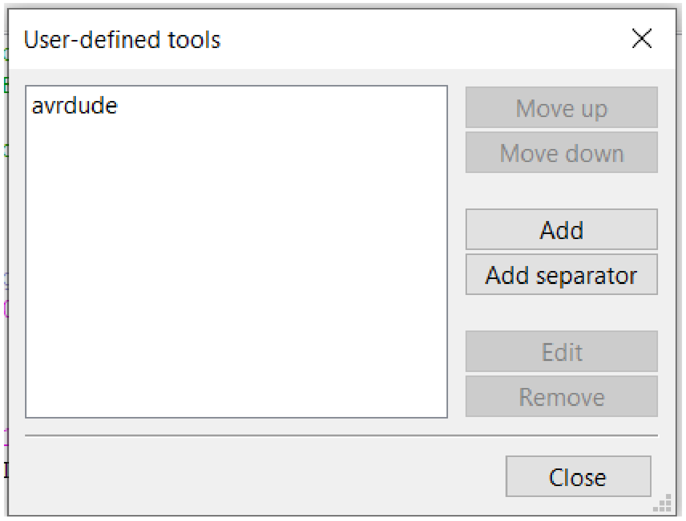
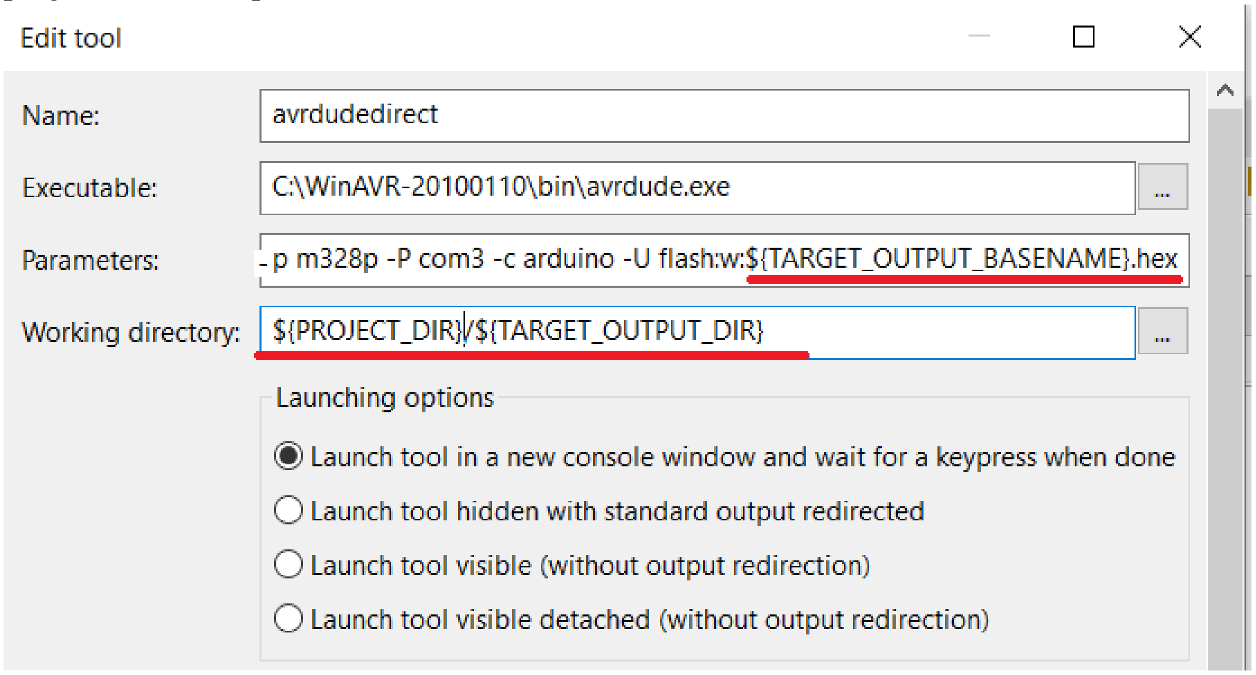


Figure 42: Added Tool **avrdude**

2) Now, select Tools from the top menu bar once again. The name of tool just added for example: **avrdude** will be available now above the Configure Tools option. Click the **avrdude** and that's it. The Log Window below will show the execution. Once the code is uploaded to ATmega328P, you will observe the blinking LED.
3) **To make it more generalized, we will now add a tool using macros. Here, instead of specifying the exact file and project names and path, we will use macros that will do the job and you won't have to type it again and again for each of your projects.**

4) **Go to Tools> Configure Tools. Either Edit the previous tool or use Add to add another one. This time set the Parameters and Working Directory in terms of macros replacing project name and path. Click OK.**



Name: avrdudedirect

Executable: C:\WinAVR-20100110\bin\avrdude.exe

Parameters: -p m328p -P com3 -c arduino -U flash:w:${TARGET_OUTPUT_BASENAME}.hex

Working directory: ${PROJECT_DIR}/${TARGET_OUTPUT_DIR}

Launching options

- [✓ Launch tool in a new console window and wait for a keypress when done]
- Launch tool hidden with standard output redirected
- Launch tool visible (without output redirection)
- Launch tool visible detached (without output redirection)

Figure 43:Adding avrdude with Generic Parameters and Working Directory

5) Now, modify the delay in your main.c file by replacing 1000 to 5000 for 5sec delay. Save it and Rebuild the file using the **Build> Rebuild** option. The new file will replace the previously generated .hex file.
6) Now, upload this one by selecting the newly added **Tool** through **Tools>avrdudedirect. Verify the successful upload by blinking of LED at 5msec delays.**

Congratulations! You have successfully created, built and tested your first AVR project on ATmega328P using Code::Blocks with AVR Toolchain.

LAB SESSION 02

OBJECTIVE:

To program the AVR ATmega328P I/O (Input/Output) ports for digital input and output

LAB OUTCOMES:

By the end of the lab, you would be able to:

1) Identify the AVR ATmega328P I/O ports
2) Utilize DDR (Direct Data Registers) for setting direction of ports
3) Take digital input and set digital output through I/O ports
4) Build required logics for digital inputs and outputs in pure C-language codes and test them on ATmega328P microcontroller
5) Utilize logical operations for bit-wise manipulation of ports

---

"There are exactly 10 types of people in the world.

Those who understand binary numbers and those who don't."

If this doesn't make sense to you, you need to brush up your number system's knowledge which is a pre-requisite to the course.

BACKGROUND:

ATmega328P Pin Diagram and I/O Ports

Figure 1 shows ATmega328P pin diagram. Here, we can see total 28 pins (14 at each side). Throughout the labs, we will be exploring functions of these pins and their utilization. Most port pins are multiplexed with alternate functions for the peripheral features on the device i.e. they have dual roles. Note that enabling the alternate function of some of the port pins does not affect the use of the other pins in the port as general digital I/O. However, **in this lab, we will focus on the pins that can be utilized for digital input and output i.e. simple I/O function, so** don't get intimidated by the pin diagram shown in Figure 1.

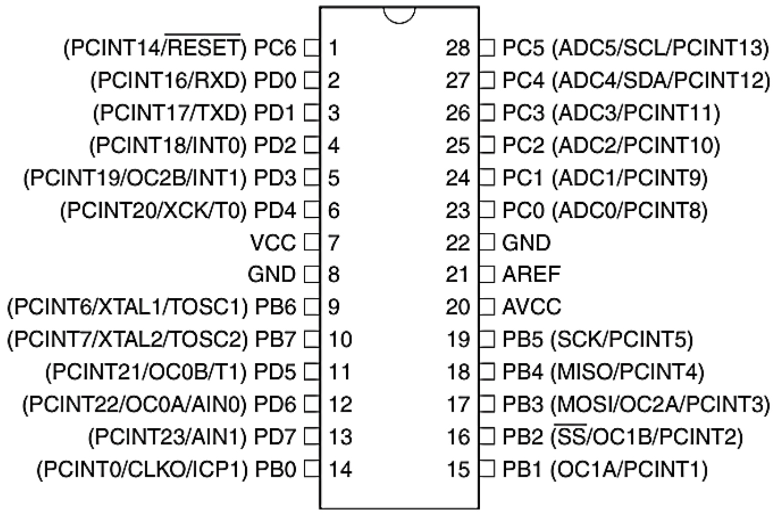


Figure 1: Pin Diagram of ATmega328P

Ports as General Digital I/O Pins

Digital I/O pins on the AVR microcontroller are grouped into ports. Each port has up to eight pins assigned to it. However, each pin can be individually configured. So, you can have a mix of input and output pins on the same port.

The ATmega328P has 23 General Purpose Digital I/O Pins assigned to 3 GPIO Ports (8-bit Ports B, D and 7-bit Port C).

Ports are designated by a letter and pins are numbered starting at 0. For example, the first pin on port B is named PB0 and the third pin on port D is named PD2. The ports are bi-directional I/O ports. The pin driver is strong enough (20mA) to drive LED displays directly.

Let's consider Figure 2 which shows the names and locations of the pins on the ATmega328P for the DIP package, as well as the corresponding locations on the Arduino Uno board. For those coming from an Arduino background, there are a couple things to take note. Firstly, you may notice that not all the digital I/O pins are available for use on the Arduino Uno board. This is because these pins are being used for alternate functions. PC6 is being used for device reset and PB6 and PB7 are connected to the crystal on the board. Secondly, even though the analog pins A0 - A5 can be used with the Analog-to-Digital converter, they may also be used for digital I/O.

Now that we know the names of the pins and where they are located, we will learn how to configure them using pure C-programming.

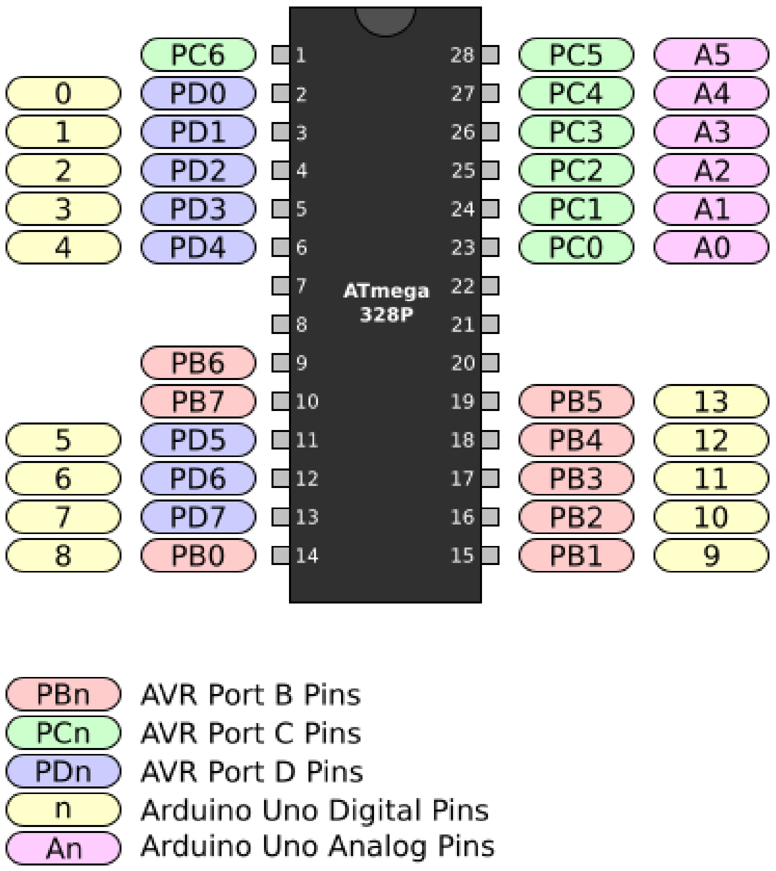


Figure 2: Pins on the ATmega328P with the corresponding locations on the Arduino Uno board.

Configuring the Pin

Each port has three I/O registers associated with it, one each for the Data Register – PORTx, Data Direction Register – DDRx, and the Port Input Pins – PINx. Also notice that each of the I/O registers is 8 bits wide, and each port has a maximum of 8 pins; therefore each bit of the I/O registers affects one of the pins. Consequently, each port pin consists of three register bits: DDxn, PORTxn, and PINxn.

The Port Input Pins I/O location is read only, while the Data Register and the Data Direction Register are read/write.

All registers and bit references in this section are written in general form. A lower case “x” represents the numbering letter for the port, and a lower case “n” represents the bit number. However, when using the register or bit defines in a program, the precise form must be used. For example, PORTB3 for bit no. 3 in Port B, here documented generally as PORTxn.

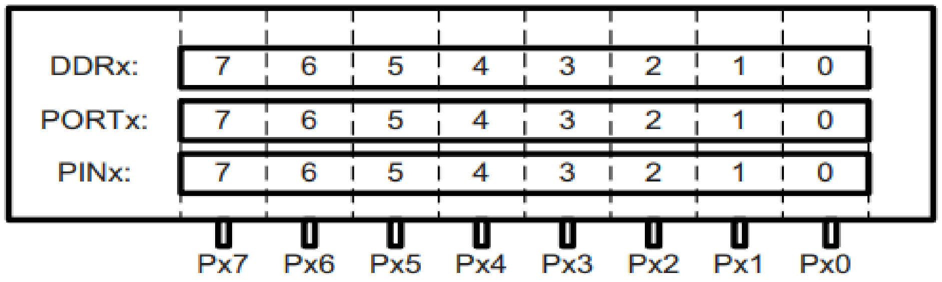


Figure 3: Relations between the Registers and the Pins of AVR

1) Role of DDRx (Data-Direct Register)

The DDRx I/O register is used solely for <u>the purpose of making a given port an input or output port</u>. To make any pin of the port an output pin, we write 1 to the corresponding bit of DDRx register. It must be noted that unless we set the DDRx bits to one, the data will not go from the port register to the pins of the AVR.

To output data to all of the pins of the Port B, we first put 0b11111111 (0xFF) into the DDRB register to make all of the pins output. To make a port an input port, we must first put 0s into the DDRx register for that port, and then bring in (read) the data present at the pins.

Notice that upon reset, all ports have the value 0x00 in their DDR registers. This means that all ports are configured as input.

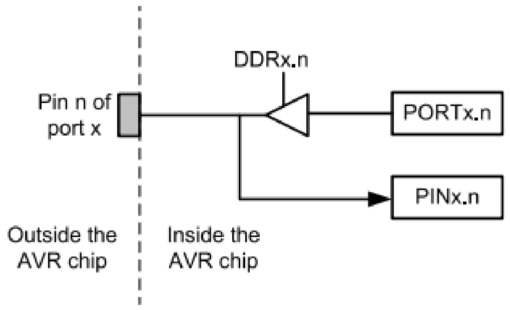
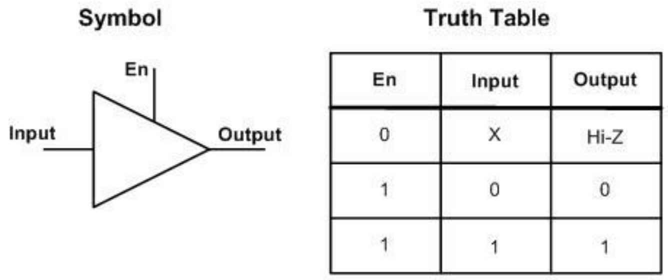


Figure 4: The I/O Port in AVR



En	Input	Output
0	X	Hi-Z
1	0	0
1	1	1

Figure 5: Buffer (DDRx.n is used an Enable pin in Figure 4)

2) Role of PINx (Pin-Register)

To read the data present at the pins, we use PIN register. It must be noted that to bring data into CPU from pins we read the contents of the PINx register.

3) Role of PORTx (Data-Register)

The PORTx register is used to send data out to pins.

Why program the AVR in C?

Compilers produce hex files that we download into the Flash of the micro-controller. The size of the hex file produced by the compiler is one of the main concerns because microcontrollers have limited on-chip Flash.

While Assembly language produces a hex file that is much **smaller than C**, programming in Assembly language is **often tedious and time consuming**. On the other hand, C programming is less time consuming and much easier to write, but the hex file size produced is much larger than if we used Assembly language. The following are some of the major reasons for writing programs in C instead of Assembly:

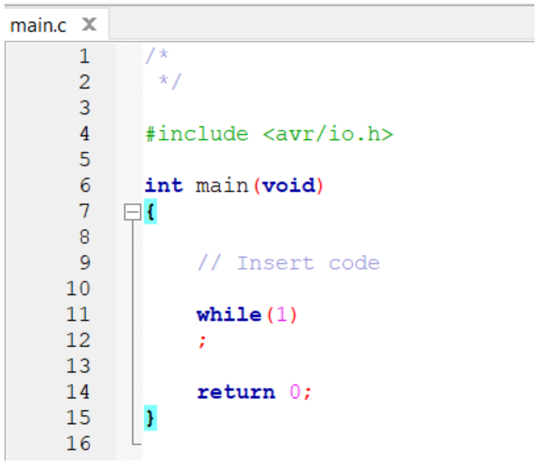
1) It is easier and less time consuming to write in C than in Assembly.
2) C is easier to modify and update.
3) You can use code available in function libraries.
4) C code is portable to other microcontrollers with little or no modification.

As seen in the last lab, we have used WinAVR GNU GCC Compiler for AVR.

Starting AVR Programming in C

This section provides a revision of basic syntax of C language, its data types, and functions which we will be needing to build our logic for AVR programming.

When we create an AVR project in Code::Blocks, we get an empty **main.c** file with a basic template shown in Figure 6.



```c
/*
 */

#include <avr/io.h>

int main(void)
{

    // Insert code

    while(1)
    ;

    return 0;
}
```

Figure 6: main.c Template Code

Comments:

//Single line comment

/* Multiple line
        Comment */

Header files:

The #include is a preprocessor directive that instructs the compiler to find the file in the < > brackets and tack it on at the head of the file you are about to compile. The **io.h** provides appropriate I/O definitions for the device we are using, and the **delay.h** provides the definitions for the delay function.

Statements control the program flow and consist of keywords, expressions, and other statements. A semicolon ends a statement.

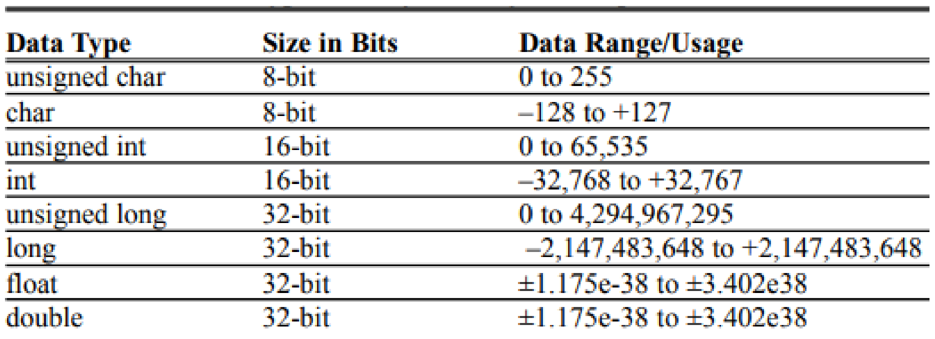
main ( ): All C programs contain the **main( )** function that contains the code and is first run when the program begins. main (void) means the function doesn't take any input. '**int main**' means that the function needs to return some integer at the end of the execution and we do so by returning 0 at the end of the program.

While Loop: It is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement. The while(1) will run the loop forever because '1' is the definition of true (false is defined as 0).

Data Types: A good understanding of C data types for the AVR can help programmers to create smaller hex files. In declaring variables, we must pay careful attention to the size of the data type and data range, refer to the Table 1.

Remember that C compilers use the signed char as the default unless we put the keyword unsigned in front of the char. In many situations, such as setting a counter value, where there is no need for signed data, we should use the unsigned char instead of the signed char. Using the ***int*** instead of the ***unsigned char*** leads to the need for more memory space.

Table 1: Data types widely used by C compilers



Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
char	8-bit	–128 to +127
unsigned int	16-bit	0 to 65,535
int	16-bit	–32,768 to +32,767
unsigned long	32-bit	0 to 4,294,967,295
long	32-bit	–2,147,483,648 to +2,147,483,648
float	32-bit	±1.175e-38 to ±3.402e38
double	32-bit	±1.175e-38 to ±3.402e38

Declaring (Creating) Variables: type variableName = value;

Where type is one of C types (such as int), and variableName is the name of the variable (such as x or myName). The equal sign is used to assign a value to the variable.

Delay functions: One way of generating time delay is to use predefined functions such as _delay_ms( ) and _delay_us( ) defined in delay.h in WinAVR For this, we need to include the header file delay.h For example: #include <utils/delay.h>

Constants: Data that cannot be changed by the program. By convention, constants are named in capital letters. These are defined at the start or in header files, then can be used anywhere in the code.

# define PI 3.1415926

AVR I/O Programming in C

All port registers of the AVR are both byte accessible and bit accessible. Let's first discuss and practice the byte-size I/O.

Byte size I/O

To access a PORT register as a byte, we use the PORTx label where x indicates the name of the port. The data direction registers are accessed using DDRx to indicate the data direction of port x. To access a PIN register as a byte, we use the PINx label where x indicates the name of the port.

Writing C-Code for Giving Output from ATmega328P Pins

Considering the above discussion, let's check how we can output digital high or low signal from ATmega328P I/O pins.

Example 1-Output through I/O Port to drive LEDs:

Consider the port D which has 8 I/O pins. Let's set the first 4 pins (D0, D1, D2 and D3) as output pins. This is done by first setting the DDR register of port D as 0b00001111.

DDRD= 0b00001111 (or, DDRD = 0X0F)

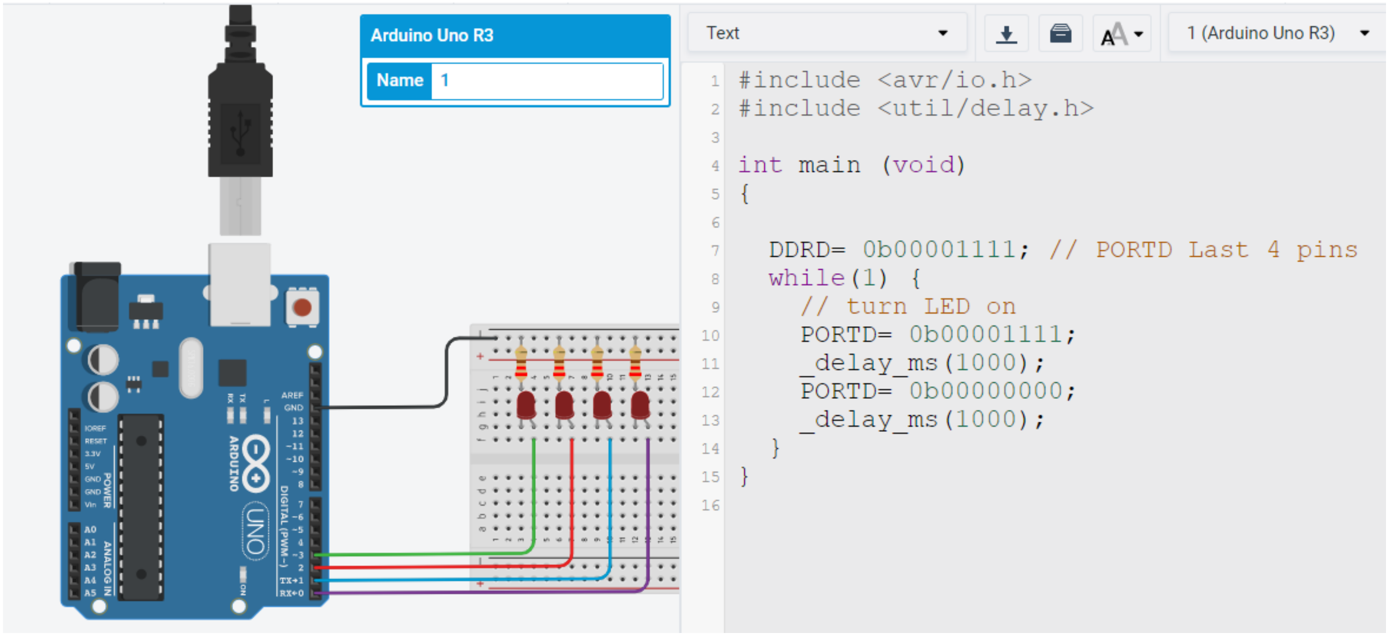
Once the port pins are set as output pins, now we can output logic high or low at these pins. Here, we will set the Port register's corresponding last 4-bits high.

PORTD= 0b00001111

Similarly, to set the alternate bits (D0 and D2) low, we can use;

PORTD = 0b00001010 (or, PORTD = 0X0A in Hex)

The complete code is shown in Figure with corresponding connections. Note that, the DDR registers are set outside the while loop in the main function once.



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

int main (void)
{

  DDRD= 0b00001111; // PORTD Last 4 pins
  while(1) {
    // turn LED on
    PORTD= 0b00001111;
    _delay_ms(1000);
    PORTD= 0b00000000;
    _delay_ms(1000);
  }
}
```

Figure 7: Code and Connections for Example 1

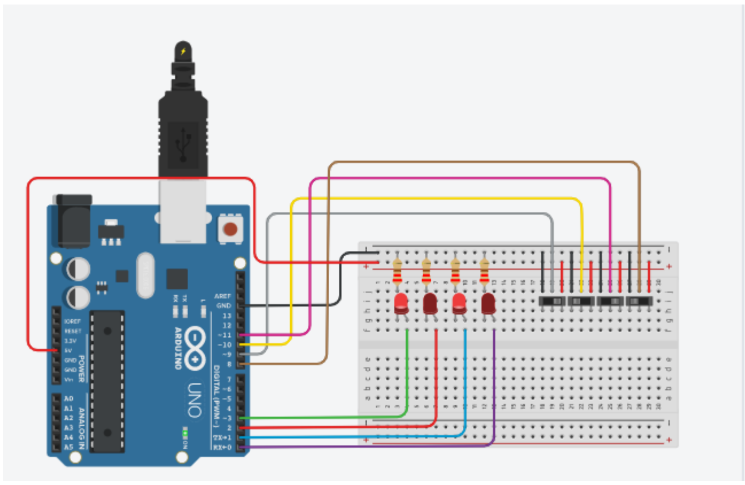
Writing C-Code for Taking Input to ATmega328P Pins

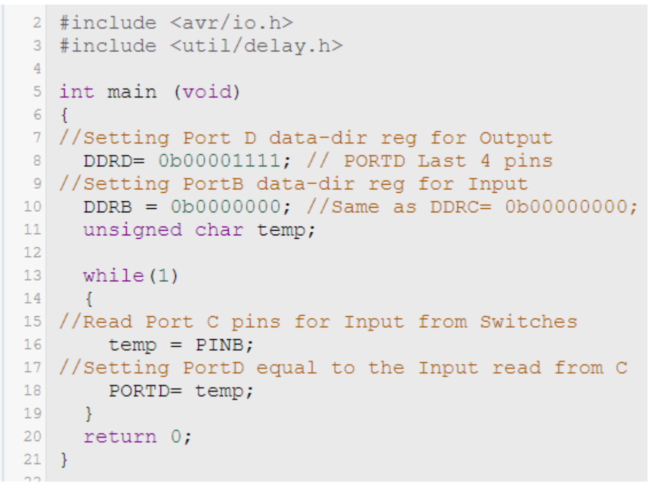
Considering the port B which has 8 pins. Let's set the pin PB0, PB1, PB2 and PB3 of port B as input pins. To take input, we need to set the DDRB port first and then we can use the PINB for reading from the port B pins.

DDRB= 0X00 // Setting all pins as input pins

temp = PINB //reading from PINB and saving it in **temp** variable

Example 2 – Taking Input through I/O Ports from Switches:





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

int main (void)
{
//Setting Port D data-dir reg for Output
  DDRD= 0b00001111; // PORTD Last 4 pins
//Setting PortB data-dir reg for Input
  DDRB = 0b0000000; //Same as DDRC= 0b00000000;
  unsigned char temp;

  while(1)
  {
//Read Port C pins for Input from Switches
    temp = PINB;
//Setting PortD equal to the Input read from C
    PORTD= temp;
  }
  return 0;
}
```

Figure 8: Code and Connections for Example 2

Bit-wise Logic Operations in C for I/O Bit Manipulation

One of the most important and powerful features of the C language is its ability to perform bit manipulation. You might be familiar with the logical operators AND (&&), OR (||), and NOT (!), but might be less familiar with the bit-wise operators AND (&), OR (|), EX-OR (^), inverter (~), shift right (>>), and shift left (<<). These bit-wise operators are widely used in software engineering for embedded systems and microcontroller-based system design and interfacing.

Masking for Bit Size I/O

We use these bit-wise logical operations to access a single bit of a given register without disturbing the rest of the byte. In this section you will see how to mask a bit of a byte.

Initially, the output port PORTD is set as;

PORTD = 0b 00000 0000; // All bits are clear

To set the 5th bit high without disturbing the other bits, we can perform bit-wise OR | operation of PORTD register's previous value with 0b00010000.

PORTD = PORTD | 0b00010000; // To set the 5th bit (bit # 4) only

Similarly, if PORTD is initially set as 0b11111111, and we just want to clear the 4th bit (Bit #3) then we can perform a bit-wise AND operation.

PORTD= PORTD & 0b111110111;

Irrespective of the value of a bit, the AND operation of the bit with 1 results in the previous value of the bit i.e. it remains unchanged. Whereas, AND operation of a bit with 0 clears the bit.

Similarly, the OR operation of a bit with 1, results in setting the bit high. However, OR operation with 0 leaves the bit unchanged.

Table 2: Bit-wise Logical Operators in C

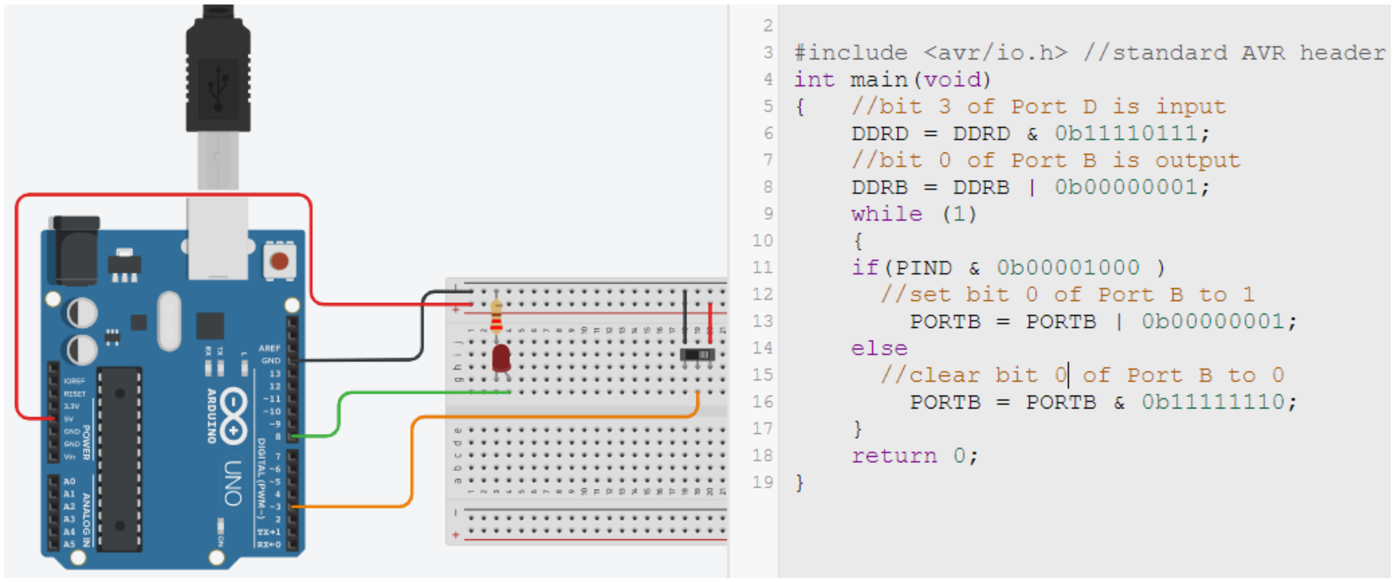
		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

To invert all bits, the bit-wise NOT (~) operator can be used.

PORTD = ~PORTD;

Example 3 – Using logical operators for bit-wise I/O from ATmega328P ports

The following examples get the status of bit 3 of Port D and send it to the bit 0 of port D continuously.



```
#include <avr/io.h> //standard AVR header
int main(void)
{    //bit 3 of Port D is input
     DDRD = DDRD & 0b11110111;
     //bit 0 of Port B is output
     DDRB = DDRB | 0b00000001;
     while (1)
     {
     if(PIND & 0b00001000 )
       //set bit 0 of Port B to 1
         PORTB = PORTB | 0b00000001;
     else
       //clear bit 0 of Port B to 0
         PORTB = PORTB & 0b11111110;
     }
     return 0;
}
```

Figure 9: Code and Connections for Example 3

Compound Assignment Operators

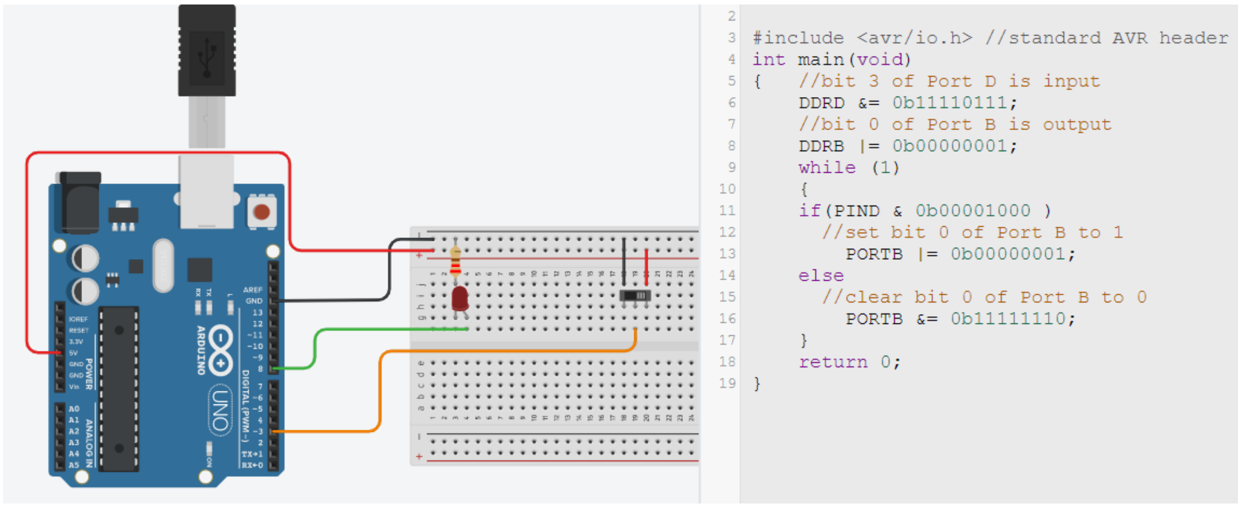
To reduce coding (typing) we can use compound statements for bit-wise operators in C.

Table 3: Compound Assignment Operators

Operation	Abbreviated Expression	Equal C Expression
And assignment	a &= b	a = a & b
OR assignment	a |= b	a = a | b

Example 4 – Using compound assignment operators for bit-wise I/O from ATmega328P ports

The Example 3 is re-written using the compound bit-wise operators. Observe the difference. Note that the compound assignment operators '|=' don't have a space '| =' in between.



```
#include <avr/io.h> //standard AVR header
int main(void)
{   //bit 3 of Port D is input
    DDRD &= 0b11110111;
    //bit 0 of Port B is output
    DDRB |= 0b00000001;
    while (1)
    {
    if(PIND & 0b00001000 )
      //set bit 0 of Port B to 1
        PORTB |= 0b00000001;
    else
      //clear bit 0 of Port B to 0
        PORTB &= 0b11111110;
    }
    return 0;
}
```

Figure 10: Code and Connections for Example 4

Shift Operation for Bit-Manipulation:

To do bit-wise I/O operation in C, we need numbers like 0b00100000 in which there are seven 0s and one 1. Only the position of the one varies in different programs. To leave the generation of ones and zeros to the compiler and improve the code clarity, we use shift operations. For example, instead of writing "0b00100000" we can write "0b00000001 << 5" or we can write simply "1<<5". Sometimes we need numbers like 0b11101111. To generate such a number, we do the shifting first and then invert it. For example, to generate 0b11101111 we can write ~(1<<4).

LAB TASKS

From Task 2 to 5, create AVR projects and test them on ATmega328P. Feel free to use different conditional statements, loops, switch-case structure to complete the C-programming related tasks.

TASK 1: Explain what makes the Blinky.c blink?

The code we tested in Lab 01 is given below. Explain what makes this Blinky.c blink?

```
//Blinky.c from Lab01
#include <avr/io.h>
#define BLINK_DELAY_MS 1000
#include <util/delay.h>
int main (void)
{
 DDRB |= 0B100000;
 while(1)  {
  PORTB |= 0B100000;
  _delay_ms(BLINK_DELAY_MS);
  PORTB &= ~ 0B100000;
  _delay_ms(BLINK_DELAY_MS);  }
}
```

TASK 2: To check and indicate the status of a sensor using the specified ports and bits of ATmega328P

A door sensor (here, assume the switch) is connected to pin 1 of Port B, and an LED is connected to pin 5 of Port C. Write an AVR C program to monitor the door sensor and, when it opens, turn on the LED.

TASK 3: To use the general I/O pins of ATmega328P as input or output pin based on the given condition

Write an AVR C program to monitor bit 7 of Port B. If it is 1, make bit 4 of Port B input; otherwise, change pin 4 of Port B to output.

TASK 4: To control the specified pins of a given port without disturbing the rest of the pins

Write an AVR C program to control a set of 8 LEDs connected to port D such that the first 4 LED glow when input from a switch is high, and remain off when the input from switch is low. The remaining 4 LED toggle continuously without disturbing the rest of the pins of port D.

TASK 5: To control the output based on combination of 2 input pins

Write an AVR C program to read pins 0 and 1 of Port B and update the LEDs at pin 0, 1 & 2 of Port D according to the following logic. You can use switch-case structure.

Input Port B [1:0] Status	**Output Port D [2:0] Status**
0b 00	0b 000
0b 01	0b 011
0b 10	0b 101
0b 11	0b 111

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**
Laboratory Session No.: ________________________ Date: ________________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)

Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: <u>Imitate</u> and <u>practice</u> given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to <u>operate</u> software environment <u>under supervision</u>, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: <u>Detect</u> Errors/Exceptions and <u>manipulate,</u> under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: <u>Copy</u> or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 03

OBJECTIVE:

To program the ATmega328P for reading analog input through its Analog-to-Digital Converter ADC module

LAB OUTCOMES:

By the end of the lab, you would be able to:

1) Identify the AVR ATmega328P pins associated with the ADC module
2) Identify the purpose of different bits of ADC registers and utilize them for their set purposes like setting reference voltage, selecting source of analog input, and indicating start / end of conversion
3) Take analog input through the ADC pins and indicate its digital equivalent
4) Build required logics for reading analog input in pure C-language codes and test them on ATmega328P microcontroller
5) Verify the analog to digital conversion ADC of microcontroller pins by testing the digital output through external DAC (digital-to-analog) R-2R circuit / DAC0808 IC

BACKGROUND:

Digital computers use binary (discrete) values, but in the physical world the signals are analog (continuous). Most physical variables are analog in nature and can take on any value within a continuous range of values. Temperature, pressure, humidity, and velocity are a few examples of physical quantities that we deal with every day. For acquisition of analog signals, we need Analog-to-Digital (ADC) module for conversion. Microcontrollers are therefore generally featured with ADC module. In this lab, we will explore ATmega328P ADC input channels that enable us to capture analog signals.

Basics of Analog-to-Digital Conversion (ADC)

An analog-to-digital converter takes an analog input voltage and after a certain amount of time produces a digital output code which represents the analog input. ADC involves the following steps:

- **Sampling:** Sampling is the processes of converting continuous- time analog signal into a discrete-time signal by taking the “samples” at discrete-time intervals. Sampling analog signals makes them discrete in time but still continuous valued. Sampling frequency determines the intervals at which samples are taken. Nyquist criterion requires that the sampling frequency be at least twice the highest frequency contained in the signal.
- **Quantization:** Quantization is the process of mapping continuous infinite values to a smaller set of discrete finite values. The quantization step size is the smallest possible difference in amplitude between samples.
- **Encoding:** After quantization, each quantization level is assigned a unique binary code.

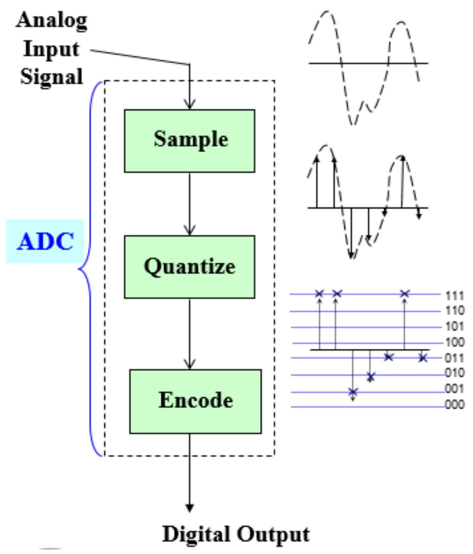


Figure 1: Analog-to-Digital Conversion

Relation between the Number of Bits, Resolution and Reference Voltage

The quantization levels are dependent on the number of bits available or required for representing the digital output. ***'n'*** is the number of bits, then the quantization levels are 2^n **(0 to** 2^{n-1}**).** Consequently, **t**he ADC has n-bit resolution. Higher-resolution ADCs provide a smaller ***step size***, where step size is the smallest change that can be discerned by an ADC. The resolution is dependent on reference voltage as well. The number of bits and reference voltage decide the step-size. This is related as:

$$Step\ size = \frac{V_{ref}}{2^n}$$

Table 1: Reference Voltage Relation with Resolution for 8-bit and 10-bit ADC

Reference Voltage	Step-size for 8-bit ADC	Step-size for 10-bit ADC
5V	5/256 = 19.53 mV	5/1024= 4.88 mV
3.3 V	3.3/256 = 12.89 mV	3.3/ 1024 = 3.22 mV
1.1 V	1.1/256 = 4.297 mV	1.1/1024 = 1.074 mV

- Range of analog input voltage for ADC: 0 to Vref.

In an 8-bit ADC we have an 8-bit digital data output of D0–D7, while in the 10-bit ADC the data output is D0–D9.

The obtained digital output D_{out} is related to the input analog voltage V_{in} as:

$$D_{out} = \frac{V_{in}}{Step - size}$$

D_{out} is the decimal equivalent of n-bit binary result.

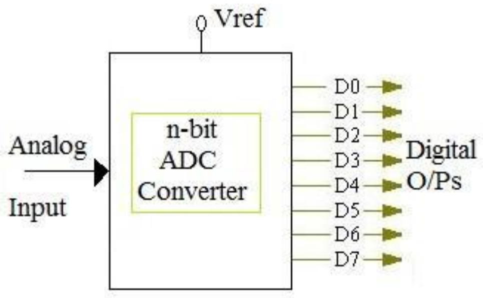


Figure 2: 8-bit ADC Representation

ADC (Analog-to-Digital Conversion) Module of ATmega328P

ATmega328P microcontroller ADC module capable of converting an analog voltage into a 10-bit number from 0 to 1023 or an 8-bit number from 0 to 255. There are 6 ADC input channels on the chip as shown in Figure 3 (pin # 23 to 28 = ADC0 to ADC5). The input to the module can be selected from any one of the 6 inputs on the chip i.e., one channel can be converted at a time. The inputs to the ADC module appear on the Arduino board as connections A0 through A5.

In Figure 3, you can see a few pins, other than ADC0 to ADC5, marked with Green color, representing their connection with analog related circuitry.

The function of different pins for utilization of the ADC module is explained with their associated registers in the following sections.

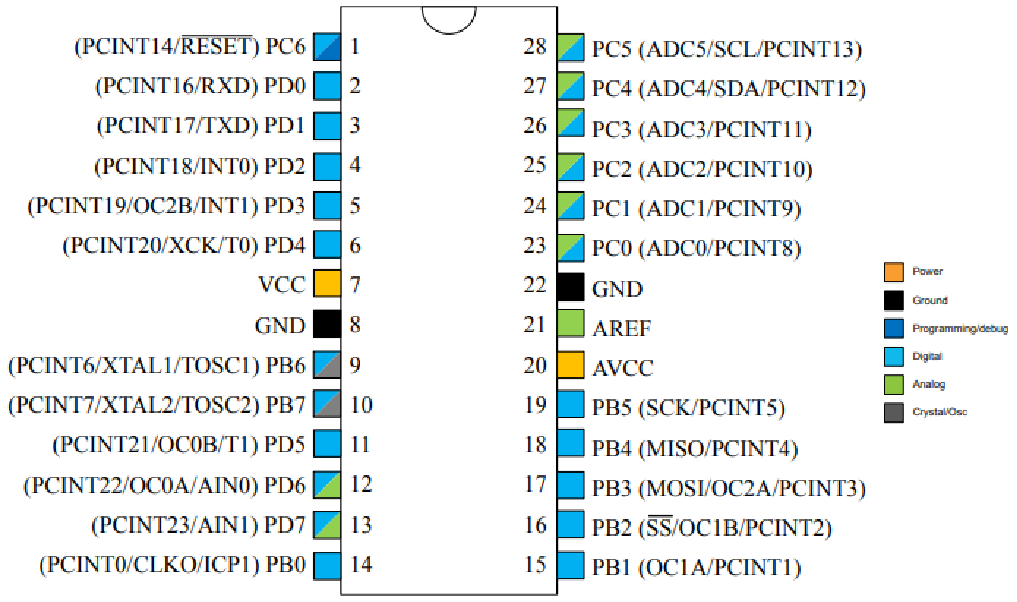


Figure 3: ATmega328P pin diagram

Pins for ATmega328P ADC

Table 2: ATmega328P pins related to the analog circuitry

Pin #	Port	ADC Module – Pin Names	Function
23	PC0	ADC0	ADC Input Channel 0
24	PC1	ADC1	ADC Input Channel 1
25	PC2	ADC2	ADC Input Channel 2
26	PC3	ADC3	ADC Input Channel 3
27	PC4	ADC4	ADC Input Channel 4
28	PC5	ADC5	ADC Input Channel 5
21	-	AREF	External voltage supply for setting reference voltage. By connecting a capacitor between the AREF pin and GND, reference voltage becomes more stable and increases the precision of ADC
20	-	AVCC	Provides the supply for analog ADC circuitry.
12	PD6	AIN0	Analog Comparator Positive & Negative Input- AIN0 & AIN1. These pins are associated with **analog comparator module.** These are not necessarily needed for ADC.
13	PD7	AIN1	

Registers for AVR ADC Programming

Five major registers are associated with the ADC module for interfacing it. Let's examine each one-by-one. The description given here is quite brief. You can refer to the datasheet for further details.

- ADCH (high data)
- ADCL (low data)
- ADCSRA and ADCSRB (ADC Control and Status Registers A & B)
- ADMUX (ADC multiplexer selection register)
- DIDR0

The name of register is written in capital letters. Each register is 8-bit wide. Indexing is done to show different bits, for example XYZ [2:0] means the least significant 3 bits of the register XYZ (XYZ2, XYZ1 & XYZ0). XYZ[2:0] = 5 would mean (XYZ2 =1, XYZ1= 0 and XYZ0 =1 since 5 = 0b101).

1) ADC Data Register Low and High Byte

After the A/D conversion is complete, the result is stored in registers **ADCL** (A/D Result Low Byte) and **ACDH** (A/D Result High Byte). For 10-bit ADC result, the eight bits sit in one 8-bit register and the remaining two bits are provided in the other register, with six bits being unused. The result can be left or right adjusted as shown below. If only eight bits of resolution are needed, the ADC value is left-justified and the high-order byte are read through ADCH.

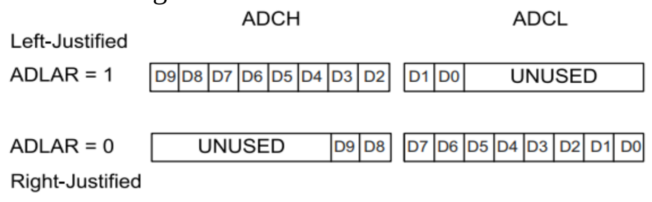


Figure 4: 10-bit ADC result adjustment in ADCH and ADCL

2) ADMUX (ADC Multiplexer Selection Register)

It is an 8-bit register with bits illustrated below.

Bit	7	6	5	4	3	2	1	0
	REFS[1:0]		ADLAR	▨	MUX[3:0]			

- **Reference Selector Bits (REFS [1:0])** The ADMUX [7:6] bits select the voltage reference for the ADC.

Table 3: Function of the reference selector bits

ADMUX[7:6] = REFS[1:0]	Reference High Voltage Selection	Description
00	AREF	Voltage provided at AREF pin externally (Internal Vref turned OFF)
01	AVCC	AVCC with external capacitor at AREF pin. Note: Arduino already has capacitor placed on line
11	Internal 1.1V	Internal 1.1 V reference fixed regardless of VCC. Note: Arduino already has capacitor placed on line

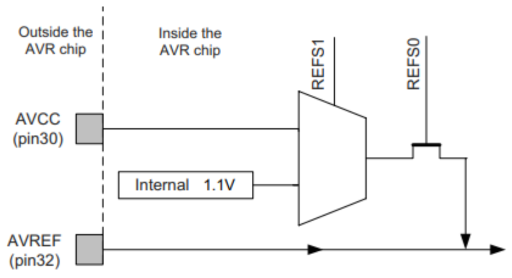


Figure 5: Reference voltage selection

- **ADC Left Adjust Result (ADLAR)** ADMUX [5] is called ADLAR. The ADLAR bit affects the presentation of the ADC conversion result in the ADC data register.

Table 4: ADLAR bit values for result adjustment

ADLAR	Conversion Result
0	Right adjusted for 10-bit result
1	Left adjusted for 8-bit result

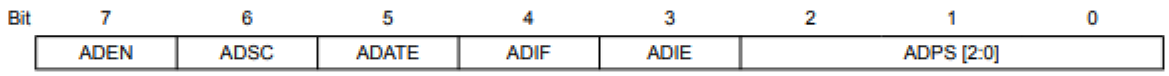
- **Analog Channel Selection (MUX[3:0])** The 4 bits ADMUX[3:0] serve as selector for the multiplexer that selects one of the 6 analog input channels to be connected to ADC.

Table 5: Input Channel Selection for ADC

MUX [3:0]	Input Channel
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
1000	Temperature Measurement

Interesting Fact: The Atmega328P has an internal temperature sensor. Refer to the datasheet and read about it. The output of the temperature sensor can be selected as one of the inputs for ADC module as shown in the description of ADMUX Channel Selection bits.

3) ADCSRA (ADC Control and Status Register A)



Bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS [2:0]		

Table 6: Function of ADCSRA Register Bits

Fields	Full-Form	ADSCRA Bit No.	Function for Different Values
ADEN	**ADC Enable**	ADSCRA[7]	1: ADC is enabled 0: ADC is turned off
ADSC	**ADC Start Conversion**	ADSCRA[6]	1: To start each conversion in Single Conversion mode Returns to 0 when conversion is complete.
ADATE	**ADC Auto Trigger Enable**	ADSCRA[5]	It is set to 1 for enabling auto-triggering of ADC through the selected trigger signal. The ADC tested in this lab is not external interrupt-driven. It is set 0 for single-conversion mode.
ADIF	**ADC Interrupt Flag**	ADSCRA[4]	This bit is set when an ADC conversion completes and the data registers are updated. It is used to identify completion of conversion.
ADIE	**ADC Interrupt Enable**	ADSCRA[3]	Not needed for now. It is used to activate ADC conversion complete interrupt.
ADPS	**ADC Pre-scalar Select**	ADSCRA[2:0]	These bits determine the division factor between the system clock frequency and the input clock to the ADC. 50kHz to 200kHz is acceptable for ADC circuitry System clock frequency for Arduino UNO is 16MHz (Crystal oscillator connected with UNO).

ADPS[2:0]- Factor	ADPS[2:0]- Factor
000 – 1	100 – 16
001 – 2	101 – 32
010 – 4	110 – 64
011 – 8	111 – 128

4) DIDR0 (Digital Input Disable Register 0)

When an analog signal is applied to the ADC5...0 pin and the digital input from this pin is not needed, this bit should be written logic one to reduce power consumption in the digital input buffer. ***DIDR0 = 0x3F***

Bit	7	6	5	4	3	2	1	0
			ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D

5) ADCSRB

This is not needed for now. It has 2 important fields: (ACME-Analog Comparator Multiplexer Enable and ADTS[2:0]- Auto Trigger Source Selection).

Steps for ATmega328P ADC Programming and C-Code

To program the A/D converter of the AVR, the following steps must be taken:	Relevant Register Values or C-Code
1. Make the pin for the selected ADC channel an <u>input pin.</u>	`DDRC=0b00000000//assume PortC`
2. <u>Turn on the ADC module</u> of the AVR because it is disabled upon power-on reset to save power. 3. Select the <u>conversion speed</u>. We use registers ADPS2:0 to select the conversion speed.	`ADCSRA =0b10000111`
4. Select <u>voltage reference</u> and ADC <u>input channels</u>.	`ADMUX = 0b01100000` Vref is AVCC (REFS = 01) Left Adjusted (ADLAR=1) ADC0 Input Channel (MUX=0000)

5. Activate the <u>start conversion</u> bit by writing a one to the ADSC bit of ADCSRA.	`ADCSRA | = (1<<6)`
	`Or, 1<< ADSC`

6. Wait for the <u>conversion to be completed</u> by checking the ADIF bit in the ADCSRA register.	`(ADSCRA & (0b00010000))` Equal to 0 as long as conversion takes place due to ADIF bit (0)

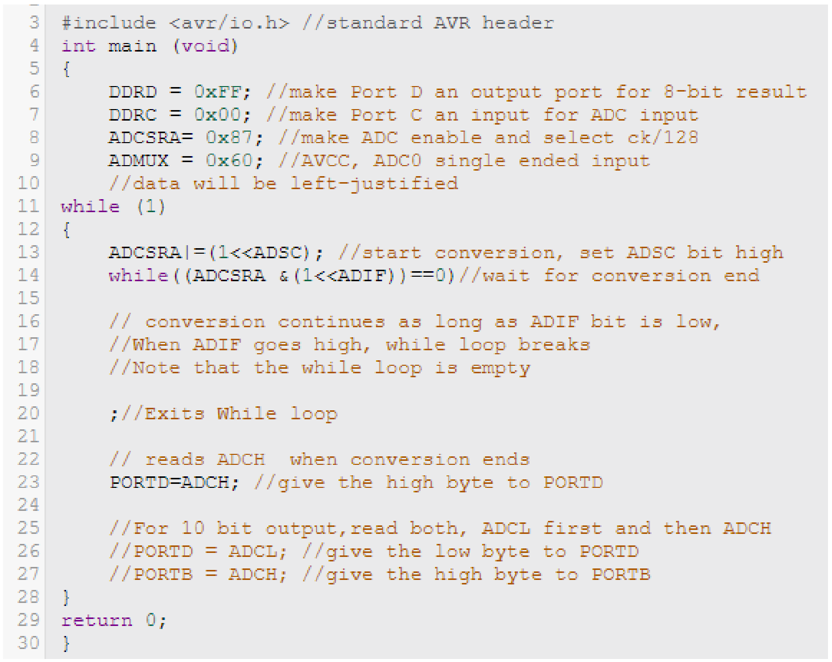
7. After the ADIF bit has gone HIGH, <u>read</u> the ADCL and ADCH registers to get the digital data output.
 	
Notice that you have to read ADCL before ADCH; otherwise, the result will not be valid.
	`(ADSCRA & (0b00010000))`
	NOT equal to 0 when conversion is complete due to ADIF bit (1)
	Read:
	For 8-bit result only, x=ADCH;
	Where, x is unsigned char
	Or to forward at output, PORTB=ADCH
	For 10-bit result, x=ADC
	Where, x is unsigned int
	`Or, PORTB=ADCL; PORTD=ADCH`

8. If you want to read the selected channel again, go back to step 5. If you want to select another Vref source or input channel, go back to step 4	

NOTE: The ADC has **two different operating modes**. In ***single conversion mode***, each conversion will be initiated by the user. In **free running mode**, the ADC is constantly sampling and updating the ADC Data Registers. 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 to start a new conversion. In free running mode, a new conversion will be started immediately after the conversion completes while ADSC remains high. For further description, refer to the ATmega328P datasheet section Analog to Digital Converter.

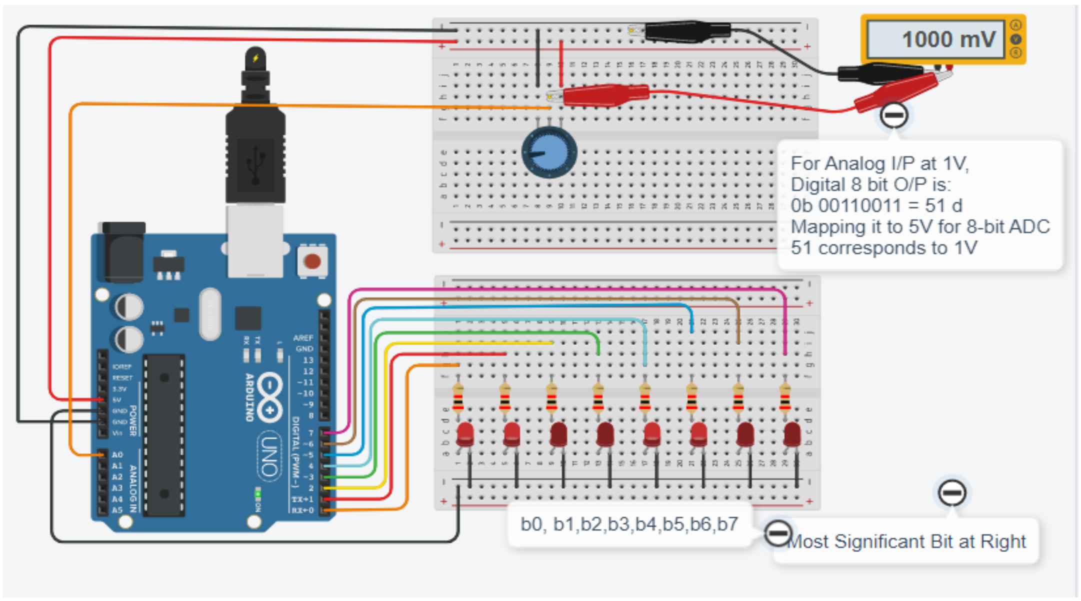
Example # 1: Reading an Analog Voltage Set by a Potentiometer

In this example, we are taking analog input through a potentiometer. The analog input voltage can vary from 0 to 5V. This is given at ADC0 input channel. The digital 8-bit output is taken through PORTD pins by reading the ADCH register. An LED is used to indicate each bit of the digital output. For a reference voltage of 5V, using 8-bit ADC, each level corresponds to 19.53 mV.



```
#include <avr/io.h> //standard AVR header
int main (void)
{
    DDRD = 0xFF; //make Port D an output port for 8-bit result
    DDRC = 0x00; //make Port C an input for ADC input
    ADCSRA= 0x87; //make ADC enable and select ck/128
    ADMUX = 0x60; //AVCC, ADC0 single ended input
    //data will be left-justified
while (1)
{
    ADCSRA|=(1<<ADSC); //start conversion, set ADSC bit high
    while((ADCSRA &(1<<ADIF))==0)//wait for conversion end

    // conversion continues as long as ADIF bit is low,
    //When ADIF goes high, while loop breaks
    //Note that the while loop is empty

    ;//Exits While loop

    // reads ADCH  when conversion ends
    PORTD=ADCH; //give the high byte to PORTD

    //For 10 bit output,read both, ADCL first and then ADCH
    //PORTD = ADCL; //give the low byte to PORTD
    //PORTB = ADCH; //give the high byte to PORTB
}
return 0;
}
```

Figure 6: Code for Example 1- Reading Analog Input from ADC0 Channel



For Analog I/P at 1V,
Digital 8 bit O/P is:
0b 00110011 = 51 d
Mapping it to 5V for 8-bit ADC
51 corresponds to 1V

Figure 7: Connections and Output Representation for Example 1

LAB TASKS

TASK 1: To test the Example 1 on ATmega328P and verify working of ADC using an external D/A Converter

1. Create a new AVR project and build the code given in the Example 1. Test the obtained digital output by varying the output from potentiometer. Observe the on / off status of LEDs (used as indicators for digital output) and verify the digital voltage by calculation for 8-bit ADC at given reference voltage.

Note that the ATmega328P has an ADC module but not a built-in DAC module so it cannot provide an analog output through any of its pins.

2. Now, remove the LEDs and provide the 8-bit digital output to an external DAC circuit i.e., convert the digital output to analog for verification. You can use a simple **R-2R** circuit of Figure 8 or use **DAC0808 IC**. For DAC0808 IC, refer to its datasheet for more information. The pin diagram and DAC circuit using this is shown in Figure 9 and 10.

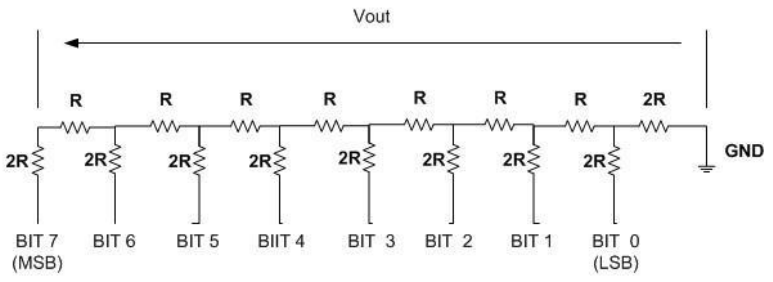


Figure 8: Layout of 8-bit R-2R Ladder Circuit-DAC

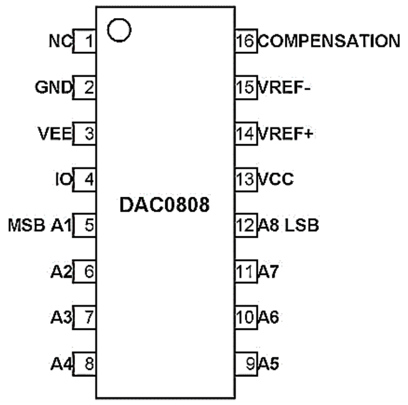


Figure 9: DAC0808 (8-bit R-2R based D/A Converter)

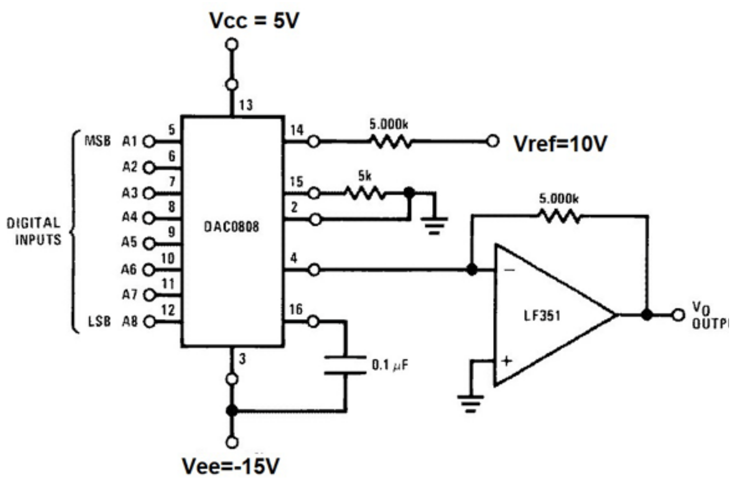
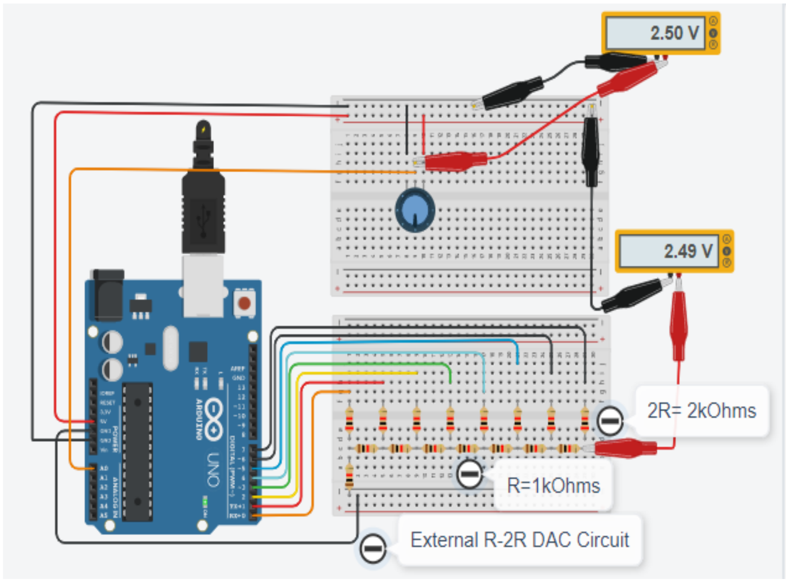
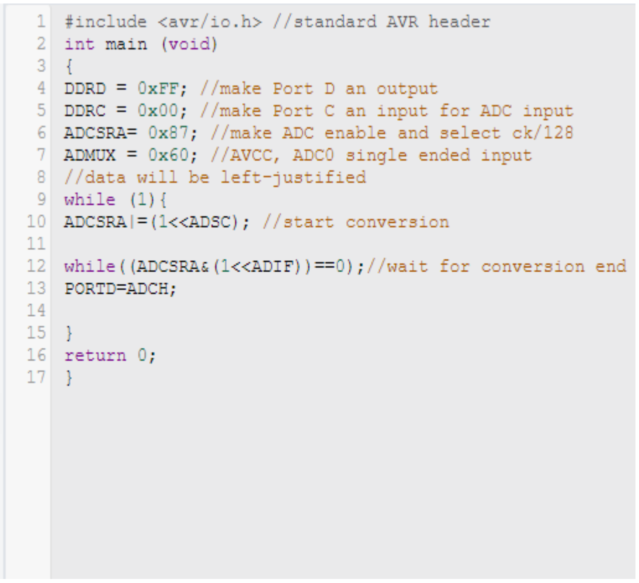


Figure 10: Sample Circuit for Digital to Analog Converter Circuit using DAC0808

3. Measure the analog input given through potentiometer and the analog output reproduced by the DAC. Compare both and verify the ADC and DAC. This is shown in Figure 11 using R-2R circuit.





```
#include <avr/io.h> //standard AVR header
int main (void)
{
DDRD = 0xFF; //make Port D an output
DDRC = 0x00; //make Port C an input for ADC input
ADCSRA= 0x87; //make ADC enable and select ck/128
ADMUX = 0x60; //AVCC, ADC0 single ended input
//data will be left-justified
while (1){
ADCSRA|=(1<<ADSC); //start conversion

while((ADCSRA&(1<<ADIF))==0);//wait for conversion end
PORTD=ADCH;

}
return 0;
}
```

Figure 11: Sample Representation of Task 1 with R-2R DAC

TASK 2: To control the status of an LED based on the value of input analog voltage

Modify the previous task or, example to read 10-bit ADC value of voltage across potentiometer instead of 8-bit result. Map the obtained 10-bit result with voltage level using the step-size. Connect an LED to indicate the voltage level. Use some conditions to build the following logic for controlling the LED status.

- If voltage is above 2.5V, the LED turns ON
- If voltage is below 2.5V, the LED red turns OFF

⚠Note: Pay attention to the **data type** of variables when you calculate the input analog voltage using the step-size and ADC result.

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**

Laboratory Session No.: ________________________ Date: ______________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: <u>*Imitate*</u> and <u>*practice*</u> given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to <u>*operate*</u> software environment <u>*under supervision*</u>, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: <u>*Detect*</u> Errors/Exceptions and <u>*manipulate,*</u> under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: <u>*Copy*</u> or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 04

OBJECTIVE:

To utilize the USART (Universal Synchronous / Asynchronous Receiver /Transmitter) of ATmega328P for transmitting and receiving data though asynchronous serial communication with PC

LAB OUTCOMES:

By the end of this lab, you should be able to:

1) Recognize the basics of serial communication protocol; baud-rate, stop bit, data bits, parity etc. and the importance of required connectors (RS-232)
2) Identify the AVR ATmega328P pins associated with the USART
3) Identify the purpose of different fields of USART registers
4) Program ATmega328P for initializing the USART with given baud-rate
5) Program ATmega328P in C-language to establish serial communication with PC
6) Test and verify data (character, string, integer and float) transmission and reception for given conditions using a Serial Terminal Emulator like TeraTerm

"The single biggest problem in communication is the illusion that it has taken place."

– George Bernard Shaw

BACKGROUND:

Microcontrollers are provided with ability to communicate with external devices like computer, other micro-controllers and peripherals. This communication is done through different protocols to allow microcontrollers to send and receive data. ATmega328P is provided with USART (Universal Synchronous/Asynchronous Transmitter/Receiver). In this lab, we will be exploring **Asynchronous Transmitter and Receiver (UART).** It is not only used as a communications link to external device but also as a debugging port to send status messages. This is one of the 3 communication options that can be established with ATmga328P. The other two are SPI and I2C which will be explored later. Before discussing the working of relevant pins and registers, we first need to understand the different types and basics of communication protocols.

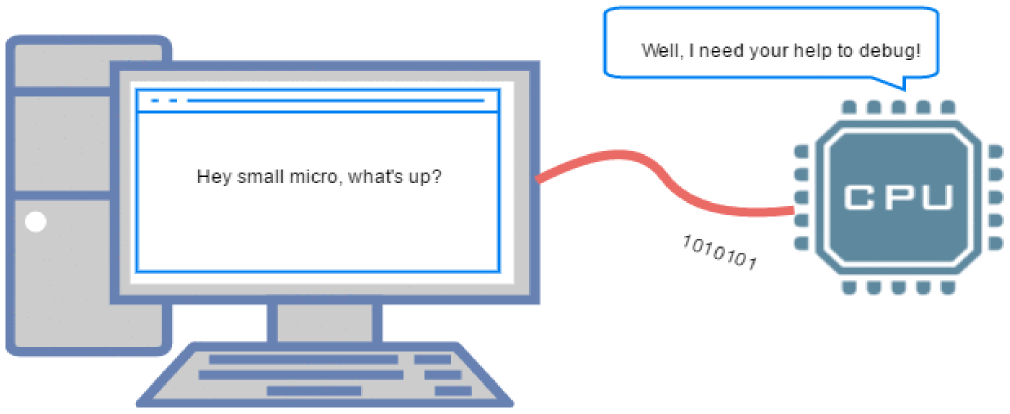


Figure 1: Serial Communication of Microcontroller with PC for Troubleshooting

Basics of Serial Communication

Parallel Vs Serial Data Transfer: Computers transfer data in two ways: parallel and serial.

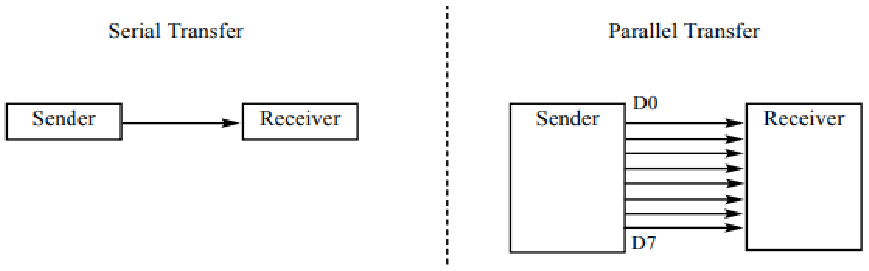


Figure 2: Serial and Parallel Data Transfer Representation

In parallel data transfer, the data is sent one byte (or multiple bits) at a time. For this multiple wires are needed and this is suitable for a short-distance like printers. In serial communication, the data is sent one bit at a time. It needs lesser number of wires. It is suitable for longer distance communication and is cheaper.

Synchronous vs Asynchronous Communication: Serial data communication can be, asynchronous or synchronous. The synchronous method transfers a block of data (characters) at a time, whereas the asynchronous method transfers a single byte at a time. In Synchronous transmission a common clock is shared by the transmitter and receiver to achieve synchronization while data transmission. In asynchronous interface, it does not have any separate clock signal. Only the data is sent on the lines and the transmitter must send the data at ***an agreed upon rate*** and in a defined manner.

Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a configurable peripheral of ATmega328p which supports both Synchronous (SPI) and Asynchronous (Serial) communication protocols. In our case we will be dealing only with the asynchronous communication.

Asynchronous Communication Protocol

Baud Rate – Data Transfer Rate: The rate of data transfer in serial data communication is stated in bps (bits per second). Another widely used terminology for bps is baud rate. In the context of microcontroller USART programming, we will be using the terms bps and baud interchangeably. The maximum data transfer rate is limited by the hardware ports but the transmitter and receiver must agree on the same baud rate.

Data Framing: In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a **start bit** and a **stop bit**. The start bit is always one bit, but the stop bit can be one or two bits. In modern PCs, however, the use of one stop bit is standard. The start bit is always a 0 (low), and the stop bit(s) is 1 (high). For example, look at Figure in which the ASCII character "A" (**8-bit binary** 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first. There are a total of 10 bits for the character: 8 data bits for the ASCII code, and 1 bit each for the start and stop bits.

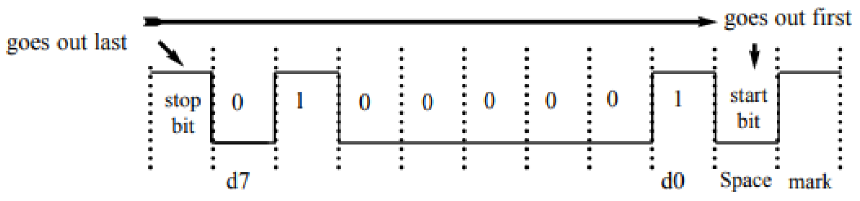


Figure 3: Framing ASCII 'A' (41H) with a stop-bit (1) and a start bit (0)

Parity Bit: UART chips allow programming of the parity bit for odd-, even-, and no-parity options. It is a single bit added to the data frame to maintain data integrity.

ATmega328P Serial Port Pins and Connection with PC

ATmega328P has one serial port. **Pin 2 and pin 3 of ATmega328P** serve as USART0 transmitter (TxD) and receiver (RxD) pins. **Arduino UNO pin 0 and 1** are therefore marked as TX and RX. These are the same pins on the chip as I/O ports PD1 and PD0. This means that applications that use the USART0 cannot also use these two bits in Port D. It is not necessary to set any bits in the DDRD register in order to use the USART0.

If LEDs placed on the RX and TX pins will flash, they indicate the transmission of data.

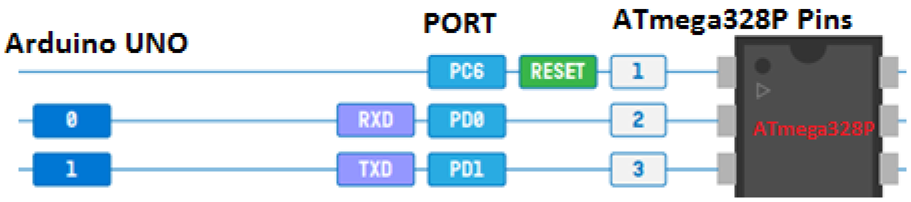


Figure 4: Connection of ATmega328P RXD and TXD pins with Arduino UNO for UART

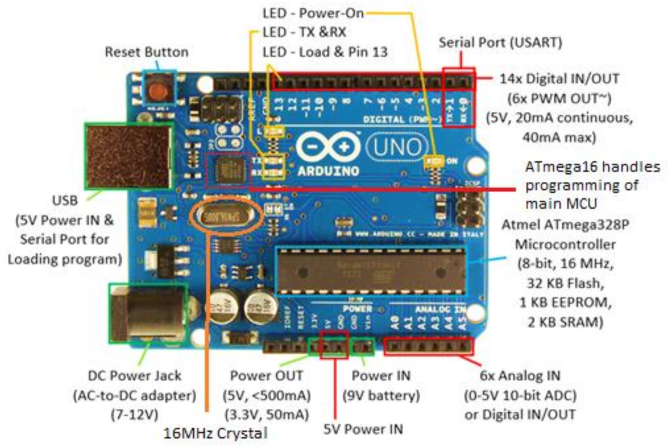

LED - Power-On
LED - TX &RX
LED - Load & Pin 13
Reset Button
Serial Port (USART)
14x Digital IN/OUT (6x PWM OUT~) (5V, 20mA continuous, 40mA max)
ATmega16 handles programming of main MCU
USB (5V Power IN & Serial Port for Loading program)
Atmel ATmega328P Microcontroller (8-bit, 16 MHz, 32 KB Flash, 1 KB EEPROM, 2 KB SRAM)
DC Power Jack (AC-to-DC adapter) (7-12V)
16MHz Crystal
Power OUT (5V, <500mA) (3.3V, 50mA)
Power IN (9V battery)
5V Power IN
6x Analog IN (0-5V 10-bit ADC) or Digital IN/OUT

Figure 5: Placement of RX and TX pins and corresponding LEDs with ATmega328P and ATmega16 on Arduino UNO board

To establish communication between microcontroller (USART pins) with PC, the PC must have a communication port to support serial data transfer. These are called COM ports. A COM port is simply an I/O interface that enables the connection of a serial device to a computer. COM ports are also referred to as serial ports. They are asynchronous interfaces that can transmit one bit of data at a time when connected to a serial device.

Figure 6: RS-232 Serial Port (DB9)

The AVR serial port can be connected to the COM port of a PC for serial communication. However, USB interfaces have largely replaced the RS232 serial ports seen in the past as a faster way of performing serial data transmission. In the absence of a COM port, a COM-to-USB converter module is needed. You can read more about it here.

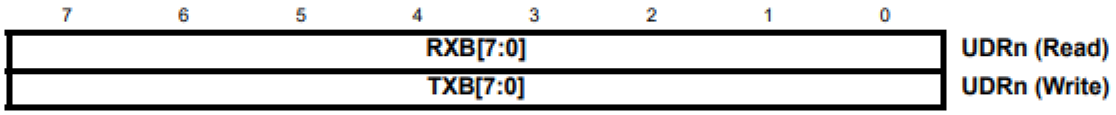
Luckily, the **Atmega16U2** incorporated on the UNO (R3) board acts as a USB-to-serial converter for serial communication using USB com drivers. On PC, a software applications is used with it to send data to or display the received data from the board.

ATmega328P Serial Port Registers

1) UDR0 – USART Data I/O Register

This register is used to hold data to be sent or received.

- The USART Transmit Data Buffer Register and USART Receive Data Buffer Registers share the same I/O address referred to as USART Data Register or UDR0.
- For data transmission, TXB will be the destination for data written to the UDR0 Register location.
- For data reception, reading the UDR0 Register location will return the contents of RXB.



2) UCSR0A – USART0 Control and Status Register A

UCSR0 A	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	RXC0	TXC0	UDRE0	FE0	DOR0	PE0	U2X0	MPCM0
Bits	**Function**							
RXC0	*USART Receive Complete* Indicates receive buffer register (RXB) status. 1: Unread data present in receive buffer 0: Receive buffer is empty							
TXC0	*USART Transmit Complete* 1: Entire frame in transmit shift register has been transmitted, no new data available in transmit data buffer register (TXB)							
UDRE0	*USART Data Register Empty* 1: Transmit data buffer register is ready to receive new data 0: TXB is not empty. Don't write to UDR if UDRE0 is 0							
FE0	*Frame Error* 1: Frame error occurred in receiving next character in receive buffer Frame error is detected if first stop bit of character in RXB is 0							
DOR0	*Data Over-Run* 1: Indicates data over-run Data over-run occurs if RXB and receive shift-register are full and new start bit is detected.							
PE0	*USART Parity Error* 1: Indicates parity error in the receive buffer if Parity Checking UPM01 is enabled.							
U2X0	*Double the USART Transmission Speed* 1: It doubles the transfer rate for asynchronous operation (baud rate divisor becomes 8 instead of 16)							
MPCM0	*Multi-processor Communication Mode* Enables the Multi-processor Communication mode. Transmitter is unaffected by it.							

The default value of **UCSR0A to 0x20 = 0b 0010 0000**

3) UCSR0B – USART0 Control and Status Register B

UCSR0B	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

Bits	**Function**
RXCIE0	*RX Complete Interrupt Enable* 1: Set 1 to enable the interrupt on the RXC flag in UCSR0A
TXCIE0	*TX Complete Interrupt Enable* 1: Set to enable the interrupt on the TXC flag in UCSR0A
UDRIE0	*USART Data Register Empty Interrupt Enable* 1: Set to one enables interrupt on the UDRE0
RXEN0	*Receiver Enable* 1: Enables the USART Receiver 0: Disables the Receiver. Flushes the RXB.
TXEN0	*Transmitter Enable* 1: Enables the USART Transmitter 0: Disables the Transmitter; effective once transmission is complete.
UCSZ02	*Character Size* The UCSZn2 bits combined with the UCSZn1:0 bit in UCSR0C sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.
RXB80	*Receive Data Bit 8* It is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from UDR0.
TXB80	*Transmit Data Bit 8* It is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to UDR0.

RXEN0, TXEN0 and UCSZ02 are most important here for enabling the receiver, and transmitted and to set the character size. The interrupt related bits are not needed now. Its default value is 0x00.

4) UCSR0C – USART0 Control and Status Register C

UCSR0C	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0

Bits	**Function**
UMSEL01 **UMSEL00**	*USART Mode Select Bits (UMSEL01-00)* 00: Asynchronous USART 01: Synchronous USART
UPM01 **UPM00**	*USART Parity Mode (UPM01-00)* These enable and set type of parity generation and check. If enabled, the Transmitter will automatically generate and send the parity of the transmitted data bits within each frame. The Receiver will generate a parity value for the incoming data and compare it to the UPM0 setting. If a mismatch is detected, the UPE0 Flag in UCSR0A will be set. 00: Disabled 10: Enabled, Even Parity 11: Enabled, Odd Parity
USBS0	*USART Stop Select Bit* Selects number of stop bits to be inserted by the Transmitter. 0: 1-bit Stop bit 1: 2-bit Stop bits

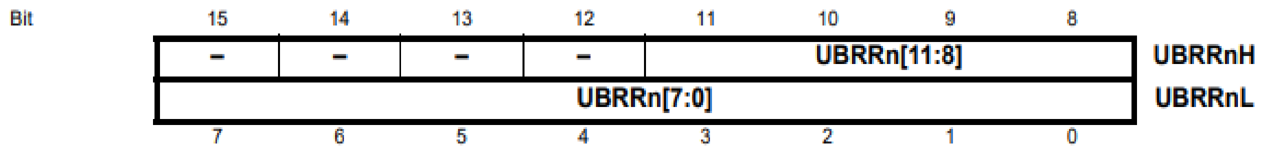
UCSZ01 **UCSZ00**	*Character Size (UCSZ01:00)* The UCSZ01:00 bits combined with the UCSZ02 bit in UCSR0B sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.
UCPOL0	*Clock Polarity* 0: The bit is cleared when asynchronous mode is used. In synchronous mode, it is useful for setting relation between data and clock.

[UCSZ02,USCZ01,UCSZ00]	Character Size
000	5-bit
001	6-bit
010	7-bit
011	8-bit
111	9-bit

Correct initialization of all the UCSR0C bits is important in setting communication protocols. Its default value is 0x06 = 0b000 0110.

5) UBRR0 – USART0 Baud Rate Register

It is used to set baud-rate by specifying the pre-scalar in its 12 bits.



UBRR[15:12] The 4 bits, reserved, are set to 0. The remaining 12 bits UBRR[11:0] contain the USART0 baud rate (pre-scalar). The UBRR0H contains the four most significant bits, and the UBRR0L contains the eight least significant bits of the USART0 baud rate.

For required baud rate 'BAUD', and oscillator frequency f_{osc}, the value for UBRR0 is calculated by;

$$UBRR0 = \frac{fosc}{16 \times BAUD} - 1$$

For a 16MHz clock, the required values of UBRR0 register are given in the Table below for different baud-rates. Note: U2X0 is set 0 here. The baud-rates can be doubled by setting U2X0 high for same UBRR0 values. The formula can be applied for verification.

Table 1: UBRR0 Values for Different Baud Rates

Baud Rate (bps)	UBRR0
2400	416 = 0x01A0
4800	207 = 0x00CF
9600	103= 0x0067
14400	68 = 0x0044
19200	51 =0x0033

Serial Terminal Emulator – TeraTerm

COM Port (communication port) is the original, yet still common, name of the serial port interface on PC-compatible computers. It can refer not only to physical ports, but also to **emulated ports**.

Serial terminal emulators are software applications that replicate physical COM ports. The virtual serial ports are fully compatible with operating systems and applications and are treated in the same way as a real port. These are used for the serial communication between the host computer and an embedded system

(Target). It is mainly used as a user interface for debugging embedded system. It is also used for sending commands, displaying result, loading firmware, logging result, etc.

Tera Term and PuTTY are famous terminal emulator applications. In this lab, we can use Tera Term. It is an open-source, free, software implemented terminal emulator (communications) program.

1) Download Tera-Term using: https://filehippo.com/download_tera-term/
2) Type Tera Term in Windows search to open it. You will be able to select Serial once you connect your device to PC USB port (connect Arduino UNO). The Serial and Port options will be enabled.

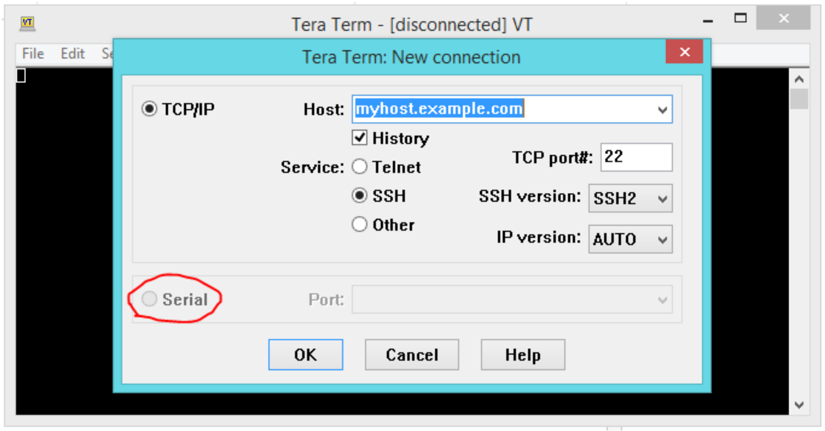
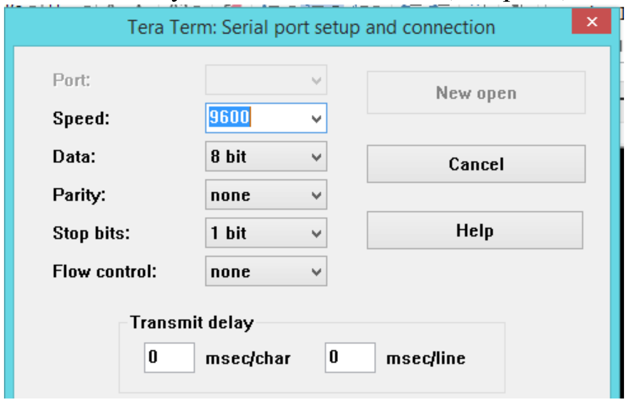


Figure 7: Tera Term Connection

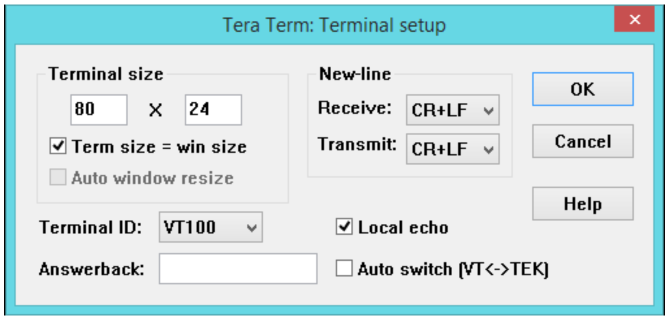
3) Go to Setup >> Serial port… It allows you to select Baud Rate, Stop Bit, Data Bits, and Parity etc.


Tera Term: Serial port setup and connection

Port:
Speed: 9600
Data: 8 bit
Parity: none
Stop bits: 1 bit
Flow control: none
Transmit delay: 0 msec/char 0 msec/line

Figure 8: Tera Term Serial Port Settings

4) For writing to Serial Terminal window (send character from PC to Microcontroller), turn on echo. Go to **Setup** >> **Terminal** and select **Local echo**.


Tera Term: Terminal setup

Terminal size: 80 X 24
☑ Term size = win size
☐ Auto window resize
New-line Receive: CR+LF
New-line Transmit: CR+LF
Terminal ID: VT100
Answerback:
☑ Local echo
☐ Auto switch (VT<->TEK)

Figure 9: Tera Term Terminal Setup

5) Save the settings for later use. Select **Setup** > **Save** setup.

ATmega328P Serial Port Programming in C

Programming for USART Initialization

The USART has to be initialized before any communication can take place.

The initialization process consists of the following steps.	Relevant Register Values (example) & C-Code
1. **Setting-up as Transmitter and / or Receiver** Enable the TxD and (or) RxD pins using TXEN0 and (or) RXEN0 bits of UCSR0B.	`UCSR0B=0b00011000 or,` `UCSR0B =(1<<TXEN)| (1<<RXEN)`
2. **Setting-up Frame Rate** Load UCSR0C to indicate asynchronous mode with 8-bit data frame, no parity, and one stop bit	`UCSR0C=0x06 or,` `UCSR0C=0b00000110`
3. **Set the baud-rate using UBRR.**	`UBRR0 = 0x67` For baud-rate of 9600 bps at 16MHz crystal frequency at U2X0 =0.

➢ **Subroutine for USART initialization**

For ease, we create a subroutine for USART initialization to avoid repeating these lines of code again and again.

```
void usart_init (void)
{
    UBRR0=0x67; //set pre-scalar to configure baud rate (9600)
    UCSR0A &= ~ (1 << U2X0); //Single Speed U2X - 0 (can be set to 1)
    UCSR0B = (1 << RXEN0) | (1 << TXEN0); // enable transmitter and receiver
       // Async mode, parity mode = disabled, 1 stop bit, character size = 8 bit
       clock polarity = 0 for async communication
    UCSR0C = (0<<USBS0) | (1 << UCSZ01) | (1 << UCSZ00);
}
```

Programming for Data Transmission

To program the USART as Transmitter, follow the steps:	Relevant Register Values or C-Code
1. Initialize the USART.	Call `usart_init( );`
2. Monitor the UDRE bit of UCSR0A to make sure UDR is ready to accept byte to transmit.	`while (!(UCSR0A & (1<<UDRE0))) { };` An empty while loop that waits to check UDR is empty (indicated by UDRE bit).
3. Write the character to be transmitted to the UDR.	`UDR0=ch;` Where, ch is an unsigned char for example 'A'.
4. Wait for complete frame transmission.	`while(!(UCSR0A &(1 << TXC0))){};`
5. To transmit the next character, go to Step 2.	

➢ **Subroutine for transmitting character**

```
void usart_putChar(unsigned char data)
{
    while (!(UCSR0A & (1 << UDRE0))) {};   //wait for data register to be empty
    UDR0 = data;                          //write data
    while (!(UCSR0A & (1 << TXC0)) ) {};   //wait for complete frame transmission
}
```

Programming for Data Reception

To program the USART as Receiver, follow the steps:	Relevant Register Values or C-Code
1. Initialize the USART.	Call `usart_init( );`
2. Monitor the RXC flag bit of the UCSR0A register to see if an entire character has been received yet.	`While (! (UCSR0A & (1<<RXC0))) ;` An empty while loop that waits for data to be received. Loop ends when RXC bit turns high as the given condition is not equal to 0.
3. When RXC is high, read the UDR as it has received the byte.	`ch=UDR;` Where, **ch** is an unsigned char type variable to store the received character.
4. To receive the next character, go to Step 4.	

➢ Function for receiving character

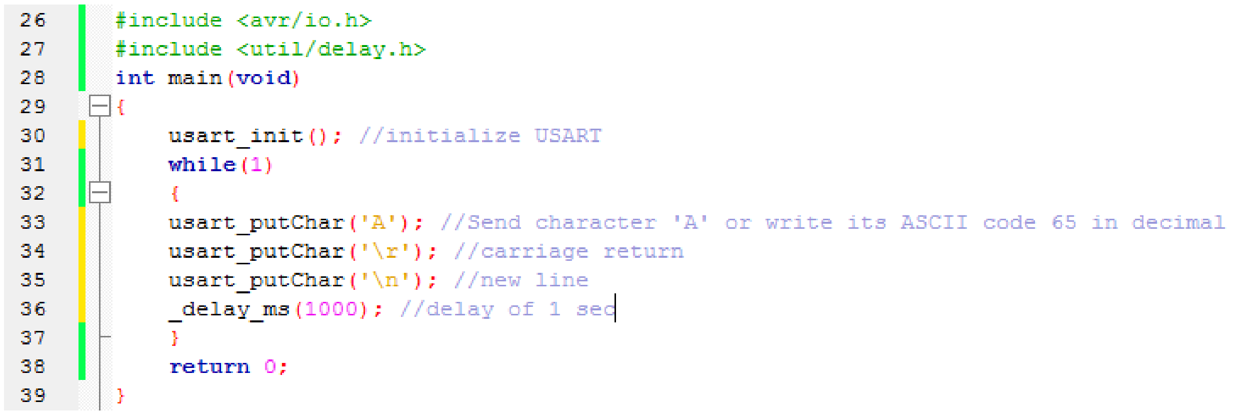
```
char usart_getChar()
{
    char data;
    while ( !(UCSR0A & (1 << RXC0))) {}; // wait for data to receive
    data = UDR0;
    return data;
}
```

EXAMPLES

The tested asynchronous communication is polling based and not interrupt based.

Example # 1: Transmitting a Character from ATmega328P to PC through UART

The following code transmits ‘A’ repeatedly with a delay of 1 sec. Note that only one character is sent at a time. The following example uses usart_init( ) and usart_putChar( ) from the listed 3 sub-routines therefore, these sub-routines must be defined in the main.c file.



```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
    usart_init(); //initialize USART
    while(1)
    {
    usart_putChar('A'); //Send character 'A' or write its ASCII code 65 in decimal
    usart_putChar('\r'); //carriage return
    usart_putChar('\n'); //new line
    _delay_ms(1000); //delay of 1 sec
    }
    return 0;
}
```

Figure 10: Code for Example#1

See the character ‘A’ is written in quotes. You can write its ASCII code (65). In that case, don’t use ‘ ’ as we want to transmit a single character ‘A’ through the ASCII code and not the integer 65 (which are 2 characters).

Now, you are able to do Task 1 and Task 2.

Example # 2: Transmitting and Receiving Strings, Integers and Floats

For transmitting a string of characters (instead of a single character), we can write a C subroutine that transmits one character at a time using the previously described usart_putChar( ). Similarly, to receive a string of characters, we can makes use of a buffer to hold the received characters in the form of a string. Look at the following subroutines.

- **Subroutines for receiving and transmitting strings**

```
//subroutine for transmitting a character string
void usart_putString(char* StringPtr)
{
    while(*StringPtr != 0x00)
    {
        usart_putChar((unsigned char)*StringPtr);
        StringPtr++;
    }
}
```

Type Casting

Converting one datatype into another is known as type casting or, type-conversion. To transmit **integer** or **float** data through USART, we first need to convert them into **string**. There are different approaches to do so, one simple technique is shown here.

- **Integer to String:**

The **`itoa(num,buffer,10)`** function coverts the integer **num** into a null-terminated character string. The string is placed in the **buffer** passed, which must be large enough to hold the output. The last input shows number format 10 for decimal. Include <stdlib.h> for itoa( ).

- **Float to String:**

The **`dtostrf (val,width,prec,s)`** function converts the double value passed in **val** into an ASCII representation that will be stored under **s**. Conversion is done in the format '[-]d.ddd'. The minimum field width of the output string (including the possible '.' and the possible sign for negative values) is given in **width**, and **prec** determines the number of digits after the decimal sign. The dtostrf() function returns the pointer to the converted string s.

The example code given here shows transmission of strings, float and integers by utilizing the subroutines and functions discussed above. You can observe that a string is received from PC to AVR too.

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
int main(void)
{
    usart_init(); //initialize USART
    //Transmitting and Receiving Character String
    volatile char CNIC[13]; //array to take input from received characters
    uint8_t a=13; //13 char input is to be received (cnic 13 digits)
    usart_putString("Enter your 13-digit CNIC \n\r"); //sending string (display message)
    usart_getString(CNIC,a); //receiving string
    usart_putString("\n\rThe entered CNIC is: \n\r");
    usart_putString(CNIC);  //verifying the received characters by transmitting again*/

    //Transmitting an Integer for Display
    usart_putString("\n \r Displaying Integers:\n\r");
    int num=5555;  //integer value to be transmitted
    char buffer [sizeof(int)*8+1];  //char array for holding string after type conversion
    itoa(num,buffer,10); //conversion from integer to string
    usart_putString(buffer);

    //Transmitting a float value
    usart_putString("\n\r Displaying Float or Double:\n\r");
    float temp=23.51;
    char str_temp[10];
    dtostrf(temp,5,5,str_temp); //double to string conversion,
    usart_putString(str_temp);
return 0;
}
```

Figure 11: Code for Example 2-Transmitting Strings, Float and Integer Data Types

Header File usart.h

For ease, you can include the header and source files (usart.h and usart.c) provided with the manual. Utilize its simple functions for transmitting and receiving data or use the sub-routines discussed above in your code to complete the given lab tasks.

LAB TASKS

TASK 1: To test Example 1 for transmitting a character serially at baud-rate of 9600 with 1 stop bit using ATmega328P USART

Test the Example 1 code using Arduino UNO. After building the given code, program the ATmega328P. Now, disconnect and reconnect Arduino UNO with PC port. Open Serial Terminal Emulator (Tera Term). Make connection with the required settings and observe the serial terminal. You should see the data transmitted by AVR (received by the PC) on serial terminal.

- Is there any impact if you select a different baud rate in Tera Term without changing the baud rate initialized in the ATmega328P code? Are you able to correctly transmit the characters when microcontroller and PC work at different baud rate?

  ____

- Add two more lines to the code and comment on the result.
  usart_putChar(65)
  usart_putChar('65')

TASK 2: To program ATmega328P for controlling the status of LED based on the received character

Modify the previous code to make the microcontroller receive a character sent by PC. If the received character is 'A', turn on the on board LED otherwise turn it off. The data should be received at baud-rate of 14400 with 1 stop bit.

TASK 3: To transmit the analog voltage across a potentiometer read by the ATmega328P ADC to PC

Extend the Task 2 of Lab 03 where you used ADC module to measure the voltage across potentiometer. Send (transmit) the following through USART of the microcontroller to the PC serial terminal.

1) The 10-bit ADC output (in range of 0 to 1023)
2) The analog voltage across potentiometer in Volts

⚠ Pay attention to the data-type. You may use any suitable baud rate of your choice.

Vary the voltage and observe the values. Verify the analog voltage reading by ADC module and transmission through the USART by comparing voltmeter reading and values displayed by Tera Term.

Sample Output:

```
**********10-bit ADC************
**********5V Reference Voltage****
*****Voltage across Potentiometer***


ADC Value: 307
Output Voltage: 1.5 Volts


ADC Value: 205
Output Voltage: 1.01 Volts
```

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**
Laboratory Session No.: ________________________ Date: ______________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: *Imitate* and *practice* given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to *operate* software environment *under supervision*, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: *Detect* Errors/Exceptions and *manipulate,* under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: *Copy* or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 05

OBJECTIVE:

To interface an LCD (Liquid Crystal Display) screen with ATmega328P by sending required commands and data

LAB OUTCOMES:

By the end of this lab, you would be able to:

1) Identify the pins and commands for controlling a 16x2 LCD
2) Interface a 16x2 LCD screen with ATmega328P and display different messages

"I don't care what it is, when it has an LCD screen, it makes it better." — *Kevin Rose*

INTRODUCTION:

Display units like LEDs, 7-segment LED displays, LCD screens etc. play an important part in establishing a good communication between the users and machines, and therefore, are vital for embedded systems. Through display screens, the user gets a feeling of knowing the system's working status. Consider the examples of ATM machine, automatic washing machine or microwave ovens. They allow us to give input through keypad or knobs or touch screens, and display useful messages on screens which guide us or show the status of process. LCD screens are now seen everywhere due to their declining prices, ease of programming due to an internal controller, and ability to display characters and graphics.

For learning purpose, we are interfacing our ATmega32P microcontroller with a standard 16x2 LCD screen. 16×2 LCD is named so because; it has 16 Columns and 2 Rows. Such dot-matrix LCDs are available in different packages like 8x1, 8x2, 16x1, and 20x4.

16x2 LCD (Liquid Crystal Display)

Features

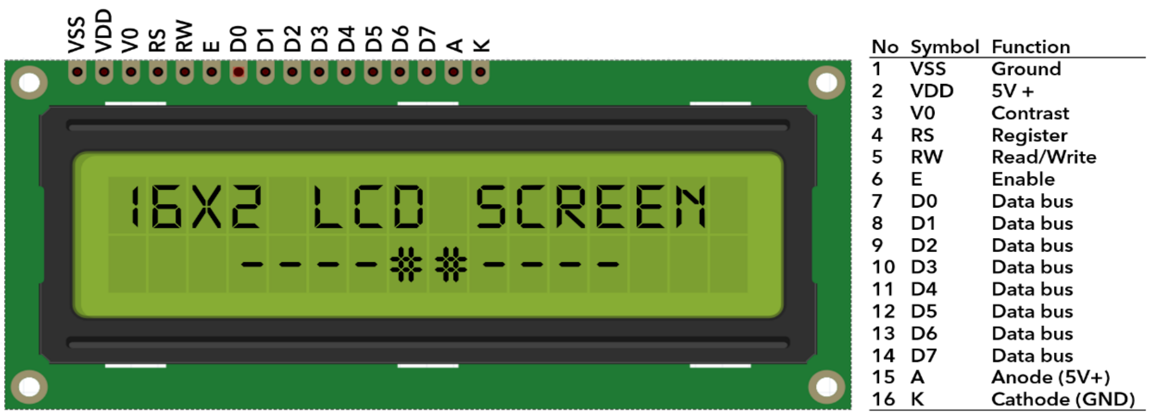
- The operating voltage of this LCD is 4.7V-5.3V
- It includes two rows where each row can produce 16-characters.
- The utilization of current is 1mA with no backlight
- Every character can be built with a ***5×8 pixel box***
- It can display alphabets, numbers and a few custom generated characters
- It can work on two modes (4-bit and 8-bit): In 4-bit mode we send the 4 bits (out of the total 8-bits) at a time and in the 8-bit mode, we can send all 8-bits in one stroke.
- These are obtainable in Blue & Green Backlight

The LCD can display 32 characters in total and each character will be made of 5*8 (40) pixel dots. The standard LCDs have HD44780 dot-matrix liquid crystal display controller / driver that is mounted on LCD module itself. The function of this interface IC is to get the commands and data (sent over **parallel** data lines) from the MCU and process them to display meaningful information onto our LCD Screen. Hence, the MCU doesn't directly have to deal with the 1280 pixels of LCD and their positions.

[Datasheet HD44780.]

LCD Pinout

Figure 1 shows the 16 pins of a 16x2 LCD and their names. Most of the LCDs have these 16 pins that are used for connection according to their functionality. Let's discuss the function of each pin one-by-one.



No	Symbol	Function
1	VSS	Ground
2	VDD	5V +
3	V0	Contrast
4	RS	Register
5	RW	Read/Write
6	E	Enable
7	D0	Data bus
8	D1	Data bus
9	D2	Data bus
10	D3	Data bus
11	D4	Data bus
12	D5	Data bus
13	D6	Data bus
14	D7	Data bus
15	A	Anode (5V+)
16	K	Cathode (GND)

Figure 1: 16x2 LCD Pinout

#	Pin Name	Description / Function	Type	Connection
1	**VSS**	Pin for LCD ground.	Source Pin	Connected to the ground of the MCU/ Power source.
2	**VDD**	Pin for LCD supply voltage.		Connected to the supply pin of power source / +5V of MCU.
3	**V0**	**Contrast Control:** Adjusts the contrast of the LCD.	Control Pins	Connected to a variable potentiometer that can source 0-5V.
4	**RS**	**Register Select:** Selects either command register or data register		Connected to a digital output pin of MCU. **0:** Command Mode **1:** Data Mode
5	**RW**	**Read/Write:** Toggles the LCD between Read/Write operations. Read operation is rarely needed for information like cursor position etc.		Connected to a digital output pin of MCU. **0:** Write Operation **1:** Read Operation
6	**E**	**Enable:** Must be held high to perform Read/Write Operation		The enable pin is used by the LCD to latch information presented to its data pins. When data is supplied to data pins, a high-to-low pulse must be applied to this pin in order for the LCD to latch in the data present at the data pins.
7-14	**D0 to D7**	**Data Bits (0-7):** Pins used to send command or data bits to the LCD.	Data/ Comman d Pins	In 4-bit mode, only 4 pins (D0-D3) are connected to MCU digital output pins. In 8-bit mode, all 8 pins (D0-D7) are connected to MCU digital output pins.
15	**A**	**LED +** Anode of backlight LED is given positive voltage.	Backlight LED Pins	LED+ and LED- are connected to 5V and Ground pins of MCU with a current limiting resistor in series.
16	**K**	**LED -** Cathode is connected to ground to illuminate backlight LED.		

In this lab, we are interfacing the LCD with ATmega328P directly ***(parallel interface)*** and sending data in 8-bit mode. Note that the 8-bit data interfacing is easier to program but uses 4 more pins.

LCD Commands

The following table hex code for the commands that are sent to LCD instruction register for the specified functions.

Table 1: Hex code for Commands

Hex Code for Command to LCD	Function
0E	Display on, cursor on
01	Clear display screen
02	Return home
04	Decrement cursor (shift cursor to left
06	Increment cursor (shift cursor to right)
05	Shift display right
07	Shift display left
0F	Display on, cursor blinking
80	Force cursor to beginning of first line
C0	Force cursor to beginning of 2nd line
38	Function Set: 2 lines and 5 × 8 matrix (D0–D7, **8-bit mode**)
08	Display off, cursor off
18	Shift the entire display to the left
1C	Shift the entire display to the right

Interfacing LCD with ATmega328P and C-Programming

The Figure 2 shows required connections for LCD 16 pins with ATmega328P in 8-bit data mode. The R/W pin can be directly connected to ground instead of utilizing an I/O pin (as we are performing write operation only).

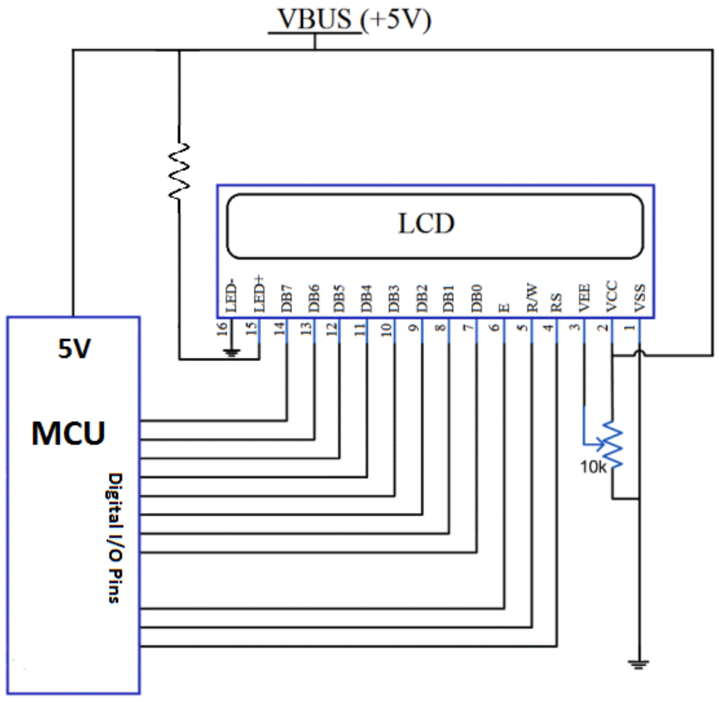


Figure 2: LCD Connections with ATmega328P for 8-bit Mode

For controlling the LCD and sending commands and data, the following steps are needed. Remember, the digital I/O pins connected with the LCD must be configured as output pins using DDRx registers as per your connections.

Initializing or Configuring the LCD:

To initialize the LCD for 2-line and 8-bit operation, the following sequence of commands should be sent to the LCD. Next we will show how to send a command to the LCD. After power-up you should wait about 15ms before sending initializing commands to the LCD. If the LCD initializer function is not the first function in your code you can omit this delay.

1) Function Set: 2 lines and 5 × 8 matrix (D0–D7, 8-bit mode)	- **0x38**
2) Display on, cursor blinking	- **0x0E**
3) Clear display screen	- **0x01**

After initialization, wait for 2msec.

Sending Command:

To send any of the commands from Table 1 to the LCD,
1) Make pin RS low for selecting command register. (R/W should be made low if not already grounded).
2) Put the command number on the data pins (D0–D7) i.e., use relevant PORTx register.
3) Send a high-to-low pulse to the E pin to enable the internal latch of the LCD.

Notice that after each command you should wait for some time (100us generally or for 2msec in some cases like clear screen and return home) to let the LCD module run the command.

Sending Data:

To send data to the LCD

1) Make pins RS = 1 (for data register) and R/W = 0.
2) Put the data on the data pins (D0–D7) i.e., use relevant PORTx register.
3) Send a high-to-low pulse to the E pin to enable the internal latch of the LCD.

Notice that after sending data you should wait about 100 μs to let the LCD module write the data on the screen.

Example Code and Subroutines

See the sample code for Example 1. It is written in in terms of **LCD_DPRT, LCD_DDDR, LCD_DPIN**. These will be replaced by the PORT, DDR and PIN register of the port with which D0-D7 are connected. In our example, it is port D. This is done to make the code more generalize, and to achieve this we have used **#define** directive (#define causes the compiler to substitute token-string for each occurrence of identifier in the source file). Similarly, **LCD_CPRT, LCD_CDDR, LCD_CPIN** are used to show the relevant registers of port with which we have connected control pins (RS and E). Here, it is port B. For the position of these pins is indicated by **LCD_RS** and **LCD_E** which is 0 and 1. If you change the hardware connections, you have to update the relevant registers only once (i.e., at the start with the #define directive) without changing the rest of the code or subroutines.

Based on the steps described earlier, 4 useful subroutines are defined before the main function.

1) ***lcdCommand*****:** It takes hex code of command as input.

2) ***lcdData*****:** It takes the data character to be displayed.

3) ***lcd_init*****:** This performs initialization steps.

4) ***lcd_print*****:** This takes a complete string to be printed and passes one character at a time by *lcdData*.

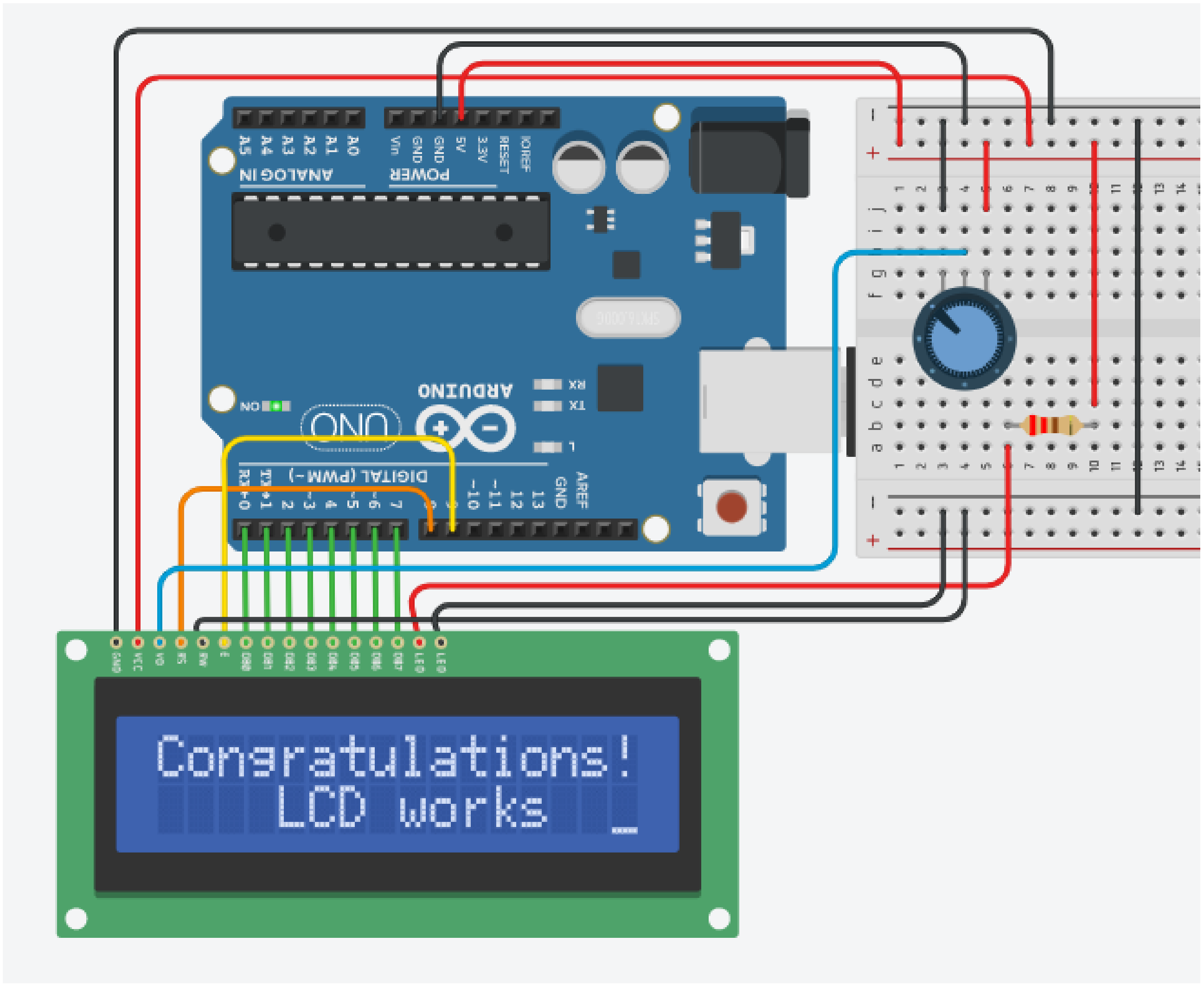


Figure 3: Connections for LCD Interfacing (8-bit Mode) with ATmega328P

Note that RW is connected with GND for writing.

```
#include <avr/io.h>
#include <util/delay.h>
#define LCD_DPRT PORTD //LCD DATA PORT
#define LCD_DDDR DDRD //LCD DATA DDR
#define LCD_DPIN PIND //LCD DATA PIN
#define LCD_CPRT PORTB //PORT for LCD Control Pins
#define LCD_CDDR DDRB //DDR for LCD Control Pins
#define LCD_CPIN PINB //PIN Reg for LCD Control Pins
#define LCD_RS 0 //LCD RS (RS is connected at PB0)
#define LCD_EN 1 //LCD EN (EN is connected at PB1)


void lcdCommand( unsigned char cmnd )
{
  LCD_DPRT = cmnd;           //send cmnd to data port
  LCD_CPRT &= ~ (1<<LCD_RS);  //RS = 0 for command
  LCD_CPRT |= (1<<LCD_EN);  //EN = 1 for H-to-L pulse
  _delay_us(1);              //wait to make enable wide
  LCD_CPRT &= ~ (1<<LCD_EN);  //EN = 0 for H-to-L pulse
  _delay_us(100);            //wait to make enable wide
}



void lcdData( unsigned char data )
{
      LCD_DPRT = data; //send data to data port
      LCD_CPRT |= (1<<LCD_RS); //RS = 1 for data
     LCD_CPRT |= (1<<LCD_EN); //EN = 1 for H-to-L pulse
     _delay_us(1); //wait to make enable wide
    LCD_CPRT &= ~ (1<<LCD_EN);
    //EN = 0 for H-to-L pulse
    _delay_us(100); //wait to make enable wide
}
```

```
void lcd_init()
{
   LCD_DDDR = 0xFF;//making data port (output)
   LCD_CDDR |= (1<<LCD_RS)|(1<<LCD_EN);
    //making control pins output pins
    LCD_CPRT &=~(1<<LCD_EN); //LCD_EN = 0
   _delay_ms(20); //wait for init.
   lcdCommand(0x38); //init. LCD 2-line, 8-bit mode
   lcdCommand(0x0F); //display on blinking
   lcdCommand(0x01); //clear LCD
   _delay_us(2000); //wait
   lcdCommand(0x06); //shift cursor right
}

void lcd_print( char * str )
{
unsigned char i = 0;
while(str[i]!=0)
  {
     lcdData(str[i]);
     i++ ;
     // _delay_ms(100); //for typing effect
   }

}

int main(void)
{
      lcd_init(); //initialize
      _delay_ms(1000);
      lcd_print("Congratulations!!");
      lcdCommand(0xC0); //Cursor at the start of 2nd line
      lcd_print("    LCD works  ");
      return 0;
}
```

Figure 4: Sample Code for Example 1 - LCD Interfacing (8-bit Mode) with ATmega328P

LAB TASKS

TASK 1: To test Example 1 using ATmega328P and 16x2 LCD

Program ATmega328P with the given code. Make connections to interface the LCD and test the results.

TASK 2: To test different commands for modifying LCD display

Modify the code and test different commands to make the display interesting.

- On line 1, display: <Your Name> *(You can have more than 16 characters in the string).*
- On line 2, display: <Roll # …… >
- Make the text moving (scrolling) continuously from left to right by using shift display commands.

Go-to Subroutine

Following is an interesting subroutine that allows you to move cursor at any specified location (y, x). Where x is the line number (1 or 2) and y is character position (1 to 16).

```
void lcd_gotoxy(unsigned char x, unsigned char y)
{
        char firstCharAdr[]={0x80,0xC0,0x94,0xD4}; //Table
        lcdCommand(firstCharAdr[y-1] + x - 1);
        _delay_us(100);
}
```

If you are interested, you can try this too!

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**
Laboratory Session No.: ________________________ Date: ______________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)

Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: *Imitate* and *practice* given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to *operate* software environment *under supervision*, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: *Detect* Errors/Exceptions and *manipulate,* under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: *Copy* or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 06

OBJECTIVE:

To utilize SPI (Serial Peripheral Interface) protocol for interfacing the max6675 module with ATmega328P and develop temperature measurement system based on the K-type thermocouple

LAB OUTCOMES:

By the end of this lab, you will be able to:

1) Recognize the difference between synchronous and asynchronous transmission
2) Identify the AVR ATmega328P pins associated with SPI communication
3) Identify the purpose of different fields of SPI registers
4) Program ATmega328P for SPI communication in ***master*** and ***slave*** modes for single-byte and multiple-byte burst read/write
5) Interface SPI protocol-based module (Max6675) as slave with ATmega328P in master mode
6) Develop the complete system for temperature measurement and transmit result through USART to PC

"Don't expect what you don't communicate clearly." — Anonymous

INTRODUCTION:

In Lab 04, we discussed briefly about serial and parallel data transmission. USART was utilized for ***asynchronous*** transmission and reception, which is one of the types of serial communication. In this lab, we will explore Serial Peripheral Interface communication protocol which is ***synchronous***. The SPI (serial peripheral interface) is a bus interface connection incorporated into many devices. The SPI bus was originally started by Motorola Corp. (now Freescale), but in recent years has become a widely used standard adapted by many semiconductor chip companies as it's faster, compact and results in reduced power consumption. Let's first understand the features of Serial Peripheral Protocol.

Serial Peripheral Interface Bus Protocol

SPI has a Master/Slave configuration. It has only **one** master device but can have **multiple slaves**. A master, that initiates communication, is usually a microcontroller and the slaves can be a microcontroller, sensors, ADC, DAC, LCD etc.

SPI is **4-wire** protocol.

- SPI devices use only **2 pins for data transfer** that are; SDI and SDO, also called MISO (Master-In Slave-Out) and MOSI (Master-Out Slave-In).
- The SPI bus has the **SCLK** or **SCK** (shift clock) pin to synchronize the data transfer between two chips.
- The last pin is **CE** chip enable, also called **SS Slave Select**, which is used to initiate and terminate the data transfer. It determines which device the master is currently communicating with.

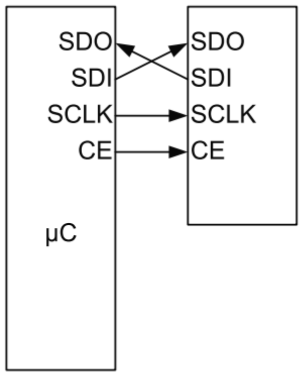


Figure 1: 4-wire SPI Bus Representation

Working of SPI

The system consists of two 8-bit wide shift registers, and a master clock generator. The SPI master initiates the communication cycle when pulling low the slave select $\overline{SS}$. Master and slave prepare the data to be sent in their respective shift registers, and the master generates the required clock pulses on the SCK line to interchange data (one-bit at a time in each clock cycle). In SPI, the shift registers are 8 bits long. It means that after 8 clock pulses, the contents of the two shift registers are interchanged. Data is always shifted from master to slave on the MOSI, line, and from slave to master on the master MISO, line. After each data packet, the master will synchronize the Slave by pulling high the slave select, $\overline{SS}$ line. It must be noted that SPI is full duplex, meaning that it sends and receives data at the same time.

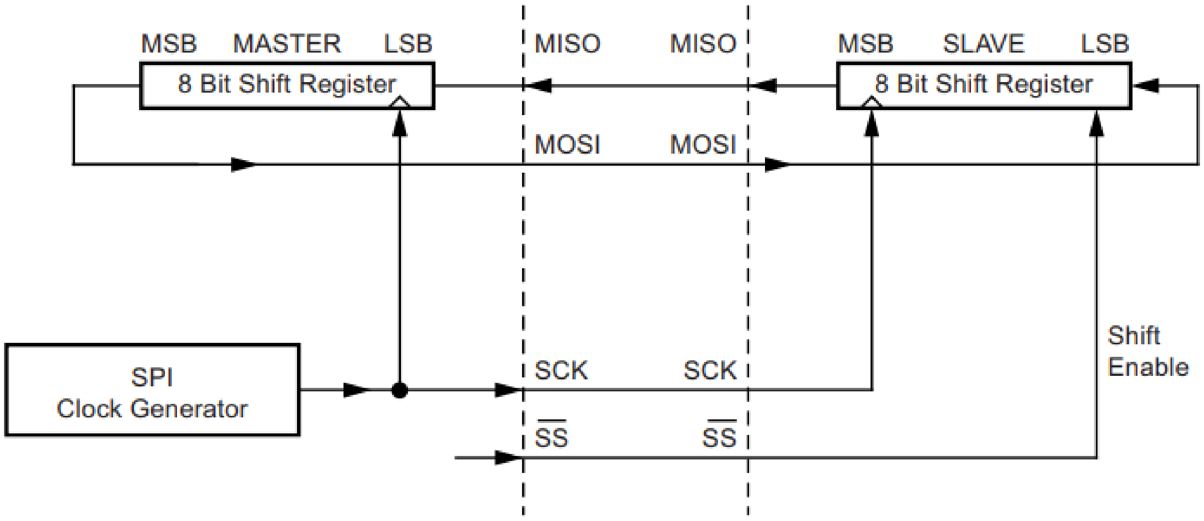


Figure 2: SPI Architecture and Master/Slave Interconnection

In connecting a device with an SPI bus to a microcontroller, we use the microcontroller as the master while the SPI device acts as a slave.

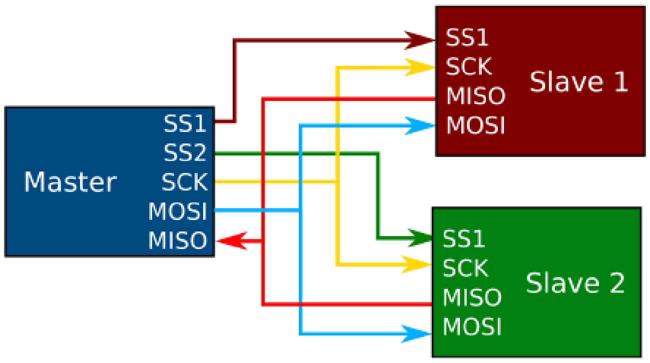


Figure 3: SPI Interface of 1-master and 2-slave devices

Serial Peripheral Interface of AVR ATmega328P Microcontroller

The serial peripheral interface (SPI) of AVR allows high-speed synchronous data transfer between the ATmega328P and peripheral devices or between several AVR devices. It can operate in master and slave modes and allows LSB first or MSB first transfer options. Figure 3 shows the pins associated with ATmega328 SPI and their connection pins on Arduino UNO board. Pin 11 or ICSP-4 is used as MOSI, Pin 12 or ICSP-1 is used as MISO, Pin 13 or ICSP-3 is connected with SCK and Pin 10 for SS.

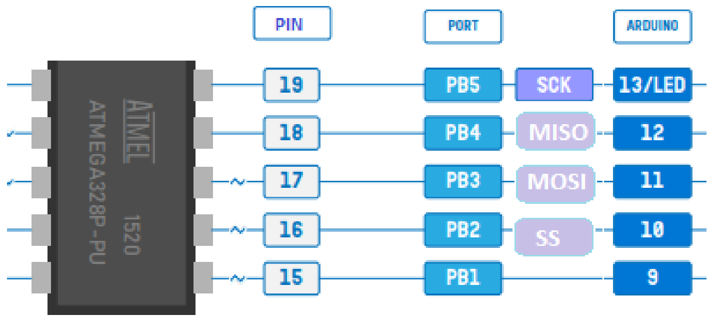


Figure 4: Pin Configuration of ATmega328P for SPI Interface and Associated Arduino UNO pinout

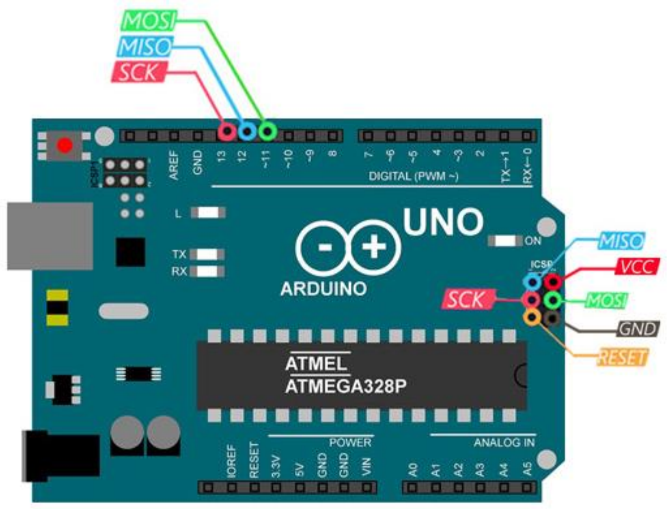


Figure 5: In-Circuit Serial Programming (ICSP) Header for SPI Communication

Table 1:

Pins		Use	Pin Configuration
Slave Select	SS	Used by Master device to enable and disable specific devices to communicate with. The SS pin is useful for packet/byte synchronization to keep the slave bit counter synchronous with the master clock generator. When the SS pin is driven high, the SPI slave will immediately reset the send and receive logic, and drop any partially received data in the shift register.	**Input** – For Slave **Output** – For Master (usually)
Master-In Slave-Out	MISO	For sending data from Slave devices to Master device.	**Input** – For Master **Output** – For Slave
Master-Out Slave-In	MOSI	For sending data from Master device to Slave devices.	**Input** – For Slave **Output** – For Master
Serial Clock	SCK	For clock pulses to synchronize data transmission from Master devices.	**Input** – For Slave **Output** – For Master

SPI Data Modes and Clock Phase with Polarity

As we mentioned before in USART communication, transmitter and receiver must agree on a clock frequency. In SPI communication, the master and slave(s) must agree on the CPOL (clock polarity) and CPHA (clock phase), with respect to the data.

CPOL	**0:** The base value of the clock is zero.	**1:** The base value of the clock is one.
CPHA	**0:** Sample (Read) on the first clock edge.	1: Sample (Read) on the second clock edge.

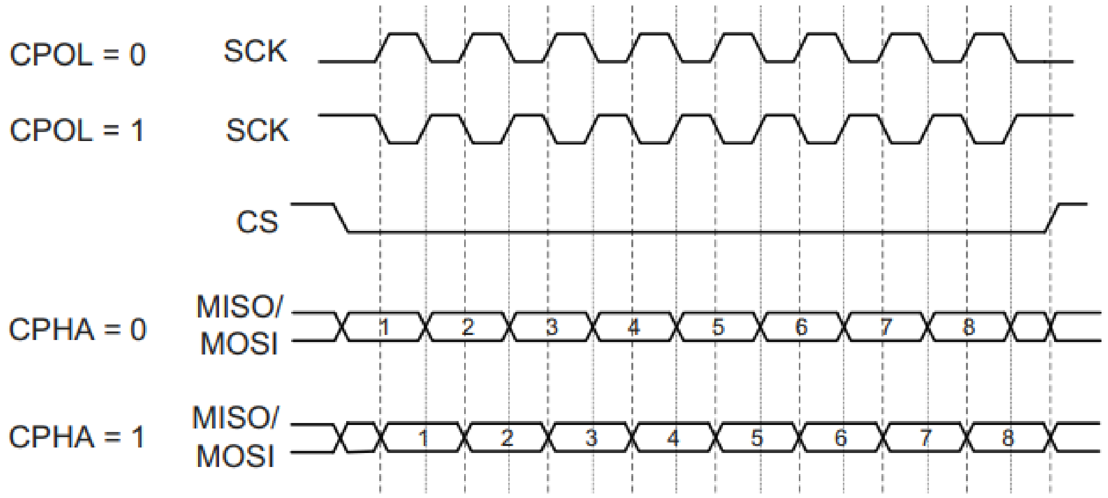


Figure 6: Transfer Format as per SPI Clock Polarity and Phase

Based on this, 4 different modes of SPI are available.

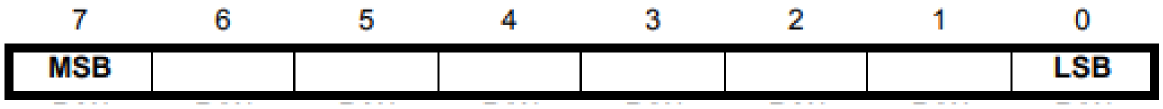
Table 2: SPI Modes

CPOL	CPHA	Data Read and Change Time	SPI Mode
0	**0**	Sample or read at rising edge. Setup or change data at falling edge.	**0**
0	**1**	Setup at rising edge. Sample at falling edge.	**1**
1	**0**	Sample at falling edge. Setup at rising edge.	**2**
1	**1**	Setup at falling edge. Sample at rising edge.	**3**

ATmega328P SPI Registers

In AVR three registers are associated with SPI. They are SPSR (SPI Status Register), SPCR (SPI Control Register), and SPDR (SPI Data Register).

1) SPDR (SPI Data Register)



The SPI Data Register is a read/write register. To write into SPI shift register, data must be written to SPDR. To read from the SPI shift register, you should read from SPDR. Writing to the SPDR register initiates data transmission.

2) SPCR (SPI Control Register)

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPIE	***SPI Interrupt Enable*** Setting this bit to one enables the SPI interrupt.
SPE	***SPI Enable*** Setting this bit to one enables the SPI
DORD	***Data Order*** The LSB is transmitted first if DORD is one; otherwise, the MSB is transmitted first.
MSTR	***Master/Slave Select*** 1: Selects master mode 0: Selects slave mode
CPOL	***Clock Polarity*** **0:** The base value of the clock is zero. **1:** The base value of the clock is one.
CPHA	***Clock Phase*** **0:** Sample (Read) on the first clock edge. 1: Sample (Read) on the second clock edge.
SPR1, SPR0	***SPI Clock Rate Select 1 and 0*** SPI2X, SPR1, and SPR0 are combined to make different clock frequencies for master.

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	Fosc/4
0	0	1	Fosc/16
0	1	0	Fosc/64
0	1	1	Fosc/128
1	0	0	Fosc/2 (Not recommended!)
1	0	1	Fosc/8
1	1	0	Fosc/32
1	1	1	Fosc/64

3) SPSR (SPI Status Register)

7	6	5	4	3	2	1	0
SPIF	WCOL	–	–	–	–	–	SPI2X

SPIF	***SPI Interrupt Flag*** This bit is set when a serial transfer is completed (in master mode if SS is configured as an output pin).
WCOL	***Write COLlision Flag*** The WCOL bit is set if you write on SPDR during a data transfer
SPI2X	***Double SPI Speed*** When the SPI is in master mode, setting this bit to one doubles the SPI speed.

Steps of Programming ATmega328P SPI

In accessing SPI devices, we have two modes of operation: **single-byte** and **multiple-byte burst**. To program ATmega328P SPI, the following steps are needed depending upon master or slave modes. The codes or subroutines are given here as sample for reference. In this lab, we will be operating our microcontroller in the master mode only.

Single-Byte Reading and Writing in Master and Slave Mode

➢ ATmega328P SPI Master Initialization

To initialize ATmega328P as Master, do the following steps

1) Make MOSI, SCK, and SS pins directions as output.
2) Make MISO pin direction as input.
3) Make SS pin High.
4) Enable SPI in Master mode by setting SPE and MSTR bits in the SPCR register.
5) Set SPI Clock Rate Bits combination to define SCK frequency and clock polarity and phase.

```
#include <avr/io.h>        //macros
#include <util/delay.h>
#define MOSI 3
#define MISO 4
#define SCK 5
#define SS 2
void spi_init_master( )
{
            // configuring SPI pins
   DDRB = (1<<MOSI) | (1<<SCK) | (1<<SS); //MOSI and SCK are output
   DDRB &= ~ (1 << MISO_BIT); // input
              // SPI Interrupt disabled
              // SPI enabled
              // Data order = MSB transmitted first
              // Master mode enabled
              // Clock polarity = Leading edge is rising
              // Clock phase = Data is sampled on trailing edge
              // SPI frequency = 16 Mhz / 16 = 1 Mhz
   SPCR = (1 << 0); // SPI clock rate select 0 (SPR0)
   SPCR |= (1 << 4); // Master/Slave select (MSTR)
   SPCR |= (1 << 2); // Clock phase (CPHA)
   SPCR |= (1 << 6); // SPI Enable (SPE)  // SPI clock rate bit for SPI clock to be 1 Mhz
   SPSR &= ~ (1 << 0); // Double SPI speed bit (SPI2X)
}
```

➢ SPI Master Write (Single-Byte)

Master writes data byte in SPDR. Writing to SPDR starts data transmission. 8-bit data starts shifting out towards slave and after the complete byte is shifted, SPI clock generator stops, and SPIF bit gets set. Follow the steps below:

Make SS low to select slave.

1. Load the 1 byte of data in the SPI shift register.

2. Wait till transmission is complete i.e., poll SPIF flag to become high.
3. Make SS high to deselect slave.

```
PORTB &= ~ (1 << SS);
void spi_write(char data)                /* SPI write data function */
{
        SPDR = data;                     /* Write data to SPI data register */
        while (!(SPSR & (1<<SPIF)));     /* Wait till transmission complete */
}
PORTB |= (1 << SS);
```

➢ **SPI Master Read (Single-Byte)**

1) Since writing to SPDR generates SCK for transmission, write dummy data in the SPDR register even if you don't want to send any data from master to slave.
2) Wait until the transmission is completed i.e. poll SPIF flag till it becomes high.
3) When the SPIF flag gets set, read requested received data in SPDR.

```
char spi_read( )                         /* SPI read data function */
{
        SPDR = 0xFF;
        while (!(SPSR & (1<<SPIF)));     /* Wait till reception complete */
        return (SPDR);                   /* Return received data */
}
```

➢ **ATmega328P SPI Slave Initialization and Read/Write Operation**

1) In slave mode there is no need to set SCK frequency because the SCK is generated by the master, but you must select the SPI mode (Clock Phase and Clock Polarity) and Data Order to match with SPI mode and Data Order of the other side (master device).
2) Make MOSI, SCK, and SS pins direction of the device as input.
3) Make MISO pin direction of the device as output.
4) Enable SPI in slave mode by setting SPE bit and clearing MSTR bit.

The Slave SPI interface remains in sleep as long as the SS pin is held high by the master. It activates only when the SS pin is driven low. Data is shifted out with incoming SCK clock from master during a write operation. SPIF is set after the complete shifting of a byte.

For the read and write operations in slave mode, almost the same steps are followed as in master mode.

```
void spi_init_slave( )                                   /* SPI Initialize function */
{
  DDRB &= ~ ((1<<MOSI)|(1<<SCK)|(1<<SS));     /* Make MOSI, SCK, SS as input pins */
  DDRB |= (1<<MISO);                     /* Make MISO pin as output pin */
  SPCR = (1<<SPE);                       /* Enable SPI in slave mode */
}
```

```
char spi_receive( )                    /* SPI Receive data function */
{
        while(!(SPSR & (1<<SPIF)));    /* Wait till reception complete */
        return(SPDR);                  /* Return received data */
}
void spi_write(char data)              /* SPI write data function */
{
        SPDR = data;                   /* Write data to SPI data register */
        while (!(SPSR & (1<<SPIF)));   /* Wait till transmission complete */
}
```

Multiple-Byte Burst Reading and Writing in Master and Slave Modes

Burst mode reading or writing is an effective way to share multiple bytes from master / slave at once. To do this,

1) Make SS low to select slave device.
2) Load the 1 byte of data in the SPI shift register.
3) Wait till transmission is complete.
4) Repeat step 2 and 3 till all bytes are transferred.
5) Make SS high to deselect slave.

We will practice this for interfacing our required module for this lab.

Max6675 Module with K-Type Thermocouple

The MAX6675 is a sophisticated thermocouple-to-digital converter with a built-in 12-bit analog-to-digital converter (ADC). The MAX6675 also contains cold-junction compensation sensing and correction, a digital controller, an SPI-compatible interface, and associated control logic.

Features

- Direct Digital Conversion of Type -K Thermocouple Output
- Cold-Junction Compensation: Simple SPI-Compatible Serial Interface
- 12-Bit, 0.25°C Resolution
- Open Thermocouple Detection

Datasheet: Max6675

The function of the thermocouple is to sense a difference in temperature between two ends of the thermocouple wires. The thermocouple's hot junction can be read from 0°C to +1023.75°C. The cold end (ambient temperature of the board on which the MAX6675 is mounted) can only range from -20°C to +85°C. While the temperature at the cold end fluctuates, the MAX6675 continues to accurately sense the temperature difference at the opposite end. The MAX6675 senses and corrects for the changes in the ambient temperature with cold-junction compensation. The device converts the ambient temperature reading into a voltage using a temperature-sensing diode. To make the actual thermocouple temperature measurement, the MAX6675 measures the voltage from the thermocouple's output and from the sensing diode. The device's internal circuitry passes the diode's voltage (sensing ambient temperature) and thermocouple voltage (sensing remote temperature minus ambient temperature) to the conversion function stored in the ADC to calculate the thermocouple's hot-junction temperature. The ADC adds the cold-junction diode measurement with the amplified thermocouple voltage and reads out the 12-bit result onto the SO pin. A sequence of all zeros means the thermocouple reading is 0°C. A sequence of all ones means the thermocouple reading is +1023.75°C.

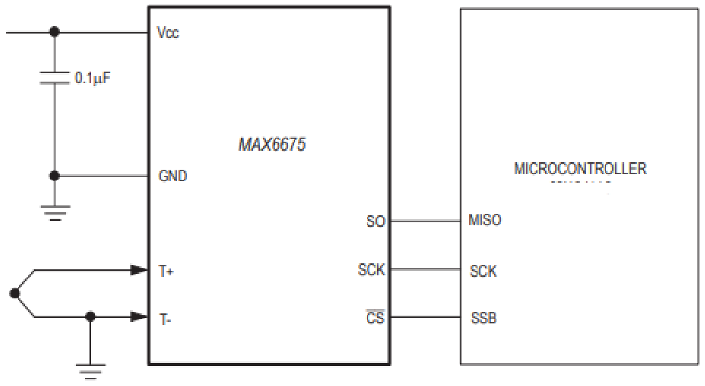


Figure 7: Typical connections of max6675 with microcontroller

As per the datasheet, following sequence of operation generates results from the max6675 module.

Force CS low to output the first bit on the SO pin. A complete serial interface read requires 16 clock cycles. Read the 16 output bits on the falling edge of the clock. The first bit, D15, is a dummy sign bit and is always zero. Bits D14–D3 contain the converted temperature in the order of MSB to LSB. Bit D2 is normally low and goes high when the thermocouple input is open. D1 is low to provide a device ID for the MAX6675 and bit D0 is three-state.

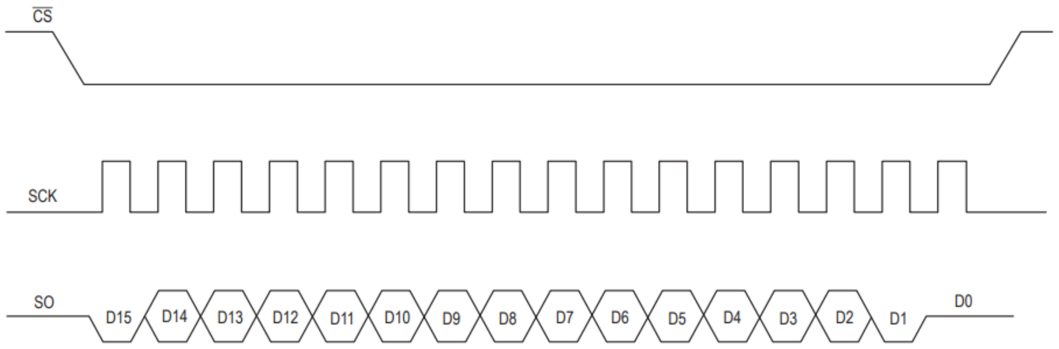


Figure 8: Serial Interface Protocol for Max6675

BIT	DUMMY SIGN BIT	12-BIT TEMPERATURE READING												THERMOCOUPLE INPUT	DEVICE ID	STATE
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	MSB											LSB		0	Three-state

Figure 9: SO Output from Max6675

Example 1: Programming ATmega328P SPI in master mode and measure temperature through Max6675

Based on the information regarding max6675 module, we need to interface it with ATmega328P through the SPI protocol. 2-byte burst read is required. Moreover, the 12 date bits from 16-bit output need to be extracted. The final result is then multiplied with resolution to get exact temperature. This is done in a sample code below.

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#define MOSI 3
#define MISO 4
#define SCK 5
#define SS 2

void spi_init()
{
   // configuring SPI pins
   DDRB = (1<<MOSI)|(1<<SCK)|(1<<SS); //MOSI and SCK are
output
   DDRB &= ~(1 << MISO); // input
   SPCR = (1 << 0); // SPI clock rate select 0 (SPR0)
   SPCR |= (1 << 4); // Master/Slave select (MSTR)
   SPCR |= (1 << 2); // Clock phase (CPHA)
   SPCR |= (1 << 6); // SPI Enable (SPE)
   // SPI clock rate bit for SPI clock to be 1 Mhz
   SPSR &= ~(1 << 0); // Double SPI speed bit
}

void spi_select()
{
   PORTB &= ~(1 << SS);
}

void spi_deselect()
{
   PORTB |= (1 << SS);
}

uint16_t spi_read16()
{
   uint16_t data;
   // select chip to enable data transfer
   spi_select();
   _delay_ms(1);
   // write dummy value in the SPI data register to read first
8 bits from slave
   SPDR = 0xFF;
   while ( !(SPSR & (1 << 7)) ){}; // wait till SPI interrupt flag
(SPIF) gets high
   // read data register
   data = SPDR;

   // left shifting data to 8 bits

data = data << 8;
   //write dummy value again in the SPI data register to read
last 8 bits from slave
   SPDR = 0xFF;
   while ( !(SPSR & (1 << 7)) ){} // wait till SPI interrupt flag
(SPIF) gets high
   // writing SPI data to lower 8 bits of the data variable
   data |= SPDR;
   // disable chip
   spi_deselect();
   return data;
}
```

```
float read_Thermocouple()
{
        uint16_t data;

        // read SPI data
        data = spi_read16();
        _delay_ms(1);

        // Bit 2 gets high if thermocouple input is open
        if (data & 0x4)
        return -1;

        // discarding 3 LSB bits
        data >>= 3;
        // factor taken from datasheet of MAX6675
        return data*0.25;
}

void usart_init(void)
{
   UBRR0=0x67; //set pre-scalar for baud rate (9600)
   UCSR0A &= ~(1 << U2X0); //Single Speed U2X - 0
   UCSR0B = (1 << RXEN0) | (1 << TXEN0); //
   UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // Async
mode,parity mode = disabled,1 stop bit,character size = 8 bit
   // clock polarity = 0 for async communication
}
void usart_putChar(unsigned char data)
{
   while( !(UCSR0A & (1 << UDRE0)) ){};   //wait for data
register to be empty
   UDR0 = data;                //write data
   while( !(UCSR0A & (1 << TXC0)) ){};   //wait for complete
}

void usart_putString(char* StringPtr)
{
   while(*StringPtr != 0x00)
   {
      usart_putChar((unsigned char)*StringPtr);
      StringPtr++;
   }
}
int main()
{
        float temp;
        char temp_string[10];
        spi_init();
        usart_init();
        while(1)
        {
                temp = read_Thermocouple();
                dtostrf(temp,5,2,temp_string);
                usart_putString("\n \r Temperature in
Celsius: ");
                usart_putString(temp_string);
                _delay_ms(1000);
        }
}
```

Task 1: To develop a temperature measurement system by interfacing Max6675 to microcontroller

1) Interface the Max6675 module with Arduino Uno by making all the required connections. Test the code given in Example 1 that measures the temperature and transmit it through USART to PC. Note that USART related subroutines from Lab 04 are utilized.
2) The serial terminal of PC should display room temperature value. Test it in different temperature conditions and observe the output.
3) Now modify the given code to add a 16x2 LCD screen to your system. The LCD should display the measured room temperature. This will make your complete temperature measurement system portable (independent of the PC display). Make appropriate connections for LCD interfacing. Carefully utilize the pins and ports of ATmega328P for utilizing SPI module along with the LCD connections.

Congratulations! You have successfully developed a small system that measures and displays accurate temperature from 0 to 1023°C.

Interesting Activity: *(Optional)

You can connect Oscilloscope probes to view SCK, MISO and SS signals at different channels simultaneously to verify the synchronous communication. Observe that SCK signal is only available when SS is set low and clock is generated. With respect to the clock signals, observe the data bits (D15 to D0). Read the 16-bit output from Max6675 and try to verify the temperature obtained using the extracted 12-bit number (D14-D3).

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**
Laboratory Session No.: ________________________ Date: ______________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: <u>*Imitate*</u> and <u>*practice*</u> given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to <u>*operate*</u> software environment <u>*under supervision*</u>, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: <u>*Detect*</u> Errors/Exceptions and <u>*manipulate,*</u> under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: <u>*Copy*</u> or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 07

OBJECTIVE:

To configure the Timer/Counter registers of AVR ATmega328P for generation of PWM (Pulse-Width Modulation) signals

LAB OUTCOMES:

By the end of this lab, you will be able to:

1) Understand the timers/counters of an AVR microcontroller and their different modes of operation.
2) Identify the AVR ATmega328P pins associated with the timer ports.
3) Identify the purpose of different fields of timer / counter registers.
4) Configure the timer/counter registers for generation of PWM signal in Fast PWM mode.
5) Program ATmgea328P for PWM signal generation and verify the pulse-width and duty cycle of the generated signals.
6) Control the position of a servo motor using PWM signal generated by ATmega328P.

"Getting into the habit of switching a ***timer*** *on will, I promise, save you from any number of kitchen disasters." — Delia Smith*

INTRODUCTION:

Microcontrollers have counter registers which can store the count of pulses from oscillator (clock) or any external signal. Such registers can be used as **Counter** or **Timer.** To count an event, the external event source can be connected to the pin of the counter register. The content of the counter is incremented whenever the external event occurs. The content of the counter represents how many times an event has occurred. To generate time delays, we connect the oscillator to the clock pin of the counter. The content of the counter is incremented when the oscillator ticks. Since the frequency of the oscillator in a microcontroller is known, and multiplying the time period with the count in the counter register, one can calculate the time elapsed. The flag is set when the counter overflows.

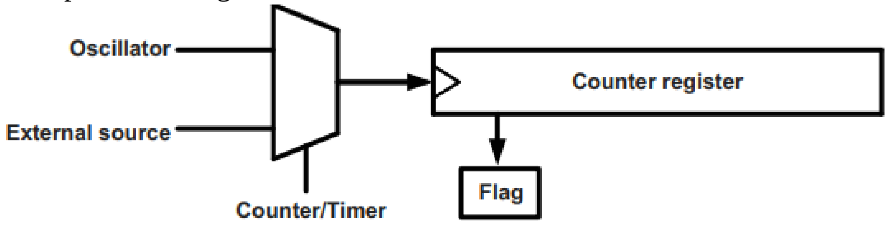


Figure 1: General View of Counters and Timers in Microcontrollers

Therefore, the timer modules of microcontrollers are used as <u>timers</u> to generate a time delay or as <u>counters</u> to count events happening outside the microcontroller. These timer modules have <u>waveform generation</u> support as well. In this lab, we will explore the ATmega328P timers, their modes of operation, and configuration for PWM (Pulse-Width Modulation) signal generation for given frequency and duty-cycle.

ATmega328P Timers and Basic Timer Registers

In ATmega328P, there are three timers, and all these are capable of generating PWM outputs.

- **Timer0** (8-bit wide)
- **Timer1** (16-bit wide)
- **Timer2** (8-bit wide)

TCNTn (Timer/Counter Register):

For each of the timer modules, there is a Timer/Counter register TCNTn (i.e. TCNT0, TCNT1 and TCNT2). This register stores the count and is cleared when reset is high. Each timer has a **Timer Overflow Flag (TOVn)** which is set high when TCNTn overflows.

OCRn (Output Compare Register):

Each timer also has an OCRn (Output Compare Register) register. The content of the OCRn is compared with the content of the TCNTn. When they are equal the OCFn (Output Compare Flag) flag will be set.

TCCRn (Timer/Counter Control Register):

The control registers are used for setting modes of operation.

The maximum and minimum values of TCNT are called TOP and BOTTOM, respectively.

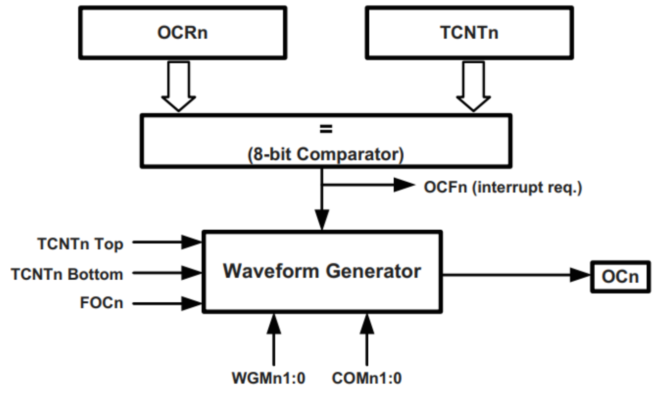


Figure 2: A General Representation of the Registers and Signals Associated with AVR times

In each timer module, there is a ***waveform generator***. The waveform generator can generate waves on the OCn pin.

We will look into the details of each of these for the three timers and their use to program timers.

ATmega328P Pins Associated with Timers/Counter Modules & PWM Generation

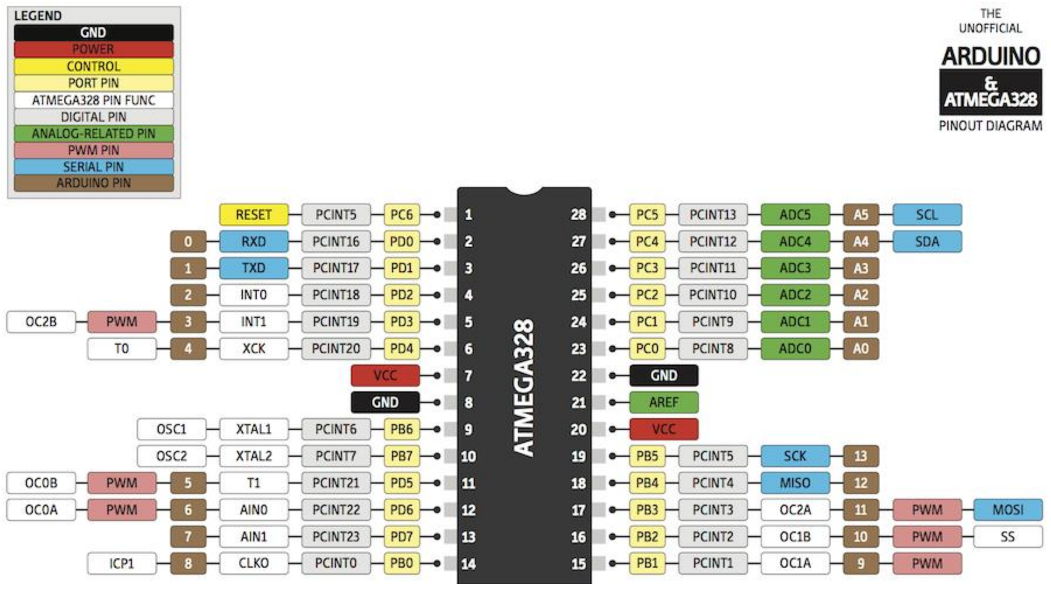


Figure 3: ATmega328P Pinout with Arduino Uno Connections

Modes of Operation of Timers/Counters

Modes of operation supported by the AVR Timer/Counter unit are:

- Normal mode (counter)
- Clear Timer on Compare Match (CTC) mode
- PWM modes (Fast PWM Mode, Phase Correct PWM Mode, and Phase and Frequency Correct PWM Mode)

Normal Mode

In this mode, the content of the timer/counter increments with each clock. It counts up until it reaches its max of 0xFF. When it rolls over from 0xFF to 0x00, it sets high a flag bit called TOV0 (Timer Overflow).

CTC (Clear Timer on Compare) Mode

The OCR0 register is used with CTC mode. As with the Normal mode, in the CTC mode, the timer is incremented with a clock. But it counts up until the content of the TCNT0 register becomes equal to the content of OCR0 (compare match occurs); then, the timer will be cleared and the OCF0 flag will be set when the next clock occurs.

In normal or CTC modes, the OC0 pin can perform one of the following actions for wave generation: (a) Remain unaffected (b) Toggle the OC0 pin (c) Clear (Drive low) the OC0 pin (d) Set (Drive high) the OC0 pin.

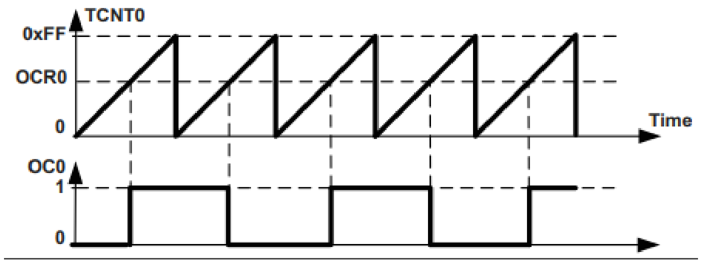


Figure 4: Sample Square Wave Generation in Normal mode (OC0 Toggle)

PWM Modes (Fast PWM Mode and Phase Correct PWM Mode)

The first output mode shown in Figure 5(a) represents the waveforms generated given a fast PWM setting where the TOP value is fixed at the maximum 8-bit value of 255. In this mode, two different output compare register values can be set independent of each other, each affecting a different output pin. That is, two separate PWM waveforms may be generated on two different port pins.

The second output mode shown in Figure 5(b) represents the waveforms generated given the phase-correct PWM setting where the TOP value is also fixed at the maximum 8-bit value of 255. As in the fast PWM case, two different output compare register values can be set independent of each other, each affecting their own output pin. As can be seen, this mode alters the TCNT register behavior as once the counter reaches the TOP value of 255, it begins counting backwards toward 0. The benefit has to do with the phase of the modulated carrier.

Notice the narrower pulses of OCB as compared to that of OCA in both Figure 5 (a-b). In the fast case, the front edges line up, whereas in the phase-correct case, the center of the pulses line up; that is, the phase of the OCA and OCB waveforms are equivalent. As a result, the period of the PWM waveform is nearly doubled. Due to the single-slope operation, the operating frequency of the fast PWM mode can be twice as high as the phase correct PWM mode that use dual-slope operation. This high frequency makes the fast

PWM mode well suited for power regulation, rectification, and DAC applications. High frequency allows physically small sized external components (coils, capacitors), hence reduces total system cost.

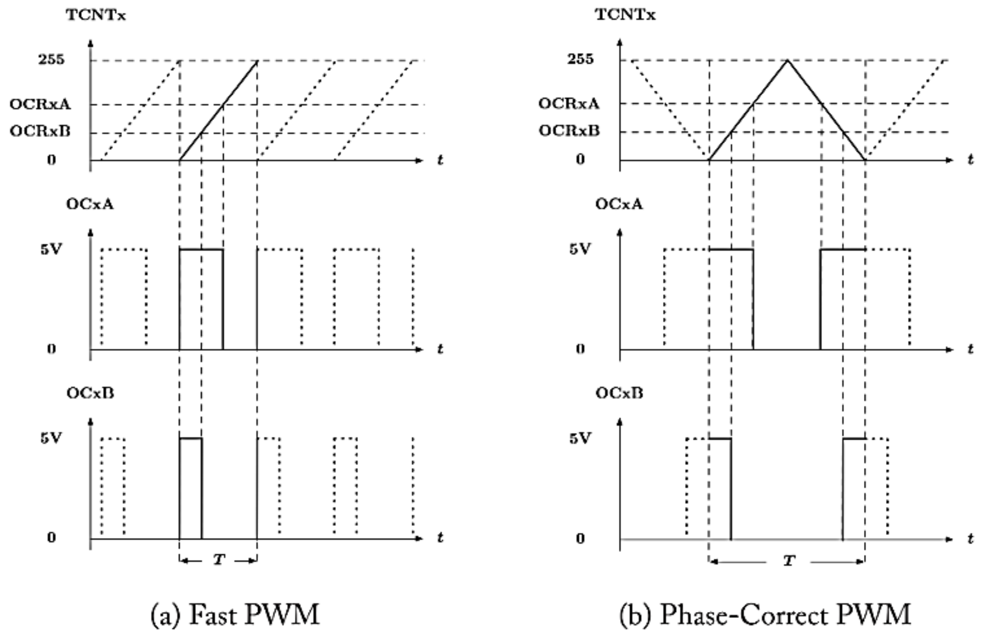


Figure 5: The two PWM configuration output waveforms with TOP = 255

The final two output modes shown in Figure 6 represent the fast and phase-correct PWM waveforms when the TOP value is set to the 8-bit value stored in OCRA. Both of these modes effectively disable the OCA pin functionality at the benefit of increasing the PWM frequency dramatically. In both cases, the TCNT register will count up to the OCRA value, and then either reset to 0 or start counting down toward 0. The only comparison that matters is that to OCRB, which will affect the OCB pin as in the previous cases. The two most significant results are that for a value loaded into OCRA, the total number of analog output levels available is reduced from 256 to OCRA + 1.

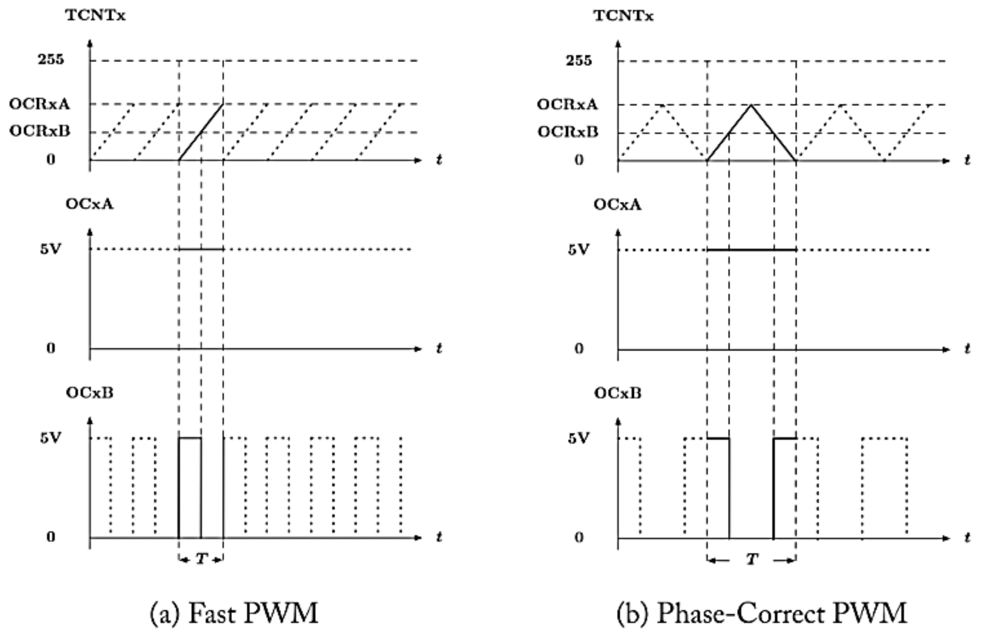
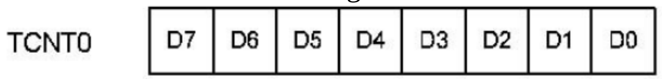


Figure 6: The two PWM configuration output waveforms with TOP = 255

Timer0 Registers and Programming

1) TCNT0 - Timer/Counter0 Register

Timer0, an 8-bit timer, has 8-bit wide timer/counter register called TCNT0.



TCNT0	D7	D6	D5	D4	D3	D2	D1	D0

2) OCR0A - Output Compare0 Register A

The Output Compare Register A contains an 8-bit value that is continuously compared with the counter value TCNT0. A match can be used to generate a waveform output on the **OC0A = PD6**.

3) OCR0B - Output Compare0 Register B

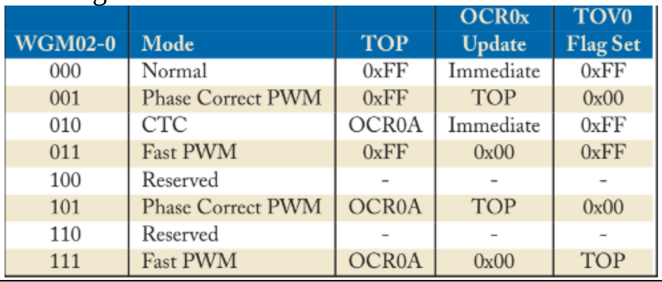
The Output Compare Register B contains an 8-bit value that is continuously compared with the counter value TCNT0. A match can be used to generate a waveform output on the **OC0B = PD5**.

4) TCCR0A (Timer/Counter0 Control Register A)

The different bits of this register used for controlling Timer0 are explained below.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00

WGM01-00	***Waveform Generation Mode*** Combined with the WGM02 bit found in the TCCR0B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used.
COM0A1-0	***Compare Output Mode for Channel A*** These control the output-compare pin (OC0A = PORTD6) behavior.
COM0B1-0	***Compare Output Mode for Channel B*** These bits control the output compare pin (OC0B = PORTD5) behavior.



WGM02-0	Mode	TOP	OCR0x Update	TOV0 Flag Set
000	Normal	0xFF	Immediate	0xFF
001	Phase Correct PWM	0xFF	TOP	0x00
010	CTC	OCR0A	Immediate	0xFF
011	Fast PWM	0xFF	0x00	0xFF
100	Reserved	-	-	-
101	Phase Correct PWM	OCR0A	TOP	0x00
110	Reserved	-	-	-
111	Fast PWM	OCR0A	0x00	TOP

COM0x1:0 bit functionality depends on the WGM02:0 bit setting. Here, the COM0 combinations are given for Fast PWM modes only (WGM is set in Mode 3 or 7).

WGM02:0 set to Fast PWM Mode

COM0A1-0

00	Normal port operation, OC0A disconnected

01	WGM02=0 (Mode 3): Normal port operation, OC0A disconnected
	WGM02=1 (Mode 7): Toggle OC0A on compare match

10	Non-inverting mode (Clear OC0A on compare match when up-counting).
11	Inverting mode (Set OC0A on compare match when up-counting).

COM0B1-0

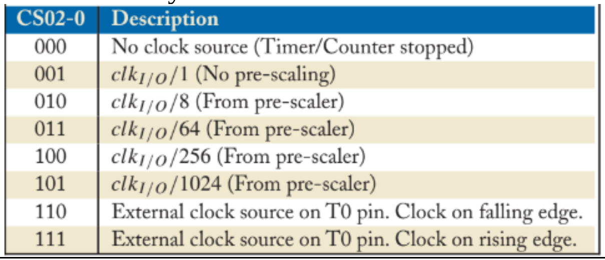
00	Normal port operation, OC0B disconnected
01	Reserved
10	Non-inverting mode (Clear OC0B on compare match when up-counting).
11	Inverting mode (Set OC0B on Compare Match. Clear at BOTTOM).

For COM bits in the other modes, refer to the datasheet of ATmega328P.

5) TCCR0B (Timer/Counter0 Control Register B)

The different bits of this register used for controlling Timer0 are explained below.

7	6	5	4	3	2	1	0
FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00

FOC0A	***Force Output Compare for Channel A and B***
FOC0B	When operating in PWM mode, these are set to 0. These are active in non-PWM mode only.
WGM02	***Waveform Generation Mode*** This bit along with WGM01-00 in TCCR0A set waveform generation mode.
CS02-00	***Clock Select*** Set the clock source to be used by Timer/Counter. 

CS02-0	Description
000	No clock source (Timer/Counter stopped)
001	$clk_{I/O}/1$ (No pre-scaling)
010	$clk_{I/O}/8$ (From pre-scaler)
011	$clk_{I/O}/64$ (From pre-scaler)
100	$clk_{I/O}/256$ (From pre-scaler)
101	$clk_{I/O}/1024$ (From pre-scaler)
110	External clock source on T0 pin. Clock on falling edge.
111	External clock source on T0 pin. Clock on rising edge.

Timer2 Registers and Programming

The Timer2 is also an 8-bit timer, therefore, it has the same registers with similar functionality as explained for the Timer0 above. The register and corresponding bits are named with 2 (for Timer2) instead of 0. The clock-select or pre-scalar combinations are different than that of Timer0. These are listed below.

- TCNT2 - Timer/Counter2 Register
- OCR2A - Output Compare2 Register A (**OC2A = PB3**)
- OCR2B - Output Compare2 Register B (**OC2B = PD3**)
- TCCR0A (Timer/Counter2 Control Register A)

7	6	5	4	3	2	1	0
COM2A1	COM2A0	COM2B1	COM2B0	-	-	WGM21	WGM20

- TCCR2B (Timer/Counter2 Control Register B)

7	6	5	4	3	2	1	0
FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20

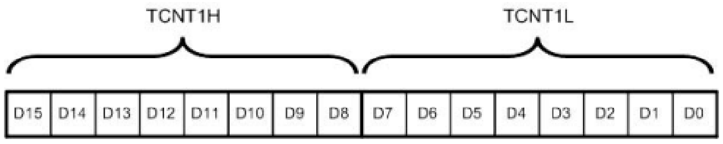
CS22	CS21	CS20	Pre-scalar
0	0	0	No Clock Source
0	0	1	1 (System Clock)
0	1	0	8
0	1	1	32
1	0	0	64
1	0	1	128
1	1	0	256
1	1	1	1024

Timer1 Registers and Programming

Timer1 is the only 16-bit timer in ATmega328P. It has 16-bit wide registers (TCNT1, OCR1A and OCR1B) and 3 8-bit wide control registers (TCCR1A, TCCR1B and TCCR1C). Timer1 unit allows accurate program execution timing (event management), wave generation, and signal timing measurement (input capture).

1) TCNT1 - Timer/Counter1 Register

The 16-bit timer/counter1 register is represented as:



2) OCR1A and OCR1B (Output Compare Register1 A and B)

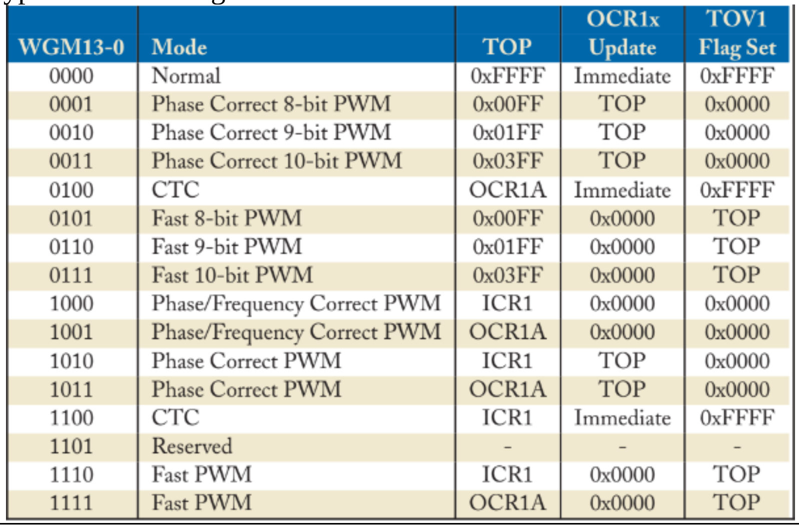
The Output Compare Register A contains a 16-bit value that is continuously compared with the counter value TCNT1. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the **OC1A = PB1.**

The Output Compare Register B contains a 16-bit value that is continuously compared with the counter value TCNT1. A match can be used to generate an Output Compare interrupt or to generate a waveform output on the **OC1B = PB2**.

3) TCCR1A (Timer/Counter1 Control Register A)

7	6	5	4	3	2	1	0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10

WGM11-0	***Waveform Generation Mode*** Combined with the WGM13:2 bits found in the TCCR1B Register, these bits control the counting sequence of the counter, the source for maximum (TOP) counter value, and what type of waveform generation to be used.

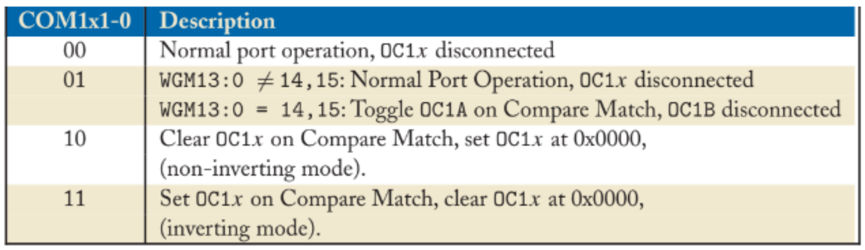


WGM13-0	Mode	TOP	OCR1x Update	TOV1 Flag Set
0000	Normal	0xFFFF	Immediate	0xFFFF
0001	Phase Correct 8-bit PWM	0x00FF	TOP	0x0000
0010	Phase Correct 9-bit PWM	0x01FF	TOP	0x0000
0011	Phase Correct 10-bit PWM	0x03FF	TOP	0x0000
0100	CTC	OCR1A	Immediate	0xFFFF
0101	Fast 8-bit PWM	0x00FF	0x0000	TOP
0110	Fast 9-bit PWM	0x01FF	0x0000	TOP
0111	Fast 10-bit PWM	0x03FF	0x0000	TOP
1000	Phase/Frequency Correct PWM	ICR1	0x0000	0x0000
1001	Phase/Frequency Correct PWM	OCR1A	0x0000	0x0000
1010	Phase Correct PWM	ICR1	TOP	0x0000
1011	Phase Correct PWM	OCR1A	TOP	0x0000
1100	CTC	ICR1	Immediate	0xFFFF
1101	Reserved	-	-	-
1110	Fast PWM	ICR1	0x0000	TOP
1111	Fast PWM	OCR1A	0x0000	TOP

COM1A1-0	***Compare Output Mode for Channel A*** These control the output-compare pin (OC1A = PB1) behavior.
COM1B1-0	***Compare Output Mode for Channel B*** These bits control the output compare pin (OC1B = PB2) behavior.

COM1x1:0 bit functionality depends on the WGM13:0 bit setting.

WGM13:0 set to Fast PWM Modes



COM1x1-0	Description
00	Normal port operation, OC1x disconnected
01	WGM13:0 ≠ 14,15: Normal Port Operation, OC1x disconnected WGM13:0 = 14,15: Toggle OC1A on Compare Match, OC1B disconnected
10	Clear OC1x on Compare Match, set OC1x at 0x0000, (non-inverting mode).
11	Set OC1x on Compare Match, clear OC1x at 0x0000, (inverting mode).

4) TCCR1B (Timer/Counter1 Control Register B)

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

INC1	***Input Capture Noise Canceler*** Setting this bit (to one) activates the Input Capture Noise Canceler which filters Input Capture Pin input (ICP1=PB0).
ICES1	***Input Capture Edge Select*** 0: Falling edge is used to trigger capture event at ICP1. 1: Rising edge is used.
WGM13-2	***Waveform Generation Mode***(Combined with WGM11-0 in TCCR1A)
CS12-10	***Clock Select:***

CS12-0	Description
000	No clock source (Timer/Counter stopped)
001	$clk_{I/O}/1$ (No pre-scaling)
010	$clk_{I/O}/8$ (From pre-scaler)
011	$clk_{I/O}/64$ (From pre-scaler)
100	$clk_{I/O}/256$ (From pre-scaler)
101	$clk_{I/O}/1024$ (From pre-scaler)
110	External clock source on T1 pin. Clock on falling edge.
111	External clock source on T1 pin. Clock on rising edge.

5) TCCR1C (Timer/Counter1 Control Register C)

7	6	5	4	3	2	1	0
FOC1A	FOC1B	-	-	-	-	-	-

FOC1A and FOC1B: ***Force Output Compare for Channel A and B***
These are active in non-PWM mode.

6) ICR1 (Input Capture Register1)

The Input Capture Register can be used for defining counter TOP value. Otherwise, it shows the count of event occurrence on the Input Capture Pin (ICP1=PB0).

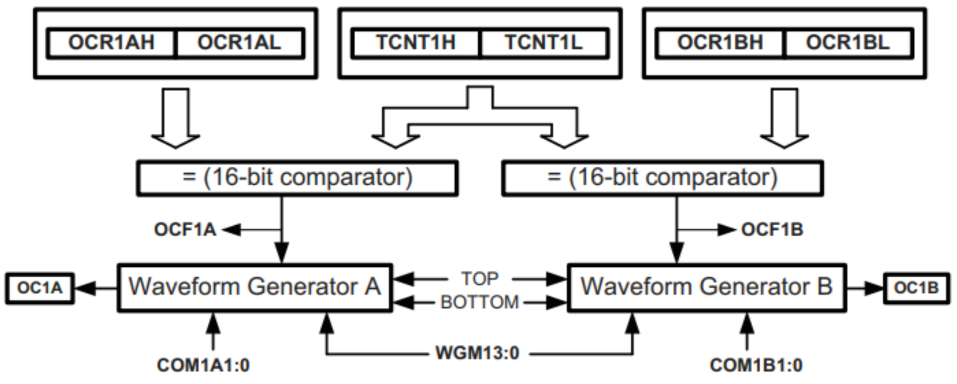


Figure 7: Representation of 16-bit Timer Module and Associated Signals and Pins

Programming 8-Bit Timers (Timer0/2) in Fast PWM Mode for PWM Signal Generation

1) Select modes using COM and WGM bits, for example; fast PWM (mode 3 or 7) and non-inverting (COM0x10= 0b10).
2) Select a pre-scalar by CS bits.
3) In Fast PWM mode, use the pins associated with the timer. Configure them as output pins.

- Timer0: OC0A (=PD6, Arduino Pin 6) / OC0B (=PD5, Arduino Pin 5)
- Timer2: OC2A (=PB3, Arduino Pin 11) / OC2B (=PD3, Arduino Pin 3)

4) The frequency of generated PWM signal is calculated as:

$$f_{desired} = \frac{system\_clock}{prescalar\ (1 + TOP)}$$

5) TOP is dependent on Mode. In Mode 3, it is fixed to 255 (0xFF).
6) In Mode 7, PWM mode works with a Compare Match. Top is the value stored in OCRxA. Calculate TOP i.e., OCRxA.

   The PWM wave of set frequency will be generated at OCxB as per the value of OCRxB.

   Mode 7 allows you to set frequency easily through OCRxA for a given pre-scalar. In mode 3, the frequency is fixed for any given pre-scalar as TOP is fixed.

7) Set OCRxA or OCRxB registers to achieve required duty-cycle. Lowering the top value can increase the PWM base frequency, but reduces the resolution. It’s easier to set duty-cycle in mode 3.
   For COM=10, the output will be set high for N+1, where N is value of OCR. Total cycles are determined by TOP+1. Where, TOP = 0xFF (255) in mode 3 and TOP=OCRxA in mode 7.

$$Duty\ Cycle = \frac{OCRx + 1}{TOP + 1}$$

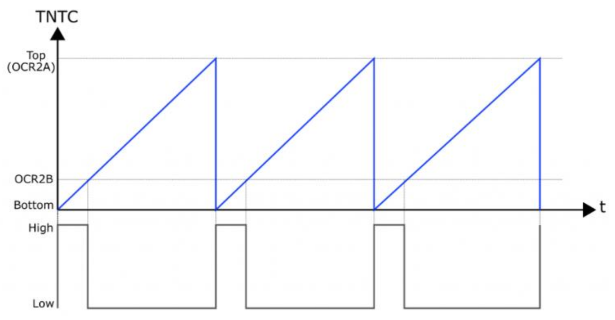
Example 1: Generating PWM Signal Using Timer2

The example code given here generates a square wave signal with a frequency of 1 kHz at OC2B. In each cycle, the signal will be high 20% of the time and 80% low. Another expression for this is: the duty cycle is 20%.

CODE

```
// Period = 1 ms => Frequenz = 1kHz
#include <avr/io.h>
int main(void)
{
  // WGM22/WGM21/WGM20 all set
 //Mode 7, fast PWM
TCCR2A = (1<<COM2B1) | (1<<WGM21) |
(1<<WGM20);
// Set OC2B at bottom,
//clear OC2B at compare match
  TCCR2B = (1<<CS22) | (1<<WGM22);
// prescaler = 64;
  OCR2A = 249;
  OCR2B = 49;
  DDRD |= (1<<PD3);
  while (1) {};
}
```

SAMPLE OUTPUT

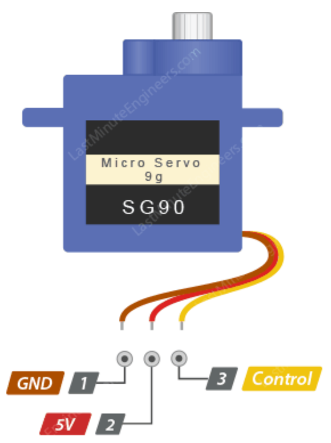


Note: Observe that in the above code, the timer registers are once initialized for PWM generation outside the while loop. In comparison to that, one can use the digital I/O pins for generation of PWM signal by setting them high for certain time and then low, using the delay functions repeatedly in the while loop. But that approach will use CPU itself to create the equivalent of PWM outputs. The advantage of using the built-in PWM feature of the AVR is that it gives us the option of programming the period and duty cycle, therefore relieving the CPU to do other important things.

PWM Signals for DC Motor Speed Control and Servo Motor Position Control

Pulse-width modulation (PWM) is a technique for controlling the speed of a DC motor by varying the width of the pulses that are applied to the motor's power supply. The duty cycle determines the average voltage that is applied to the motor. A higher duty cycle results in a higher average voltage, which in turn results in a higher motor speed.

A servo motor is a motor whose shaft position can be controlled precisely. The motor has an internal servomechanism that provides feedback about the position of shaft. SG90 is a small “Servo Motor” whose position can be controlled by a PWM signal. The required duty-cycle for different positions of this servo motor are given in Table below. The power requiements of SG90 can be met by Arduino Uno board. Utilizing the timers of ATmega328P, you can easily generate the required PWM signals for controlling the position of SG90.



- Required Frequency=50 Hz
- At Duty cylce ~ 5 %, Shaft Position = 0 º
- Duty cycle ~ 7.5%, Shaft Position = 90 º
- Duty cycle ~ 10%, Shaft Position = 180 º

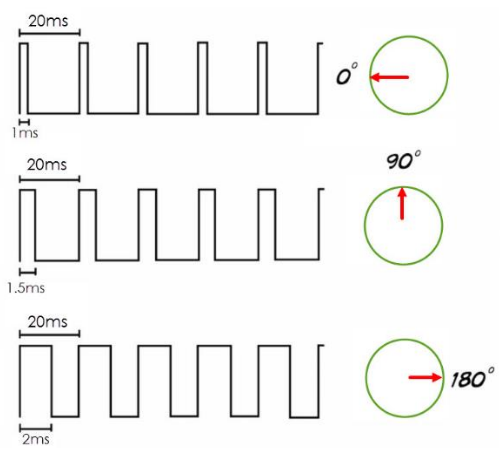


Figure 8: SG90 Servo-Motor Pinout and PWM Signal Requirements

Programming 16-Bit Timer (Timer1) in Fast PWM Mode

1) Select the mode of operation, i.e., the behavior of the Timer/Counter and the output compare pins, by the combination of the waveform generation mode (WGM13:0) and compare output mode (COM1x1:0) bits.
   Note:
   Fast PWM modes (WGM13:0 = 5, 6, 7, 14, or 15): In fast PWM mode the counter is incremented until the counter value matches either one of the fixed values 0x00FF, 0x01FF, or 0x03FF (WGM13:0 = 5, 6, or 7), the value in ICR1 (WGM13:0 = 14), or the value in OCR1A (WGM13:0 = 15).
   The PWM resolution for fast PWM can be fixed to 8-, 9-, or 10-bit, or defined by either ICR1 or OCR1A. The minimum resolution allowed is 2-bit (ICR1 or OCR1A set to 0x0003), and the maximum resolution is 16-bit (ICR1 or OCR1A set to MAX).
   Using the ICR1 register for defining TOP works well when using fixed TOP values. By using ICR1, the OCR1A register is free to be used for generating a PWM output on OC1A. However, if the base PWM frequency is actively changed (by changing the TOP value), using the OCR1A as TOP is clearly a better choice due to its double buffer feature which allows it to be written anytime.
2) Set TCCR1A and TCCR1B according to the selected modes.
3) Configure the associated OC1x pin as output pin.
4) Set the TOP by initializing ICR1 or OCR1A as per the selected modes and required frequency.

$$f_{desired} = \frac{system\_clock}{prescalar\ (1 + TOP)}$$

5) Set the duty-cycle by initializing OCR1A / OCR1B.

$$Duty\_Cycle = \frac{OCR1x + 1}{(1 + TOP)}$$

Example 2: Generating PWM Signal using Timer 1 for Servo-Motor Position Control

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
    {
        DDRB |= (1<<PB1); // Set PB1 as output
        TCCR1A |= (1<<COM1A1) | (1<<WGM11); // Fast PWM, non-inverting mode
        TCCR1B |= (1<<WGM13) | (1<<WGM12) | (1<<CS11);
                                    // Fast PWM, prescaler = 8
        ICR1=39999;   //20ms PWM period - TOP
        while (1){
            OCR1A = 1999; // Set position to 0 degrees
            _delay_ms(1000);
            OCR1A = 2999; // Set position to 90 degrees
            _delay_ms(1000);
            OCR1A = 3999; // Set position to 180 degrees
            _delay_ms(1000);
        }
    }
```

LAB TASKS

TASK 1: To verify PWM signal generation by 8-bit Timers using Example 1

1) Create an AVR project and build the code given in Example 1. Program the microcontroller with it and test the PWM signal generated at PD3 using an Oscilloscope.
2) Measure the frequency, time-period and duty-cycle of the generated waveform for verification.
3) Modify the code to generate a waveform at 2k Hz frequency with duty-cycle of 50% using Timer0. Show the modified code and results.

TASK 2: To test Example 2 for servo-motor position control using Timer1 PWM signal

1) Create an AVR project and build the code given in Example 2. Program the microcontroller with it.
2) Make appropriate connections to power up the SG90 servo-motor and to provide PWM signal for control.
3) Observe the position of motor-shaft. Does it rotate by 90 degrees after every 1 second?
4) Can you control the SG90 servo motor using Timer0 or Timer2? What is the minimum frequency achievable by Timer0, 1 and 2 in Fast PWM Modes? Consider the higher pre-scalar.

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**
Laboratory Session No.: ________________________ Date: ______________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: <u>*Imitate*</u> and <u>*practice*</u> given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to <u>*operate*</u> software environment <u>*under supervision*</u>, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: <u>*Detect*</u> Errors/Exceptions and <u>*manipulate,*</u> under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: <u>*Copy*</u> or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 08

OEL (Open Ended Lab)

OBJECTIVE:

To interface analog voltage sensor ZMPT101B for measurement of phase voltage and display its true RMS (Root Mean Square) value on LCD (Liquid Crystal Display) screen.

DELIVERABLES:

- Report
- C-Code
- Complete hardware setup (ZMPT101B module and LCD interfaced with Microprocessor)

NED University of Engineering & Technology
Department of Electrical Engineering

Course Code: **EE-354** Course Title: **Embedded Systems**
Laboratory Session No.: ________________________ Date: ______________________________

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Software Initialisation and Configuration: Set up and *recognise* software initialisation and configuration steps	Completely unable to recognise initialisation and configuration	Able to recognise initialisation but could not configure	Able to recognise initialisation but configuration is erroneous	Able to recognise initialisation and configuration with minimal errors	Able to recognise initialisation and configuration with complete success
Equipment Identification and Handling: *Sensory* skill to identify equipment and its components along with adherence to safe handling	Completely unable to identify equipment and components and no regard to safe handling	—	Ability to identify equipment but makes mistakes in recognising components, demonstrates decent equipment handling capacity	—	Ability to identify equipment and recognises all components, practices careful and safe handling
Establish and Verify Hardware-Software Connection: *Recognise* interface between computer and hardware kit and *establish* connectivity with software	Unable to perform hardware and software connection verification	—	Able to verify hardware connection but unable to establish software connection verification	—	Able to verify hardware connection and successfully establishes software connection verification
Following step-by-step procedure to complete lab work: *Observe, imitate and operate* hardware in conjunction with software to complete the provided sequence of steps	Inability to recognise and perform given lab procedures	Able to recognise given lab procedures and perform them but could not follow the prescribed order of steps	Able to recognise given lab procedures and perform them by following prescribed order of steps, with frequent mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with occasional mistakes	Able to recognise given lab procedures and perform them by following prescribed order of steps, with no mistakes

Psychomotor Domain Assessment Rubric for Laboratory (Level P3)					
Skill(s) to be assessed	Extent of Achievement				
	0	1	2	3	4
Programming the Controller for given Embedded System Problem: *Imitate* and *practice* given embedded C instructions for implementing specific control strategy and store required variables	Incorrect selection and use of programming constructs and instructions	Correct selection of programming constructs and instructions but their use is incorrect	Correct selection and use of programming constructs and instructions with many syntax/logical errors	Correct selection and use of programming constructs and instructions with little to no syntax/logical errors	Correct selection and use of programming constructs and instructions with no syntax/logical errors
Software Menu Identification and Usage: Ability to *operate* software environment *under supervision*, using menus, shortcuts, instructions etc.	Unable to understand and use software menu	Little ability and understanding of software menu operation, makes many mistake	Moderate ability and understanding of software menu operation, makes lesser mistakes	Reasonable understanding of software menu operation, makes no major mistakes	Demonstrates command over software menu usage with occasional use of advance menu options
Detecting and Removing Errors/Exceptions in Hardware and Software: *Detect* Errors/Exceptions and *manipulate,* under supervision, to rectify the embedded C program	Unable to check and detect error messages in software and hardware	Able to find error messages in software but no sense of hardware error identification	Able to find error messages in software and recognise them on hardware. Still unable to understand the error type and possible causes	Able to find error messages in software and recognise them on hardware. Moderately able in understanding error type and possible causes	Able to find error messages in software and recognise them on hardware. Reasonably able in understanding error type and possible causes
Visualisation, Comparison and analysis of results: *Copy* or enter results in analysis software to visualise and compare them with inputs. Use analysis tools to compute standard indices from result	Unable to understand and utilise visualisation, plotting and analysis software	Ability to understand and utilise visualisation and plotting instructions with errors. Unable to compute standard indices	Ability to understand and utilise visualisation and plotting instructions with occasional errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions with no errors. Able to partially compute standard indices	Ability to understand and utilise visualisation and plotting instructions without errors. Able to compute standard indices completely

LAB SESSION 09

OBJECTIVE:

To set up Inter-Integrated Circuit (I2C) communication on Atmega328P micro-controller for controlling a 16x2 LCD screen through PCF8574 I2C I/O (Input/Output) expander

LAB OUTCOMES:

By the end of this lab, you will be able to:

1) Explain the Inter-Integrated Communication (I2C) protocol
2) Identify the AVR ATmega328P pins associated with I2C interface (Two-Wire Interface TWI)
3) Identify the purpose of different fields of TWI registers
4) Program ATmega328P TWI in master and slave modes for single-byte and multiple-byte burst read/write
5) Operate a 16x2 LCD in 4-bit mode
6) Identify the pins of PCF8574 I2C I/O expander module
7) Program ATmega328P TWI for controlling 16x2 LCD screen through PCF8574 expander

"A lack of communication breeds assumptions of what the other is thinking or feeling; and assumptions are, more often than not incorrect" — Misty Lynn Walker

INTRODUCTION:

The Inter-Integrated Circuit (I^2C or I2C or IIC) serial communication protocol was created by Philips to attach low-speed peripherals to an embedded micro-processor for reliable short-distance communication. I2C is a multi-point protocol in which more than two devices are able to communicate along the serial interface. It uses only 2 pins for data transfer. They are called:

- **SCL (**Serial Clock), which synchronizes the data transfer between two chips
- **SDA** (Serial Data), which carries the data.

These two pins, SDA and SCK, make the I2C a 2-wire interface. In many application notes, including AVR datasheets, I2C is referred to as **Two-Wire Serial Interface (TWI).** We will be using I2C and TWI interchangeably.

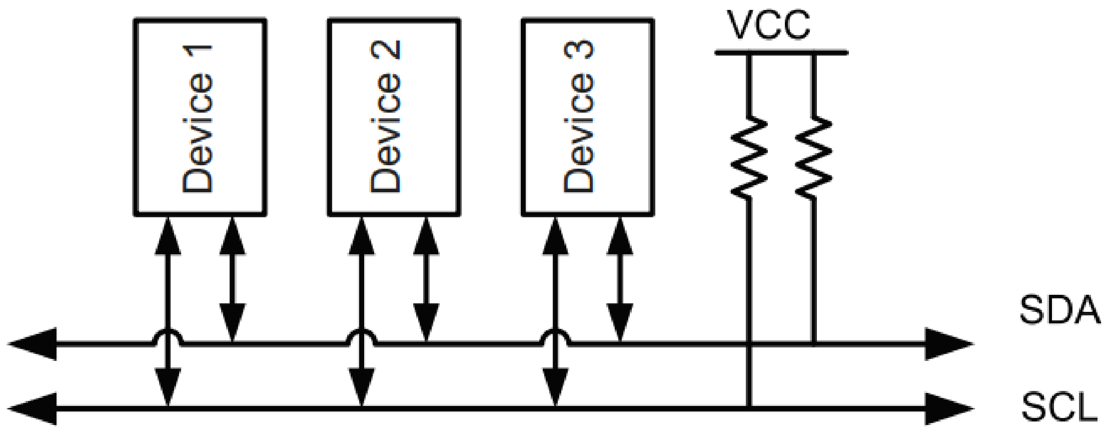


Figure 1: I2C Bus Interface Representation

The 2 pins are bidirectional open-drain pins which means that a 4.7 kilo Ohm pull-up resistor for each of line is needed as shown in Figure 1. If one or more devices pull the line to low (zero) level, the line state is zero and otherwise the line state remains high (one).

Each of the devices connected with I2C multipoint bus interface is called a **node**. Each node can operate as either master or slave.

- **Master** is a device that generates the clock for the system; it also initiates and terminates a transmission.
- **Slave** is the node that receives the clock and is addressed by the master.

In I2C, both master and slave can receive or transmit data, so there are **four modes of operation**.

- Master transmitter
- Master receiver
- Slave transmitter
- Slave receiver.

Notice that each node can have more than one mode of operation at different times, but it has only **one mode of operation at a given time**.

Working of I2C Protocol

I2C is a **synchronous** serial protocol; each data bit transferred on the SDA line is **synchronized** by a high-to-low pulse of clock on the SCL line. The data line cannot change when the clock line is high; it can change only when the clock line is low. The START and STOP conditions are the only exceptions to this rule.

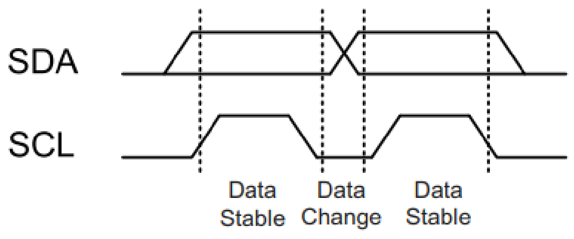


Figure 2: I2C Bit Format

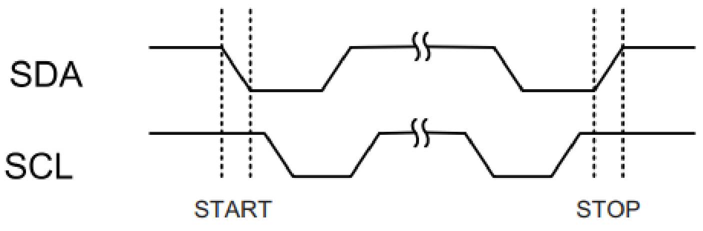


Figure 3: I2C Start and Stop Conditions

Each transmission is initiated by a START condition and is terminated by a STOP condition. The START and STOP conditions are generated by the master. START and STOP conditions are generated by keeping the level of the SCL line high and then changing the level of the SDA line. The START condition is generated by a high-to-low change in the SDA line when SCL is high. The STOP condition is generated by a low-to-high change in the SDA line when SCL is low. The bus is considered busy between each pair of START and STOP conditions, and no other master tries to take control of the bus when it is busy.

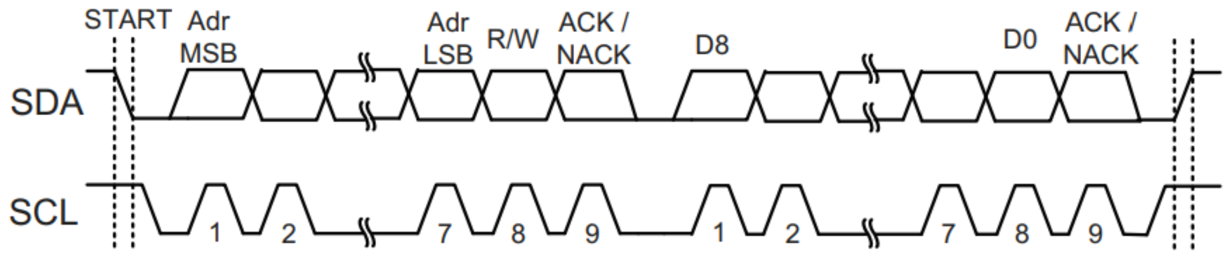


Figure 4: I2C Typical Transmission (Start + Address Packet + Data Packet(s) + Stop)

- In I2C, normally, a transmission is started by a START condition.

- This is followed by an address packet (SLA + R/W). Address packet consists 8 or 9 bits. The first 7-bits are slave address (which allows connection of 128 devices). The 8th bit shows operation (1 for read, 0 for write). The 9th bit is an ACK (acknowledge - 0) or NACK (not acknowledge - 1) by the receiver. ACK means that it is ready to receive the data byte.

Operation	Address Bits + Control (SLA+R/W)
Master writes data to SDA (sent to slave)	**SLA +W** A_6 A_5 A_4 A_3 A_2 A_1 A_0 **0**
Master reads from data from SDA (sent by slave)	**SLA + R** A_6 A_5 A_4 A_3 A_2 A_1 A_0 **1**

- The address packet is followed by one or more data packets which are also 9 bits long. The first 8 bits are a byte of data to be transmitted, and the 9th bit is ACK/NACK.
- The transmission is finished by a STOP condition.

AVR ATmega328P Two-Wire Interface (TWI) Module

The TWI module in the AVR is composed of four submodules: bit rate generation unit, bus interface unit, address match unit, and control unit. The bit rate generation unit controls the frequency of the system clock (SCL) when operating in a master mode. The bus interface unit detects and generates START, REPEATED START and STOP conditions. It also detects arbitration, controls sending or receiving ACK, and also transfers packets of data or address. The address match unit compares the received address byte with the 7-bit address in TWI address register and informs the control unit upon an address match. The control unit controls the TWI module and generates responses according to settings in the TWI control register. It also sets the contents of the status register according to current state.

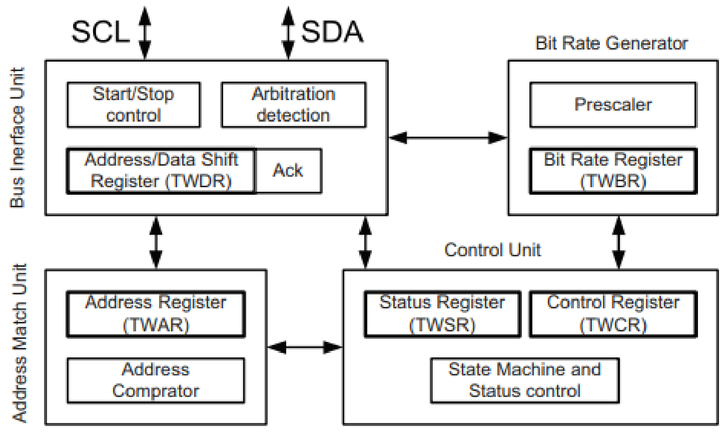


Figure 5: TWI (I2C) Module in AVR

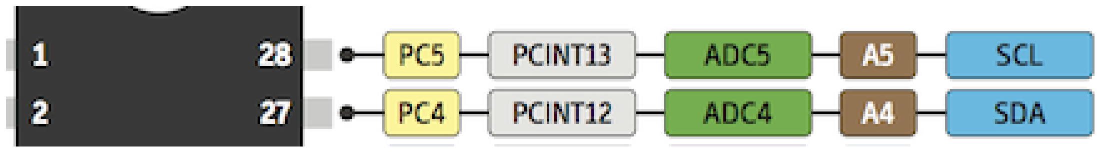


Figure 6: ATmega328P TWI SDA and SCL Pins

The ATmega328P uses **pins 27 and 28** for the TWI data (SDA) and clock (SCL), respectively.

ATmega328P TWI Registers

1) TWBR (TWI Bit Rate Register)

TWBR (8-bit register) selects the division factor for the bit rate generator. The bit rate generator is a frequency divider which generates the SCL clock frequency in the Master modes.

$$f_{SCL} = \frac{16\ MHz}{16 + 2(TWBR)(Prescalar\ Value)}$$

2) TWDR (TWI Bit Data Register)

In Transmit mode, TWDR contains the next byte to be transmitted. In Receive mode, the TWDR contains the last byte received.

3) TWCR (TWI Control Register)

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
TWINT	**TWEA**	**TWSTA**	**TWSTO**	**TWWC**	**TWEN**	**-**	**TWIE**

TWINT	***TWI Interrupt Flag*** This bit is set by hardware when the TWI has finished its current job (like start, stop, transmit, receive, etc.). The TWINT flag must be cleared by software by writing a logic one to it. Clearing this flag starts the operation of the TWI.
TWEA	***TWI Enable Acknowledge Bit*** If the TWEA bit is written to one, the ACK pulse is generated on the TWI bus if the device's own slave address has been received.
TWSTA	***TWI START Condition Bit*** To initiate master mode, TWSTA bit is written 1. The TWI hardware checks if the bus is available and generates a START condition on the bus if it is free.
TWSTO	***TWI STOP Condition Bit*** Writing the TWSTO bit to one in Master mode will generate a STOP condition. When the STOP condition is executed on the bus, the TWSTO bit is cleared automatically.
TWWC	***TWI Write Collision Flag*** It is set when attempting to write to the TWI Data Register (TWDR) when TWINT is low.
TWEN	***TWI Enable Bit*** It enables TWI operation and activates the TWI interface. When TWEN is written to one, the TWI takes control over the I/O pins connected to the SCL and SDA pins.
TWIE	***TWI Interrupt Enable*** When this bit is written to one, and the I-bit in SREG is set, the TWI interrupt request will be activated for as long as the TWINT flag is high.

4) TWSR (TWI Status Register)

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]
TWS7	**TWS6**	**TWS5**	**TWS4**	**TWS4**	**-**	**TWPS1**	**TWPS0**

TWS [7-3]	***TWI Status Bits*** These 5 bits reflect the status of the TWI logic and the 2-wire serial bus. Different status codes are assigned for different conditions.
TWPS [1-0]	***TWI Pre-scalar Bits*** These bits set pre-scalar value for clock frequency SCL.

TWPS[1:0]	Pre-scalar Value
00	1
01	4
10	16
11	64

Other Registers

There are 2 other registers; **TWAR** (TWI Slave Address Register) and **TWAMR** – TWI (Slave) Address Mask Register. These are used for programming ATmega328P as I2C slave. Since, we will be operating out microcontroller in master mode, so these two will not be used. Refer to the datasheet for further description.

Programming ATmega328P Two-Wire Interface (TWI or I2C) in Master Mode

In this section we will discuss the steps of programming our microcontroller ATmega328P in master mode. Here we will focus on the simplest form of TWI programming without checking the status register. In most applications, if you are not dealing with critical systems and there is not more than one master on a single bus, you can use this method. If you want to deal with multi-master or critical designs you must check the value of the status register.

To program ATmega328P in master operating mode, the steps are explained below. For each individual step, a sub-routine is created to make the complete process easier. Here we have discussed single byte read and write only. Multiple byte burst read and write operations are also supported in I2C communication.

Initialization

To initialize the TWI module to operate in master operating mode, we should do the following steps:

1) Set the TWI module clock frequency by setting the values of the TWBR register and the TWPS bits in the TWSR register.
2) Enable the TWI module by setting the TWEN bit in the TWCR register to one.

```
void i2c_init(void)
{
	TWSR=0x00; //set pre-scaler bits to zero
	TWBR=0x62; // 75 kHz for XTAL=16MHz
	TWCR=0x04; //enable the TWI module
}
```

Transmit a START Condition

START condition is sent by setting the TWEN, TWSTA, and TWINT bits of TWCR to one.

1) Setting the TWEN bit to one enables the TWI module.
2) Setting the TWSTA bit to one tells the TWI to initiate a START condition when the bus is free.
3) Setting the TWINT bit to one clears the interrupt flag to initiate operation of the TWI module to transmit the START condition.
4) Then we should poll the TWINT flag in the TWCR register to see whether the START condition transmitted completely.

```
void i2c_start(void)
{
      TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
      while ((TWCR & (1 << TWINT)) == 0);
}
```

Send Device Address

After START condition, device address is sent with a control bit. SLA + W (Slave Address + Write) or SLA + R (Slave Address + Read). To transmit SLA + R or SLA + W, a byte is sent to the slave. To write the address byte, the same steps described for sending data byte are followed.

Send Data

These steps are followed for sending a byte (data or address) to slave

1) Copy the data byte to the TWDR.
2) Set the TWEN and TWINT bits of the TWCR register to one to start sending the byte.
3) Poll the TWINT flag in the TWCR register to see whether the byte is transmitted completely.

```
void i2c_write(unsigned char data)
{
      TWDR = data ;
      TWCR  = (1<< TWINT) | (1<<TWEN);
      while ((TWCR & (1 <<TWINT)) == 0);
}
```

Receive Data

After transmitting SLA+R, the following steps are followed to receive data byte

1) Set the TWEN and TWINT bits of the TWCR register to one to start receiving a byte. To return ACK after receiving data, the TWEA bit of the TWCR register is also set to one.
2) Poll the TWINT flag in the TWCR register to see whether a byte has been received completely.
3) Copy the received byte from the TWDR to another register to save it.

```
unsigned char i2c_read(unsigned char isLast)
{
   if (isLast == 0) //if want to read more than 1 byte
      TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWEA);
   else //if want to read only one byte
      TWCR = (1<< TWINT)|(1<<TWEN);
   while ((TWCR & (1 <<TWINT)) == 0);
   return TWDR ;
}
```

Transmit a STOP Condition

To stop data transfer, we must transmit a STOP condition. This is done by setting the TWEN, TWSTO, and TWINT bits of the TWCR register to one.

```
void i2c_stop()
{
      TWCR = (1<< TWINT)|(1<<TWEN)|(1<<TWSTO);
}
```

PCF8574 I2C I/O Expander Module

PCF8574 is an IC that can be used as I2C to Parallel-Port Expander. The device features an 8-bit quasi-bidirectional I/O port (P0–P7). It has SDA and SCL for I2C inputs for interface.

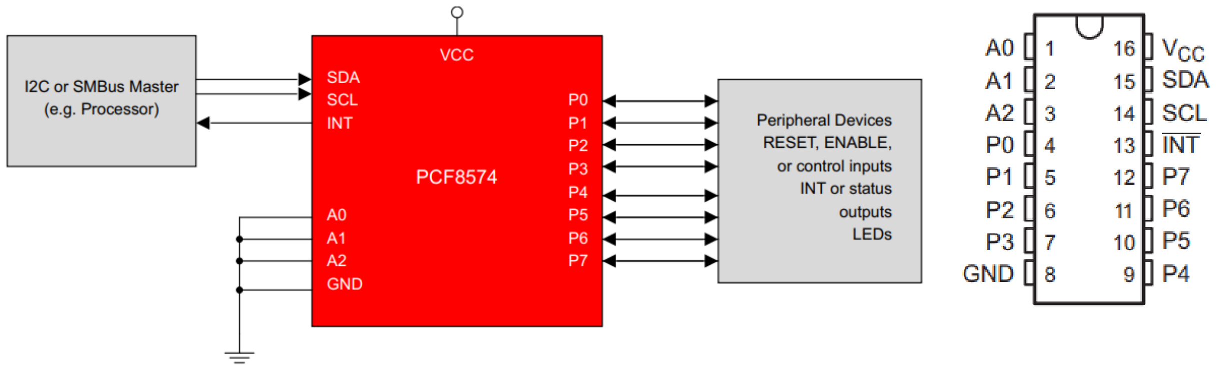


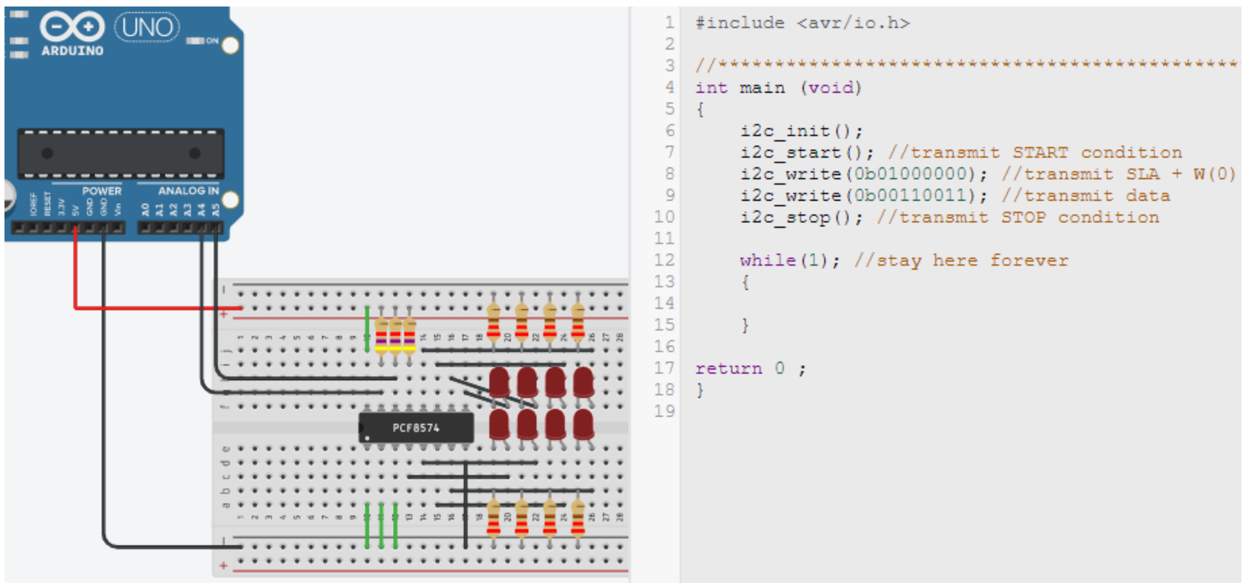
Figure 7: PCF8574 IC Pinout

Table 3: Functions of PCF8574 Pins

A0-A2	Address Pins (All connected to ground corresponds to device address 000. All connected to VCC corresponds to the device address 111)
Vcc	Voltage Supply
GND	Ground
P7-P0	P-port input/output.
INT	Interrupt output.
SDA	Serial data line. Connect to VCC through a pullup resistor
SCL	Serial clock line. Connect to VCC through a pullup resistor
The 7-bit slave address of this IC (PCF8574) is **010 0$A_0A_1A_2$** which is 0x20 for the connections shown above. This corresponds to the slave address SLA+W = 010 0000 + 0 = 0100 0000 =**0x40**.	

Example 1: Using PCF8574 IC as I2C to 8-Bit Output Expander for Controlling LEDs

To demonstrate the programming of ATmega328P two-wire interface (I2C) and required connections with PCF8574 IC, consider the example code and circuit shown in Figure 8.



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

//*************************************************
int main (void)
{
    i2c_init();
    i2c_start(); //transmit START condition
    i2c_write(0b01000000); //transmit SLA + W(0)
    i2c_write(0b00110011); //transmit data
    i2c_stop(); //transmit STOP condition

    while(1); //stay here forever
    {

    }

return 0 ;
}
```

Figure 8: Example 1 Code and Connections

The slave address with control bit is sent after calling **i2c_init( )** and **i2c_start( )** sub-routines. After sending the address byte, data byte is sent (0b00110011). This is received by the PCF8574 IC and given at output through its output port (P7 to P0). The output drives 8 LEDs. Simulate the circuit to verify output and understand connections. Note that the pull-up resistors are connected with SDA and SCL pins (as needed for I2C interface). The resistors with LEDs are simply current limiting resistors.

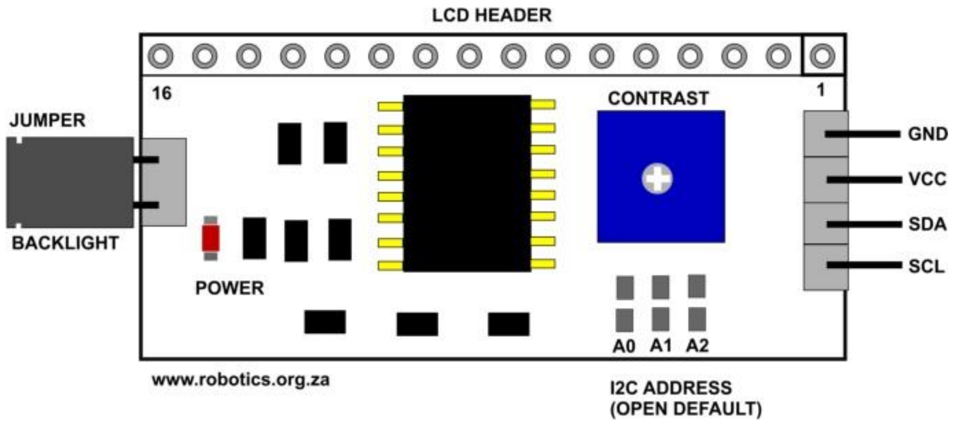


Figure 9: PCF8574 I2C Expander Module

Using PCF8574 IC, an <u>expander module for LCD connection through I2C interface</u> is available. It is also known as Serial LCD I2C Module. This expander module provides ease of connection with the 16 pins of LCD. However, **the PCF8574 IC has only 8 I/O pins**, therefore, out of the 11 (D0-D7, RS, EN and RW) LCD pins, only 8 are internally connected and can be controlled. The connections listed below. This is the reason we have to operate the **LCD in 4-bit mode** with this expander instead of the earlier utilized 8-bit mode.

Table 4: Connections of Expander Module with LCD Pins

P0	**P1**	**P2**	**P3**	**P4-P7**	**A0-A2**
RS	R/W	E	-	D4-D7	111 (Default) SLA+W = 010 0111 +0 = **0x4E**

With this module, **pull-up resistors for SDA and SCL are not needed externally**. The rest of the pins of LCD for power supply and backlight LEDs are connected with PCF8574 power pins. The contrast control pin is connected with on-board potentiometer. Some LCDs have this module already connected to them as backpack since it is compatible with HD44780.

4-bit Mode Operation of 16x2 LCD Screen

Previously we used LCD in 8-bit mode. To utilize the PCF8574 module for controlling the LCD screen through I2C interface, we first need to understand the 4-bit mode operation of 16x2 LCD and its requirements.

- **Sending data/command in 4-bit Mode**
  In 4-bit mode the data is sent in **nibbles** (nibble is group of 4 bits). First the higher nibble and then the lower nibble. To send both; command or data, higher 4-bits are separately sent and then the lower 4-bits.
  The common steps are:
  1. Mask lower 4-bits
  2. Send to the LCD port

3. Toggle enable signal (send a pulse at enable to latch the data sent)
4. Mask higher 4-bits
5. Send to LCD port
6. Toggle enable signal (send a pulse at enable to latch the data sent)

- **Resetting or Initializing the LCD:**
  To enable the 4-bit mode of LCD, follow special sequence of initialization that tells the LCD controller that user has selected 4-bit mode of operation. Following is the reset sequence of LCD.
  1. Wait for about 20msec initially after powering-up the LCD
  2. Send the first initial value (**0x30**)
  3. Wait for about **4.1msec**
  4. Send second initial value (**0x30**)
  5. Wait for about **100usec**
  6. Send third initial value (**0x30**)
  7. Wait for **100usec**
  8. Send (**0x02**) to switch to the four-bit mode. This is not Function Set instruction (0x38 for 8-bit mode or 0x28 for 4-bit mode), but it signals that a real Function Set instruction will be sent after it.
  9. Wait for 100usec
  10. Select bus width by sending command (**0x28**) for function set: 4-bit mode and 2 line display
  11. Wait for 1msec

Subroutines for LCD Control in 4-bit Mode through the I2C Interface

A few subroutines are created for initializing the LCD, sending data and commands, sending a pulse at Enable pin, and for printing message strings on LCD in 4-bit mode. These sub-routines utilize the earlier mentioned **i2c_write( )** function repeatedly, since we are interfacing the LCD through I2C expander module.

PCF8574	**P7**	**P6**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**
LCD	**D7**	**D6**	**D5**	**D4**	**-**	**E**	**R/W**	**RS**
Functions	Data or command bits (4-bits at a time)				**-**	After placing data, a pulse is sent to latch	0: Write Mode 1:Read Mode	0: Command Mode 1: Data Mode

The power pins of LCD and backlight are powered through expander module.

Subroutine for Sending Pulse at Enable

```
void toggle()
{
      i2c_write((TWDR |= 0x04)); //E=1, E is at bit 2 (P2)
      _delay_us(1); //Enable pulse for short time
      i2c_write((TWDR &= ~0x04)); // E=0
      _delay_us(100); //make pulse longer wait for at least 100usec
}
```

Subroutine for Sending Command

```
void lcd_cmd(char v2)
{
  i2c_write((TWDR&=~0x03)); //RW =0 for Read, RS=0 for Command Mode

  i2c_write((TWDR &= 0x0F)); // clear the 4bits (D7-D4)before
                             //sending any new data or command

  i2c_write((TWDR |= (v2 & 0xF0))); // place command at TWDR and mask
                      //the lower 4bits, to send higher nibble
  toggle();            // send pulse at E

  i2c_write((TWDR &= 0x0F));    // clear the 4 bits (D7-D4)

  i2c_write((TWDR |= ((v2 & 0x0F)<<4)));//command’s lower nibble sent

  toggle(); //send pulse at E
}
```

Subroutine for Data Write

```
void lcd_dwr(char v3)
{
    i2c_write((TWDR|=0x01)); //RS=1 for Data Mode
    i2c_write((TWDR &= 0x0F)); //Clear data pins (D7-D4)
    i2c_write((TWDR |= (v3 & 0xF0))); //mask lower nibble & send data
    toggle();      //Pulse at E

    i2c_write((TWDR &= 0x0F)); //clear data pins (D7-D4)
    i2c_write((TWDR |= ((v3 & 0x0F)<<4))); //mask higher nibble and
                                         //send lower nibble
    toggle();
}
```

Subroutine for LCD Initialization

```
void lcd_init()
{
  _delay_ms(100);       //wait before sending initialization
  lcd_cmd(0x30);       //-----Sequence for initializing LCD
  _delay_ms(10);
  lcd_cmd(0x30);
  _delay_ms(5);
  lcd_cmd(0x30);       //    "                "
  _delay_ms(5);
  lcd_cmd(0x02);
  _delay_ms(1);
  lcd_cmd(0x28);          //-----Selecting 16 x 2 LCD in 4Bit mode
  _delay_ms(1);
  lcd_cmd(0x0C);          //-----Display ON Cursor OFF
  lcd_cmd(0x06);          //-----Cursor Auto Increment
  lcd_cmd(0x01);          //-----Clear display
  _delay_ms(4);       //2msec delay required after initialization
}
```

Subroutine for Printing Strings / Messages

```
void lcd_msg(char *c)
{
     while(*c != 0) //----Wait till all String char are passed
     lcd_dwr(*c++); //--Send the char to LCD & inc c for next char
}
```

LAB TASKS

TASK 1: To interface LCD in 4-Bit Mode with ATmega328P through I2C Expander

1) Program Atmega328P for controlling 16x2 LCD screen in 4-bit mode using I2C interface. A sample program is given at the end of this lab. This code uses the functions given in previous sections.
2) Make appropriate hardware connections of ATmega328P (TWI) with PCF8574 module. Connect the module with 16x2 LCD screen.
3) Test your setup and observe output at LCD.
4) **Modify** the code to make the display blink with a certain delay. You can refer to the Lab 05 manual for LCD related commands.

TASK 2: To compare communication through UART, SPI and I2C

You have studied and practiced communication of ATmega328P through different synchronous and asynchronous methods. Compare UART, SPI, and I2C. List differences.

Sample Code for Task 1

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>

//Including I2C functions
void i2c_init()
{
        TWBR = 0x62;            //Setting bit rate
        TWCR = (1<<TWEN);       //Enable I2C
        TWSR = 0x00;            //Prescaler set to 1
}


//Start condition
void i2c_start()
{
TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
        //start condition
while (!(TWCR & (1<<TWINT)));
        //check for start condition
}
```

```

void lcd_init()
{
   _delay_ms(100);     //wait initialization
  lcd_cmd(0x30);     //-----Sequence for initializing LCD
   _delay_ms(10);
  lcd_cmd(0x30);
   _delay_ms(5);
  lcd_cmd(0x30);
  _delay_ms(5);
   lcd_cmd(0x02);
   _delay_ms(1);
  lcd_cmd(0x28);
   _delay_ms(1);
lcd_cmd(0x0C);        //-----Display ON Cursor OFF
lcd_cmd(0x06);        //-----Cursor Auto Increment
lcd_cmd(0x01);        //-----Clear display
_delay_ms(4);        //2msec delay

}
```

```
//I2C write (for sending address and data)
void i2c_write(char x)
{
            TWDR = x;  //Move value to I2C
            TWCR = (1<<TWINT) | (1<<TWEN);
           //Enable I2C and clear interrupt
            while  (!(TWCR &(1<<TWINT)));
}

//-----LCD 4-bit Mode functions using I2C-------//


void toggle()
{
            i2c_write((TWDR |= 0x04));
            _delay_us(1); //Enable pulse
            i2c_write((TWDR &= ~0x04));
            _delay_us(100); //make pulse longer or wait
//for at least 100usec after sending each command
}

void lcd_cmd(char v2)
{
            i2c_write((TWDR&=~0x03));
   i2c_write((TWDR &= 0x0F));
            i2c_write((TWDR |= (v2 & 0xF0)));
            toggle();

            i2c_write((TWDR &= 0x0F));
            i2c_write((TWDR |= ((v2 & 0x0F)<<4)));
            toggle();
}

void lcd_dwr(char v3)
{
            i2c_write((TWDR|=0x01));
            i2c_write((TWDR &= 0x0F));
   i2c_write((TWDR |= (v3 & 0xF0)));
            toggle();

            i2c_write((TWDR &= 0x0F));
            i2c_write((TWDR |= ((v3 & 0x0F)<<4)));
            toggle();

}
```

```
void lcd_msg(char *c)
{
          while(*c != 0)  //Wait till all String are passed
          lcd_dwr(*c++);    //----Send the String to LCD
}

int main (void)

{

  i2c_init();    //initialize i2c
  i2c_start();    // start i2c
  i2c_write(0x4E);
//01001110=4E
                   //(Device Address)
  lcd_init();  //initialize LCD

  float Percentage=75.5; //Message %
  char buffer_str[10];     //buffer
  dtostrf(Percentage,5,1, buffer_str);

     // converting float to string

  lcd_cmd(0x80); //Cursor start of Line 1
  lcd_msg("Expected ES Lab"); //Message
  lcd_cmd(0xC0);  //Move cursor to Line 2
  lcd_msg("Result ");
  lcd_msg(buffer_str);
  lcd_msg("% :)");

  while(1)
  {}
}
```