MICROPROCESSOR
LAB MANUAL

Contents

CHAPER 1 INTRODUCTION TO THE EXPERIMENT KIT

1.1 INTRODUCTION

The microprocessor experiment kit is designed based on the AVR microcontroller family, intended to provide users with a quicker understanding of fundamental AVR microcontroller concepts. The experimentation manual comprises instructions for using the kit, practical experiments, and some source code for reference.

The instructional material will introduce the components of the experimentation kit, organized into sections as follows:

- **Basic Theory:** This section will briefly summarize the theoretical knowledge relevant to the experiments.
- **Hardware Design:** The content of this section will help users understand the details of the circuit diagram and how the hardware components of the experimentation kit are designed. Users need to have a clear understanding of the content covered in this section.
- **Hardware Connection and Programming:** This section will assist users in learning techniques for connecting signals and developing software to meet the experiment's requirements.

  Each experiment is organized into the following sections:

- **Objective:** This section helps learners understand the specific goals of the experiment.
- **Requirements**: This part outlines the specific requirements of the experiment.
- **Instructions**: This section provides some guidance to make programming easier for the users.
- **Assessment**: This helps users evaluate the extent to which they have achieved the experiment's objectives and suggests some adjustments to enrich the experiment's content.

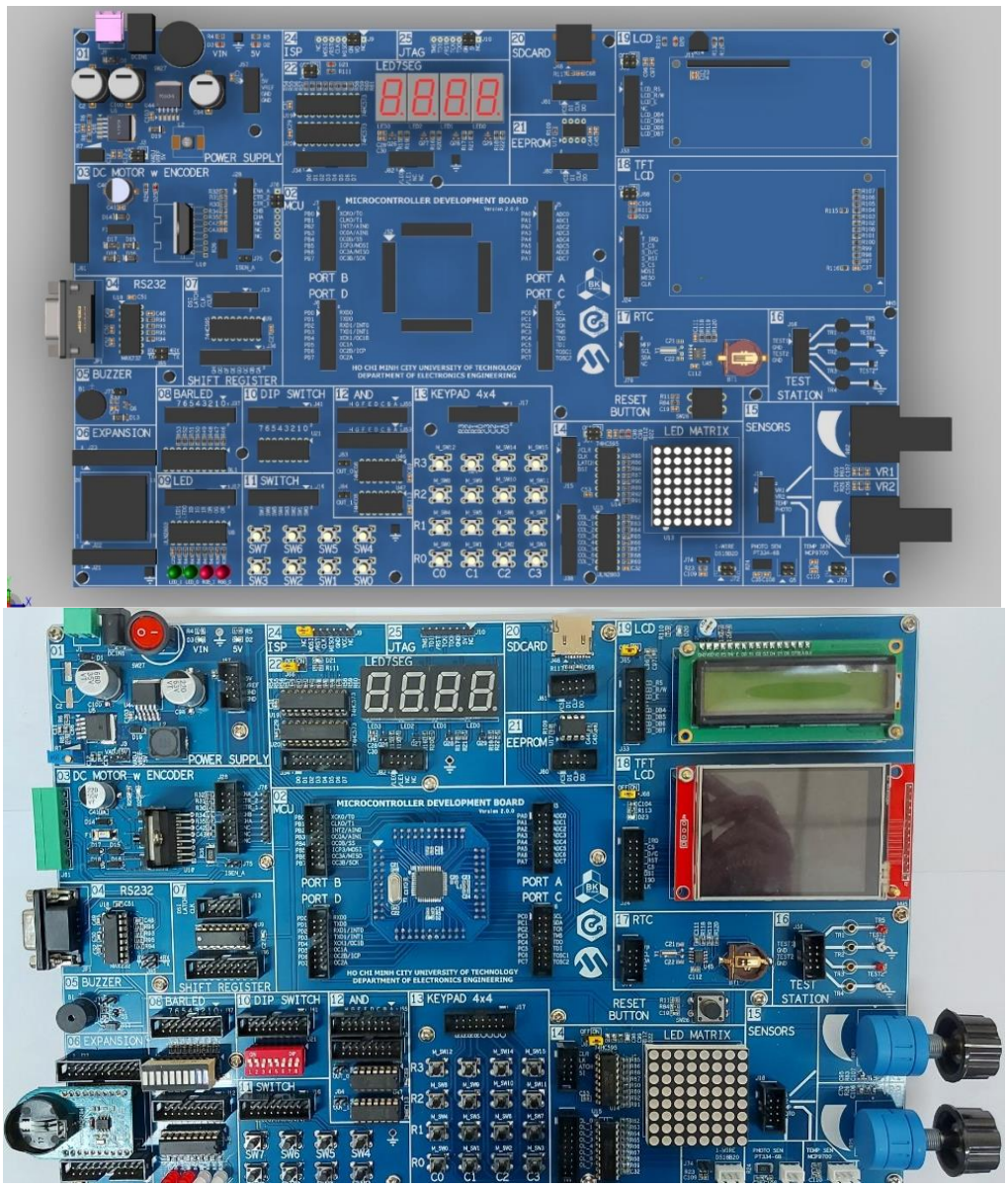
Note: Students should read the lab material and the manual and prepare the program at home in advance. Before going to the lab, students must show the lab instructor their preparation.

If any errors or questions arise, students can directly inform the instructor or send an email to ***buiquocbao@hcmut.edu.vn***

1.2 OVERVIEW OF THE EXPERIMENTATION KIT

The experimentation kit has form and basic components as shown in Figure 1. It comes with a set of connecting wires, various sensors, a DC motor, and a power adapter.

Figure 1: Overview of the Microprocessor Experimentation Kit



1.2.1 Signal Connections on the Microprocessor Experimentation Kit

The microprocessor experimentation kit is designed to be used flexibly. Around the MCU block, there are 4 headers used to connect signals from PORT A, B, C, D to external

peripherals. Students can use 8x2 cables or pin header wires to establish these signal connections.

1.2.2 Testpoint Blocks

To facilitate the use of a VOM (Volt-Ohm-Milliammeter) and oscilloscope for signal measurements, the experimentation kit provides testpoints. Students can use single wires to connect from the signals they want to measure to these terminals and connect probe wires from the oscilloscope to the testpoints for conducting measurements.

Please note that testpoints TR2, TR6, TR4, and TR8 are already connected to GND (Ground). When connecting, be careful to avoid mistakenly connecting signals to GND.

Figure 2: Connecting and Measuring Signals

1.3 USING THE EXPERIMENTATION KIT

1.3.1 Configuring the Experimentation Kit and Connecting to a Computer

- Connect the PICKIT 4 module to the JTAG port on the experimentation kit. If you want to use pins PC2, PC3, PC4, PC5, connect to the ISP port. Note that pin 1 of the PICKIT should be plugged into pin 1 of the JTAG/ISP port.
- If connected to the SPI port, you can program but cannot debug the program.
- Connect the PICKIT 4 to the computer via a USB port.
- Connect the necessary signals from the MCU port pins to the required peripherals.
- Power up the experimentation kit.
- Begin the experimentation process.

MPLAB PICkit 4			DEBUG	
Connector	Pin #	Pin Name	AVR® JTAG	AVR ISP(&DW)
	1	TVPP		
	2	TVDD	VTG	VTG
	3	GND	GND	GND
	4	PGD	TDO	MISO
	5	PGC	TCK	SCK
	6	TAUX	RESET	RESET
	7	TTDI	TDI	MOSI
	8	TTMS	TMS	

Figure 1 Connecting the PICKIT 4 to the Experimentation Kit

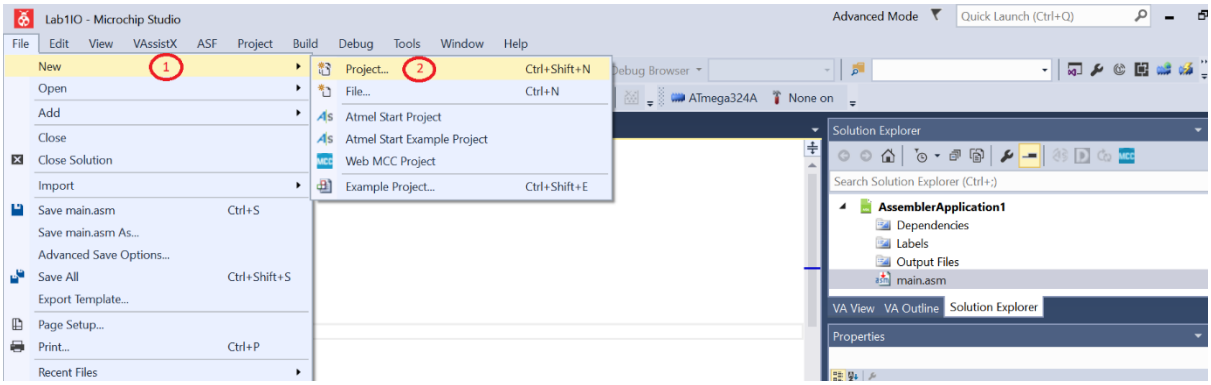
1.4 PROGRAMMING WITH MICROCHIP STUDIO

Microprocessor or microcontroller systems all require firmware (programs) to control their operations. This firmware is stored in the program memory of the MCU. At the lowest level, the program in the system consists of binary bits, often referred to as machine code.

However, programming with binary bits can be very challenging. In practice, programs are written on a computer in assembly language or other high-level languages like C/C++, Basic, etc. These programs need to go through compilation and linking processes to be converted into machine code suitable for the MCU in use. The tools to perform these steps are called assemblers, compilers, and linkers. Each type of MCU typically has its own assembler program.

In this lab, programmers can use Microchip Studio for AVR and SAM Devices, which can be downloaded for free from the Microchip website.Các bước để tạo một Project:

1. Select File -> New -> Project.



2. Create a Project:

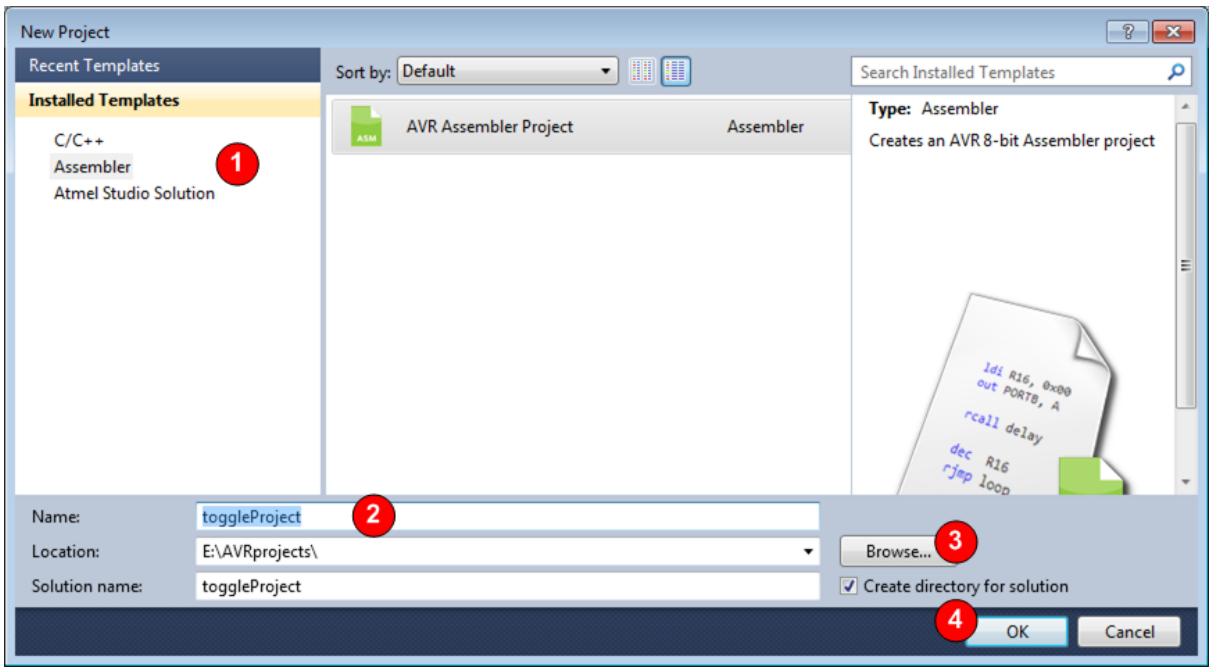
If creating an Assembly Project:

a. Choose AVR Assembler Project.

b. Name your project.

c. Select a folder to store the project.

d. Click OK.



If creating a C Project:

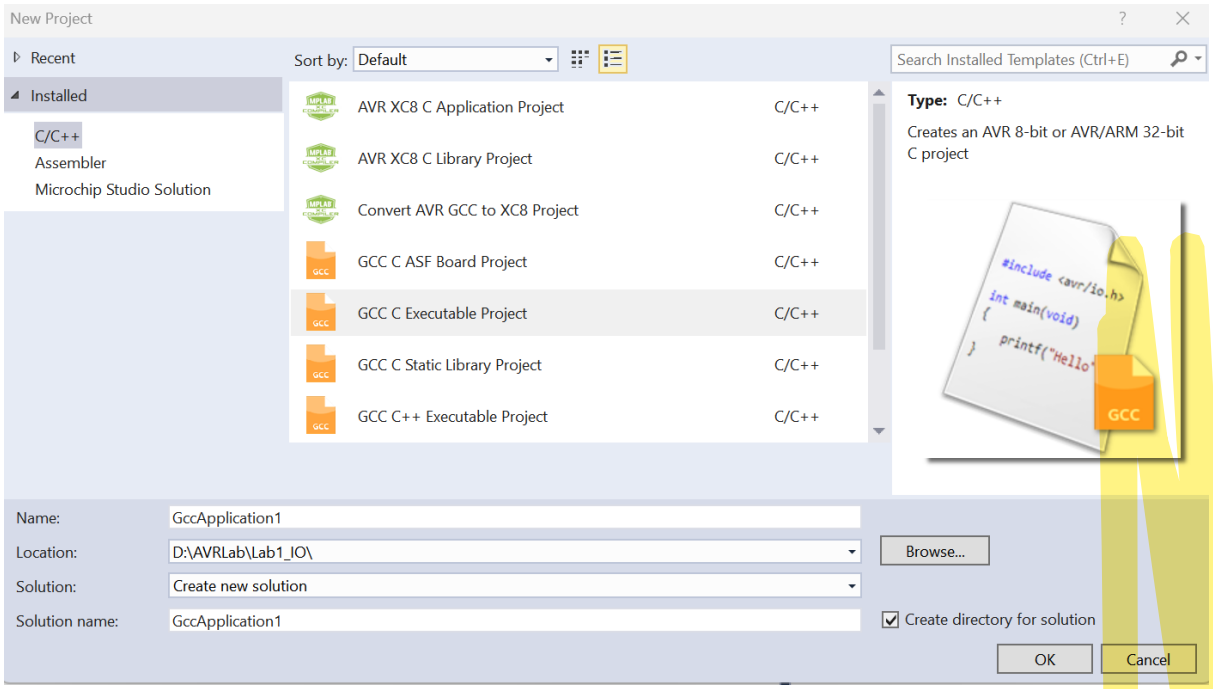
a. Choose C/C++.

b. Select GCC C Executable Project.

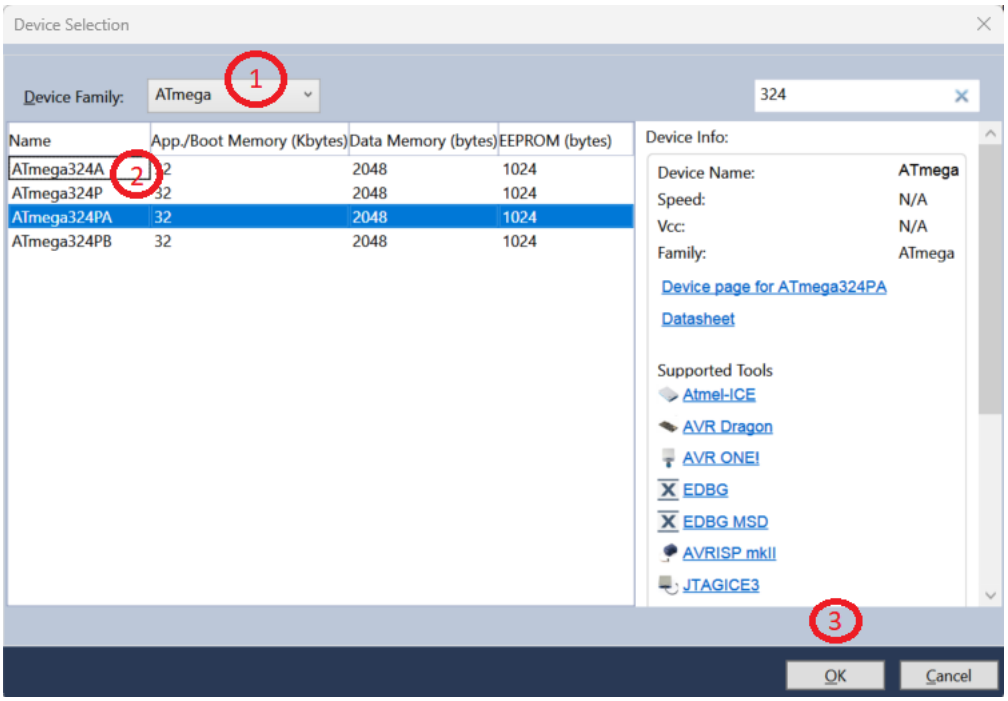
c. Name your project.

d. Choose a folder to store the project.

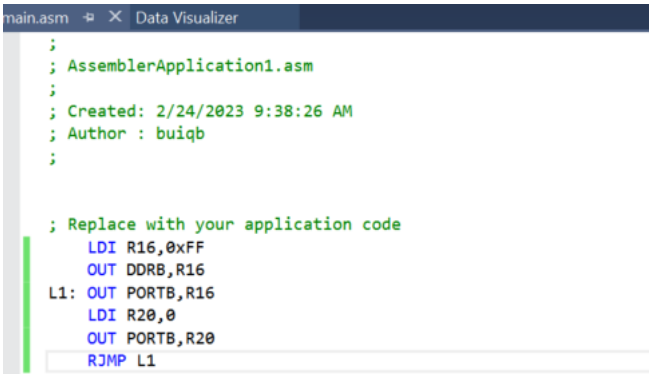
e. Click OK.



3. In the Device Selection section:
   a. Choose Atmega for the Device Family.
   b. Select Atmega324PA.
   c. Click OK.

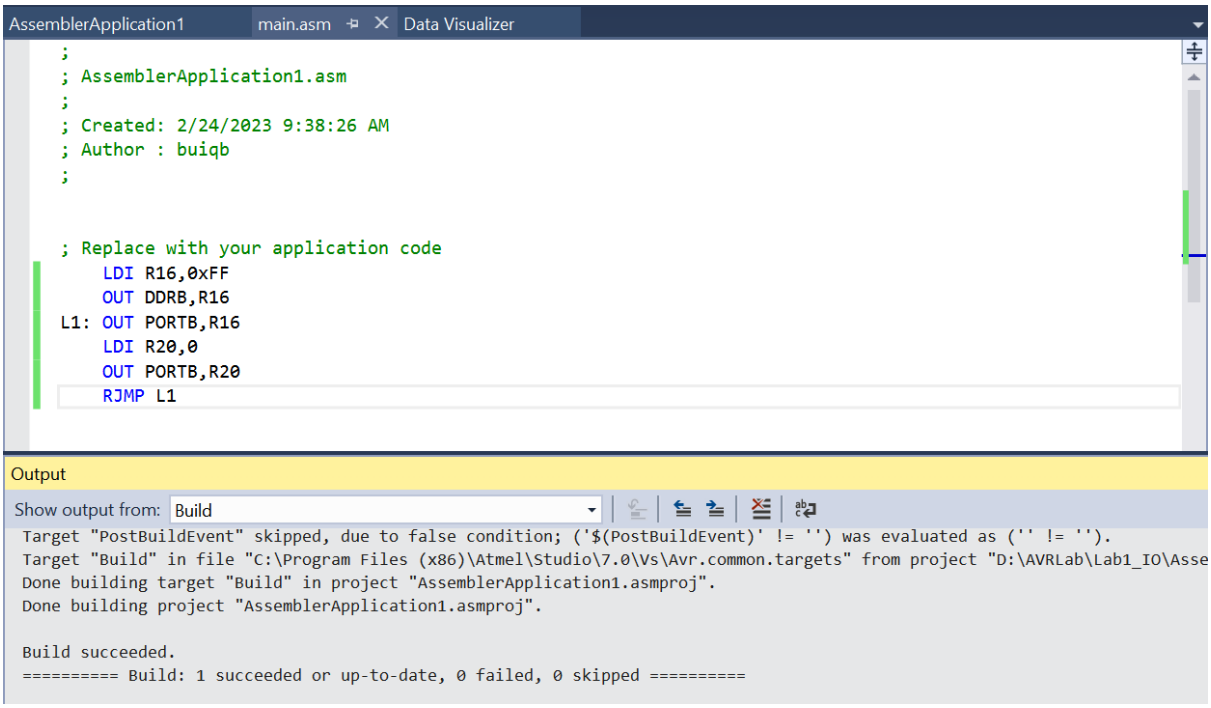


4. The compiler automatically generates a project with the main.asm file. Students can edit this program file or copy their program code into it. After editing, compile it to upload it to the microcontroller.



5. Compile the program:

Press F7 or select Build -> Build Solution. The compilation result will be displayed in the Output window. If the compilation is successful, you will have a .hex file and other output files in the Debug or Release folder.



6. Programming the AVR MCU:

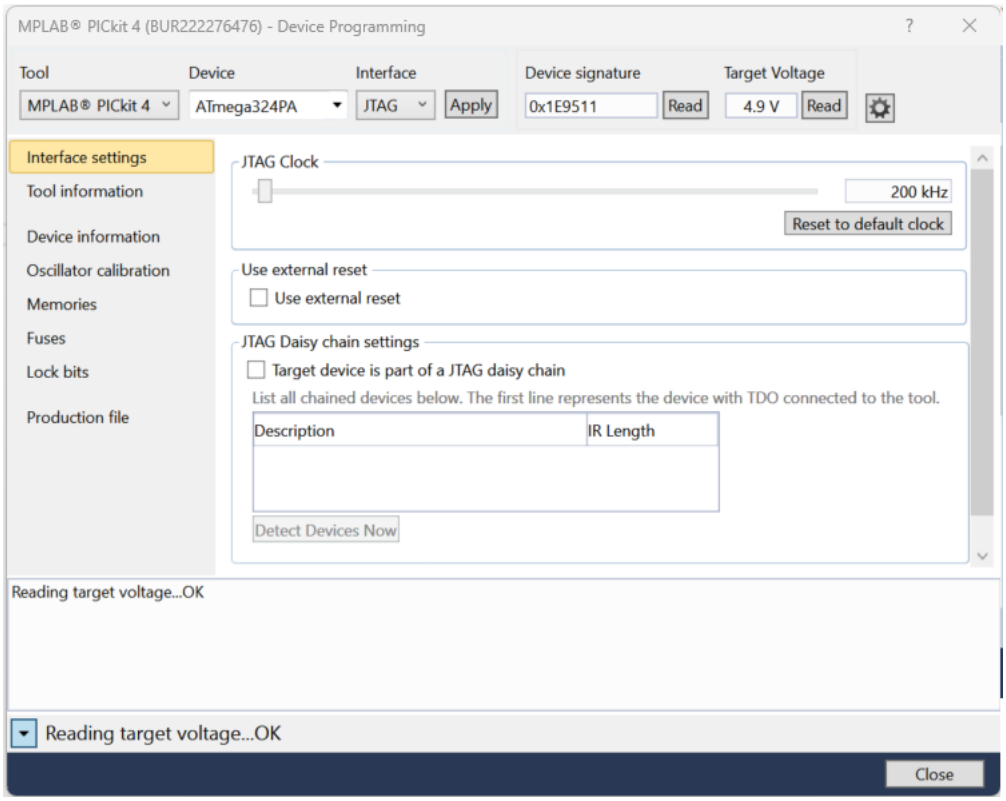
You can program the microcontroller when the PICKIT is connected to the experimentation kit via ISP or JTAG interface.

a. Select Tool -> Device Programming.

b. In the Tool section, choose MPLAB PICKIT 4, and select the JTAG or ISP interface in the Interface section (depending on your current connection). Click Apply.

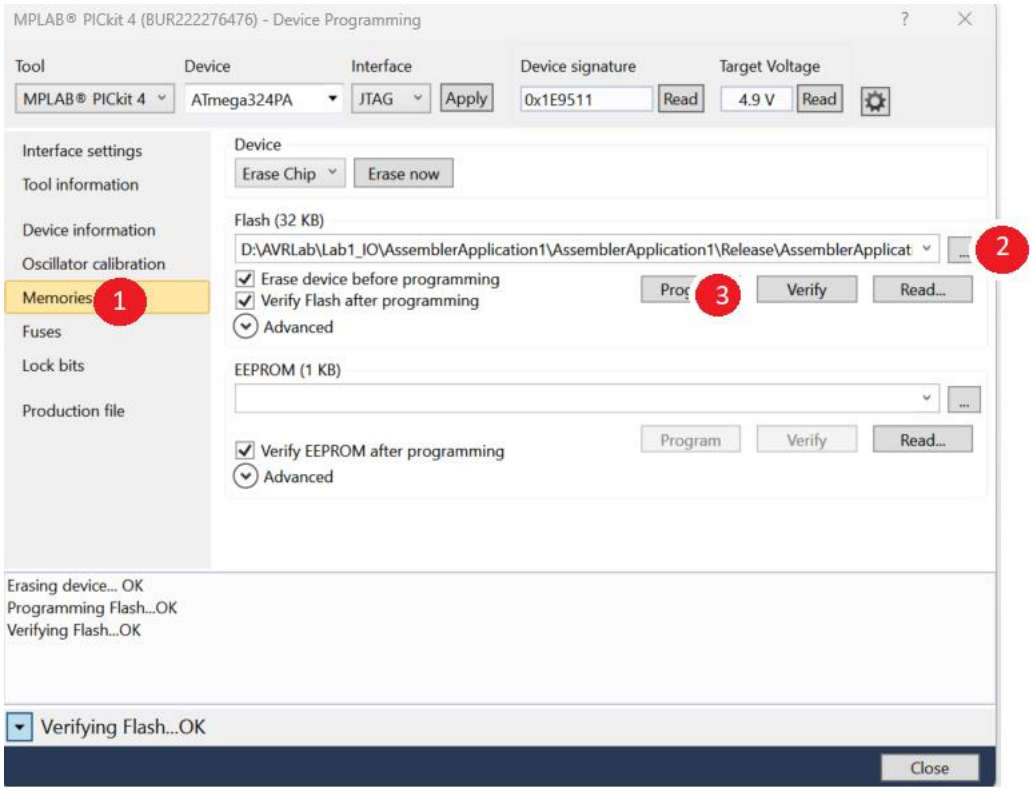
c. In the Device Signature section, click Read.

d. In the Target Voltage section, click Read.



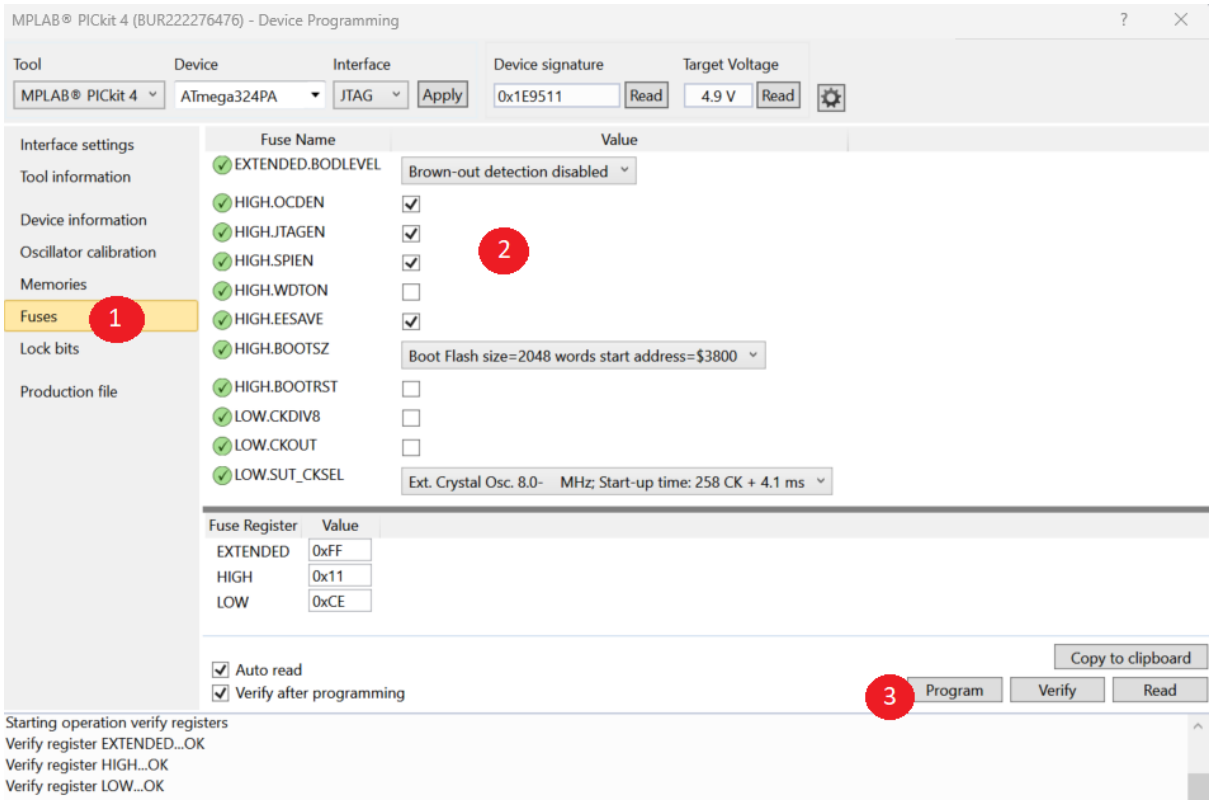
7. Programming the Chip:

Select Memories. In the Flash section, choose the compiled Hex file. Click Program to upload the program to the microcontroller.



8. Programming the Fuses:

If you want to change the operational configurations of the microcontroller, you need to program the fuses. Click on Fuses, select the fuses you want to program, and then click Program.



9. Meaning of Fuse Bits:

When a fuse bit is assigned a **value of 0**, it means the fuse bit is **programmed**, while a value of **1** means it is **unprogrammed**. Fuse bits are used to configure various settings and features of the microcontroller and setting them to specific values can alter the behavior of the microcontroller.

High Fuse Byte	Bit No.	Description	Default Value
OCDEN(1)	7	Enable OCD	1 (unprogrammed, OCD disabled)
JTAGEN	6	Enable JTAG	0 (programmed, JTAG enabled)
SPIEN(2)	5	Enable Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
WDTON(3)	4	Watchdog Timer Always On	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed), EEPROM not reserved
BOOTSZ1(4)	2	Select Boot Size	0 (programmed)
BOOTSZ0(4)	1	Select Boot Size	0 (programmed)
BOOTRST	0	Boot Reset vector Enabled	1 (unprogrammed)

- **Bit 7 (OCDEN):** Activates the On-Chip Debugger (OCD) function. By default, OCD is disabled. To use JTAG for debugging, this bit must be programmed.

- **Bit 6 (JTAGEN):** When this bit is set to 0, it allows the use of JTAG pins for debugging or programming, and these JTAG pins are not used as regular I/O ports. By default, JTAG is enabled. To use JTAG for debugging, OCDEN must be programmed.
- **Bit 5 (SPIEN):** Enables programming via ISP using SPI.
- **Bit 4 (WDTON):** Enables the watchdog timer.
- **Bit 3 (EESAVE):** Determines whether to retain (save) EEPROM data during chip reprogramming. If EESAVE is set to 0, all data in the EEPROM will not be erased during chip programming, or vice versa.
- **Bit 2 (BOOTSZ1) and Bit 1 (BOOTSZ0):** These two bits select the bootloader section size.

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Application Flash Section	Boot Loader Flash Section	End Application Section	Boot Reset Address (Start Boot Loader Section)
1	1	256 words	4	0x0000 - 0x3EFF	0x3F00 - 0x3FFF	0x3EFF	0x3F00
1	0	512 words	8	0x0000 - 0x3DFF	0x3E00 - 0x3FFF	0x3DFF	0x3E00
0	1	1024 words	16	0x0000 - 0x3BFF	0x3C00 - 0x3FFF	0x3BFF	0x3C00
0	0	2048 words	32	0x0000 - 0x37FF	0x3800 - 0x3FFF	0x37FF	0x3800

- **Bit 0 (BOOTRST):** Allows running the bootloader program in the bootloader memory section. When BOOTRST is set to 1, the program execution starts from address 0x0000, whereas when BOOTRST is set to 0, the memory location at the beginning of the bootloader section is the starting point, meaning the bootloader program is executed.

Low Fuse Byte	Bit No.	Description	Default Value
CKDIV8[4]	7	Divide clock by 8	0 (programmed)
CKOUT[3]	6	Clock output	1 (unprogrammed)
SUT1	5	Select start-up time	1 (unprogrammed)[1]
SUT0	4	Select start-up time	0 (programmed)[1]
CKSEL3	3	Select Clock source	0 (programmed)[2]
CKSEL2	2	Select Clock source	0 (programmed)[2]
CKSEL1	1	Select Clock source	1 (unprogrammed)[2]
CKSEL0	0	Select Clock source	0 (programmed)[2]

- **Bit 7 (CKDIV8)**: Divides the oscillator frequency by 8.
- **Bit 3 (CKSEL3,2,1,0):** Selects the clock source:

Device Clocking Option	CKSEL[3:0]
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved	0001

Table 1: Fuses configuration for AVR

When CKSEL[3:0] = 0010, the on-chip RC oscillator is selected as the clock source, and the oscillator frequency ranges from 7.3 to 8MHz, depending on the chip's temperature. If CKDIV bit is set to 0, the selected oscillator frequency is 8MHz/8 = 1MHz. By default, the microcontroller chooses the internal oscillator with a frequency of 1MHz.

You can choose one of these values, CKSEL[3:0] = 0111-0110, to select an external crystal oscillator as the clock source with a frequency of up to 20MHz.

Extended Fuse Byte	Bit No.	Description	Default Value
–	5	–	0
–	4	–	0
-	3	-	0 (0: Disabled)
BODLEVEL2[1]	2	Brown-out Detector trigger level	1 (unprogrammed)
BODLEVEL1[1]	1	Brown-out Detector trigger level	1 (unprogrammed)
BODLEVEL0[1]	0	Brown-out Detector trigger level	1 (unprogrammed)

When the power supply voltage for the microcontroller drops below a certain threshold level, the microcontroller will initiate a self-reset.

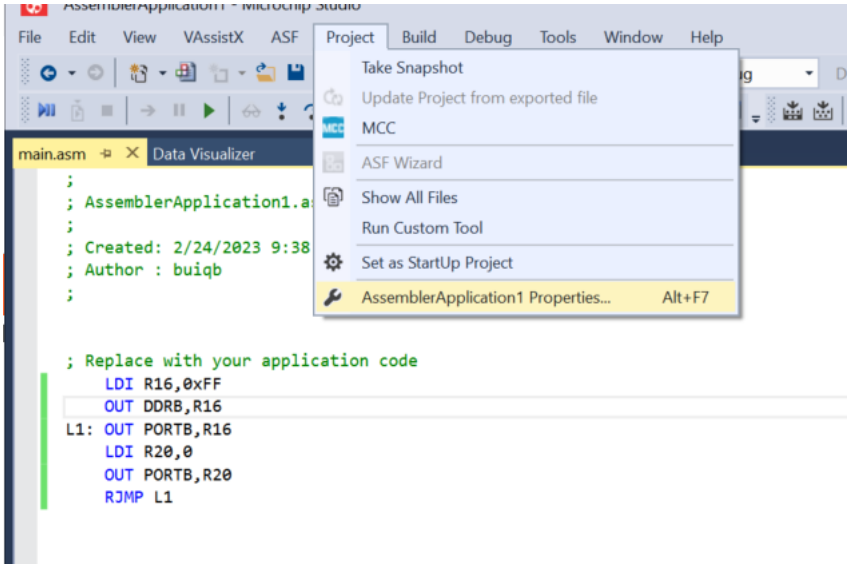
BODLEVEL [2:0] Fuses	Min. V_{BOT}	Typ V_{BOT}	Max V_{BOT}	Units
111	BOD Disabled			
110	1.7	1.8	2.0	V
101	2.5	2.7	2.9	
100	4.1	4.3	4.5	
011	Reserved			
010				
001				
000				

Note: The experimentation kit has been preconfigured with fuse settings that allow debugging of the program using the JTAG port, with an external 8MHz crystal oscillator, without division by 8. Therefore, the CPU clock frequency is 8MHz, which corresponds to 1 cycle being equal to 0.125µs.

10. Debugging the MCU:

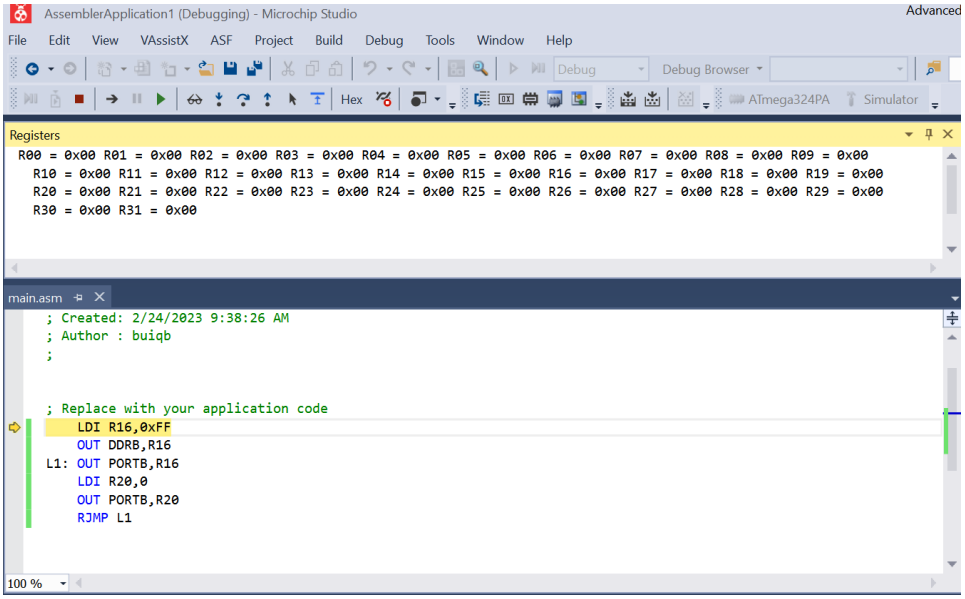
You can debug the program by either simulating it or running it directly on the microcontroller using the PICKIT 4 debugger. In this case, you should connect the PICKIT 4 to the JTAG port.

a. Select Project -> Property.

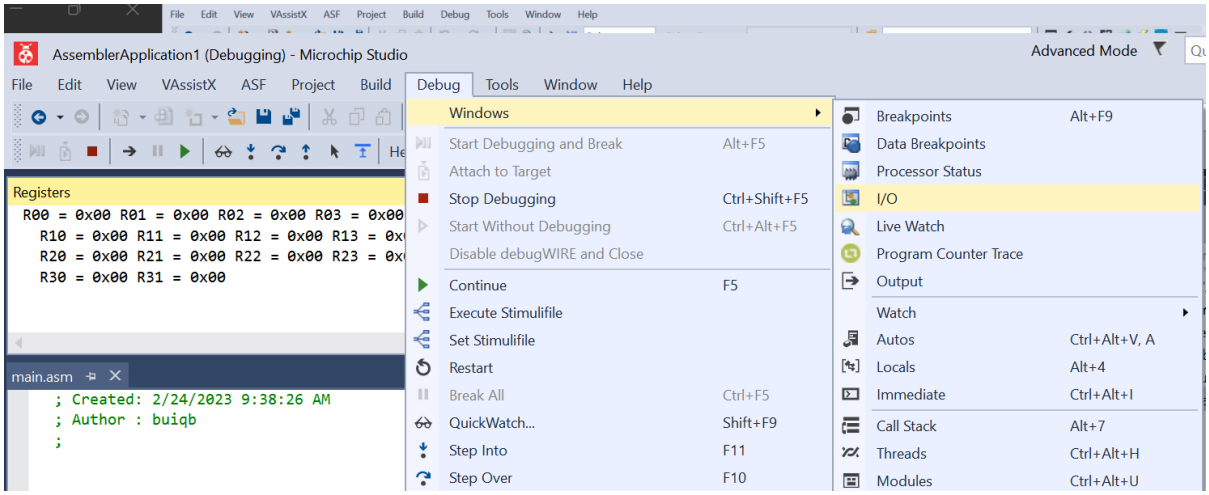


a. Choose Tool. Under the "Select Debugger/Programmer", select Simulator if you want to simulate or select MPLAB PICKIT4 if you want to debug directly on the chip. Press Ctrl-S to save the configuration.

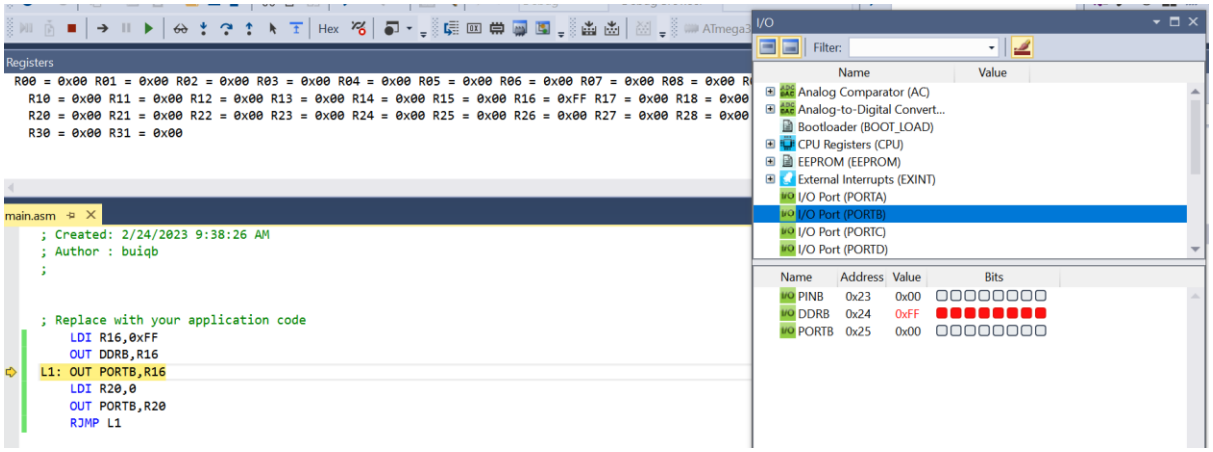
b. Select Debug -> Start Debugging and Break. The program will pause at the first line of code with a yellow highlight and is ready for debugging.



c. To display the status windows for peripherals, select Debug -> Windows and choose the desired windows.



d. Select the I/O window and choose PORT B to observe the values of the registers related to PORT B. Press F10 or select Debug -> Step Over to watch the program execution process and the changes in the values of the PORTB registers



f. To run the program step by step, press F10 or click on the Step Over icon.

Step into vs Step Over
Both the Step Over (F10) and Step Into (F11) functions execute one instruction and proceed to the next. However, when the executing instruction is a function call, Step Into will go to the first instruction of the called function, whereas Step Over will execute the entire called function and then proceed to the next instruction after the Call instruction.

Step Out
If you are currently executing a subroutine (function), you can complete the execution of that subroutine and return to the calling function using the "Step Out" command. This command will allow you to finish executing the current subroutine and then continue with the calling function. It's a useful way to quickly return to the main program or higher-level function after you've stepped into a subroutine for debugging purposes.

Run to cursor
You can right-click on a line of code and then select "Run To Cursor." The program will run until it reaches the line of code indicated by the mouse cursor. This is a convenient way to quickly execute code up to a specific point in your program during debugging.

Break Point
You can make the program run and stop at a predefined location by setting a breakpoint at the desired location. Breakpoints are markers in your code where the debugger will pause the program's execution when reached. This allows you to inspect variables, step through code, and debug effectively at specific points in your program.

The details on how to use the Simulator and Debugger for AVR microcontrollers can be found in the document " Getting-Started-with-Microchip-Studio-DS50002712B.pdf", section 1.12: AVR MCU Simulator Debugging

1.5 Using ISP and JTAG Programming Interfaces

1.5.1 ISP Programming Interface:

Header J9 allows connection to the PICKIT interface for programming the MCU via the SPI interface. To program via this interface, the SPIEN fuse bit must be programmed.

After downloading the program to the kit, the SPI signals on the AVR can be used as usual. Please note that debugging cannot be done through the ISP interface.

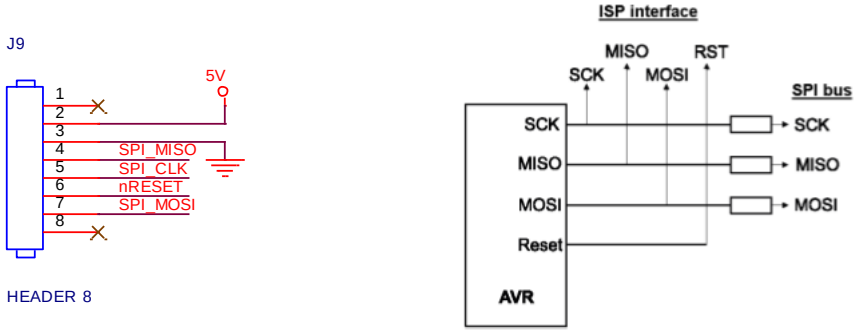


Figure 4: Use SPI pins for programming

1.5.2 JTAG debugging interface:

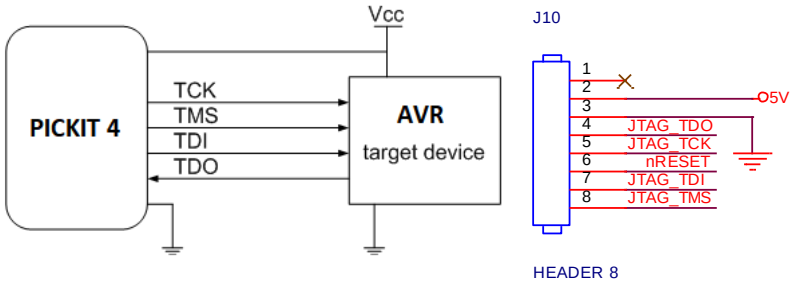


Figure 5: Giao diện JTAG Interface

To debug a program, you need to use the JTAG interface, which includes signals as shown in Figure 5. On the experimentation kit, header J10 is used to connect the PICKIT 4 debugger. This allows you to perform debugging tasks using the JTAG interface.

MCU	JTAG Signal
PC5	TDI
PC4	TDO
PC3	TMS
PC2	TCK

As can be seen, when the JTAG interface is enabled, pins PC5..PC2 cannot be used. To use these pins, you need to disable the JTAGEN fuse bit. When you do this, you can only use the ISP programming interface via the SPI pins. This configuration allows you to use those pins for general I/O purposes.

CHAPER 2 PUSH BUTTONS, SWITCHES AND LEDs

2.1 BASIC THEORY

In practice, LEDs and push buttons are two simple yet highly effective means of user interaction. Users can utilize push buttons to interact with control systems and send commands to them. Conversely, LEDs can be used to display the internal state of a system. LEDs are particularly useful for assisting programmers in debugging their programs. For instance, programmers can adjust the LED blinking speed to indicate various errors or states within the program.

2.2 HARDWARE DESIGN

2.2.1 Push Button Block

A switch (also known as a push button) is a mechanical component used to connect two or more terminals when acted upon. Typically, a switch has two terminals (a single-pole, single-throw switch).

To interface with a single push button, there are two common methods:

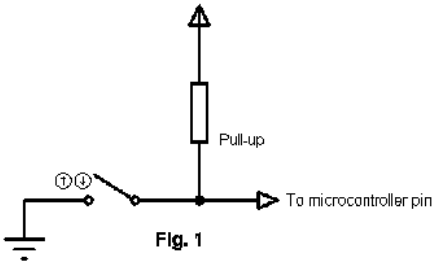


Fig. 1

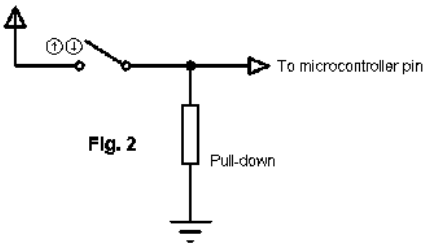


Fig. 2

Figure 2: Push Button Interfacing

In the configuration above, a resistor is connected between the VCC (power supply) and one terminal of the switch, and this terminal is also connected to the microcontroller's pin, with the

other terminal of the switch connected to GND (ground). When the switch is open (not pressed), the output signal will be HIGH (logic 1). When the switch is closed (pressed), the output signal will be LOW (logic 0). This connection method utilizes a pull-up resistor.

The other method of interfacing, the pull-down resistor configuration, is exactly the opposite, using a pull-down resistor instead of a pull-up resistor.

On the experimentation kit, there are 8 single push buttons connected to header J16. To interact with these push buttons, you need to connect the signals from J16 to the corresponding port pins on the microcontroller, using single wires or a bus of wires to connect the entire port. It's important to note that in this design, there are no external pull-up resistors provided.

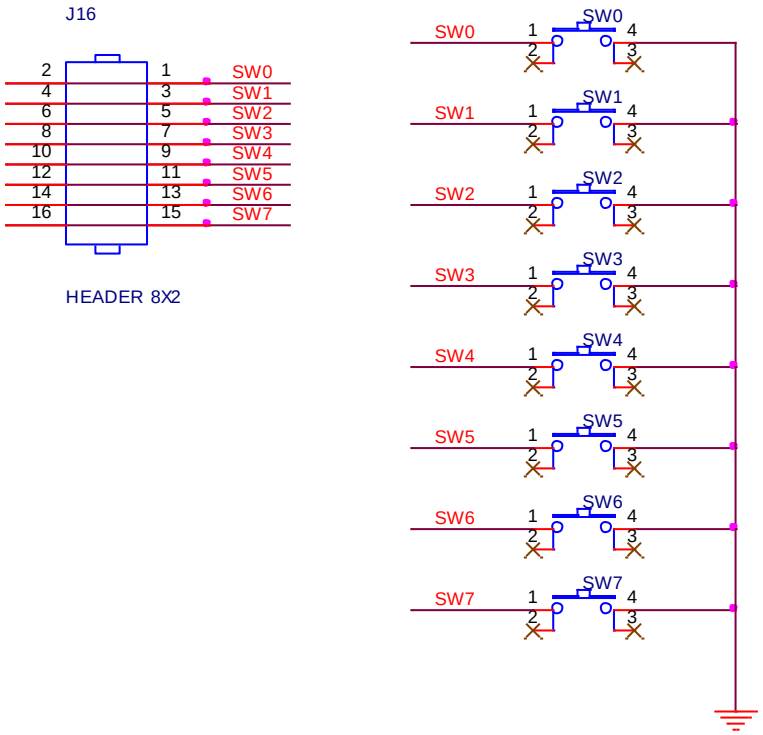


Figure 3: Single push button connection schematic

2.2.2 Dip Switch Block

On the kit, there is a DIP switch consisting of 8 switches, connected to header J41. To interact with these DIP switches, you need to connect the signals from J41 to the corresponding port pins on the microcontroller, using single wires or a bus of wires to connect the entire port. It's important to note that in this design, there are no external pull-up resistors provided.

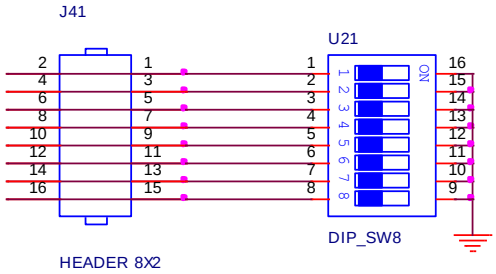
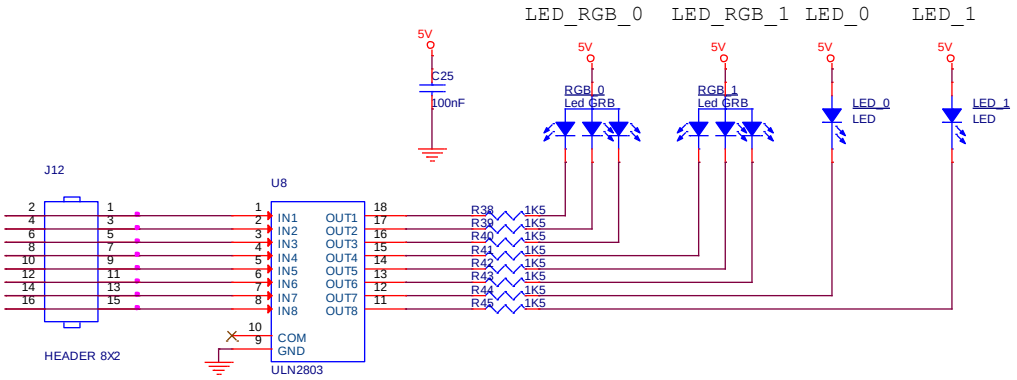


Figure 4: DIP Switch Connection shematic

2.2.3 LED block

The experimentation kit supports 2 RGB LEDs and 2 individual LEDs, connected to J12 and driven by the ULN2803 IC. To turn on one LED, the corresponding signal on J12 should be set to 1.



Hình 5 Sơ đồ kết nối khối LED đơn

2.2.4 BARLED Block

A LED-Bar consists of 8 individual LEDs connected together and can be connected to J37. To illuminate a specific LED within the 1-bar LED, the corresponding signal on J37 needs to be set to logic 1.

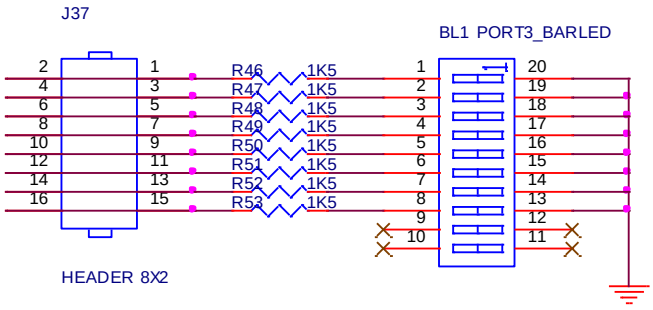


Figure 6: LED-bar connection schematic

2.2.5 Shift Register Block

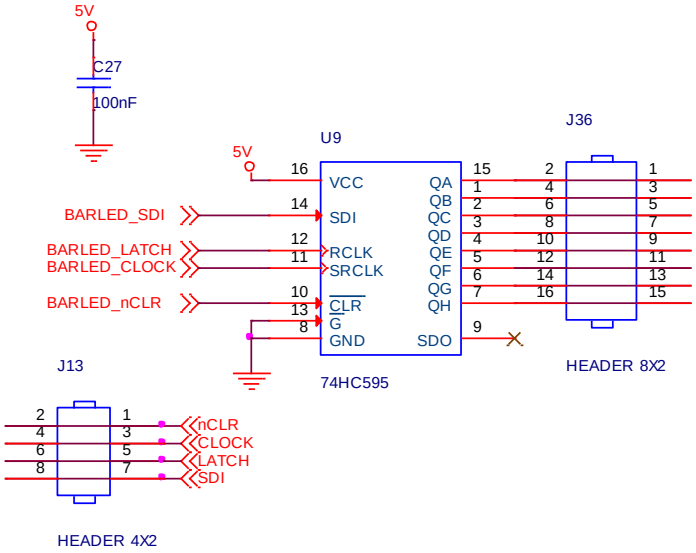


Figure 7: Shift Register Schematic Diagram

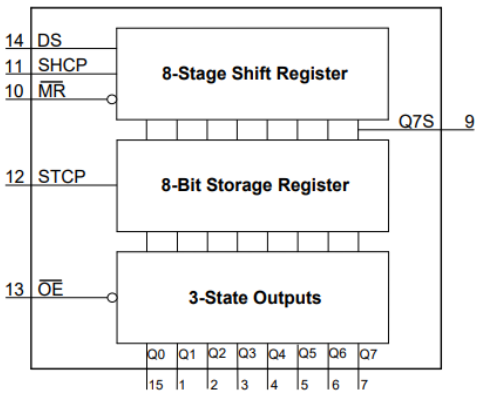
To expand the number of output signals, the experimentation kit supports the 74HC595 shift register IC with a connection diagram as shown in Figure 11.

To use the shift register module, the control signals on J13 are connected to the PORT pins of the microcontroller, and the output pins on J36 are connected to individual LEDs or bar LEDs. Data is shifted out sequentially, then latched and output as per the timing diagram shown in Figure 12.

Pin Descriptions

Pin Number	Pin Name	Function
1	Q1	Parallel Data Output 1
2	Q2	Parallel Data Output 2
3	Q3	Parallel Data Output 3
4	Q4	Parallel Data Output 4
5	Q5	Parallel Data Output 5
6	Q6	Parallel Data Output 6
7	Q7	Parallel Data Output 7
8	GND	Ground
9	Q7S	Serial Data Output
10	$\overline{MR}$	Master Reset Input
11	SHCP	Shift Register Clock Input
12	STCP	Storage Register Clock Input
13	$\overline{OE}$	Output Enable Input
14	DS	Serial Data Input
15	Q0	Parallel Data Output 0
16	V_{CC}	Supply Voltage

Functional Diagram



Control				Input	Output		Function
SHCP	STCP	$\overline{OE}$	$\overline{MR}$	DS	Q7S	Qn	
X	X	L	L	–	L	NC	Low-level asserted on MR clears shift register. Storage register is unchanged.
X	↑	L	L	–	L	L	Empty shift register transferred to storage register.
X	X	H	L	–	L	Z	Shift register remains clear; All Q ouputs in Z state.
↑	X	L	H	–	Q6S	NC	HIGH is shifted into first stage of Shift Register Contents of each register shifted to next register. The content of Q6S has been shifted to Q7S and now appears on device pin Q7S.
X	↑	L	H	–	NC	QnS	Contents of shift register copied to storage register. With output now in active state the storage resister contents appear on Q outputs.
↑	↑	L	H	–	Q6S	QnS	Contents of shift register copied to output register then shift register shifted.

H=HIGH Voltage State
L=LOW Voltage State
↑=LOW to HIGH Transition
X= Don'T Care – High or Low (Not Floating)
NC= No Change
Z= High-Impedance State

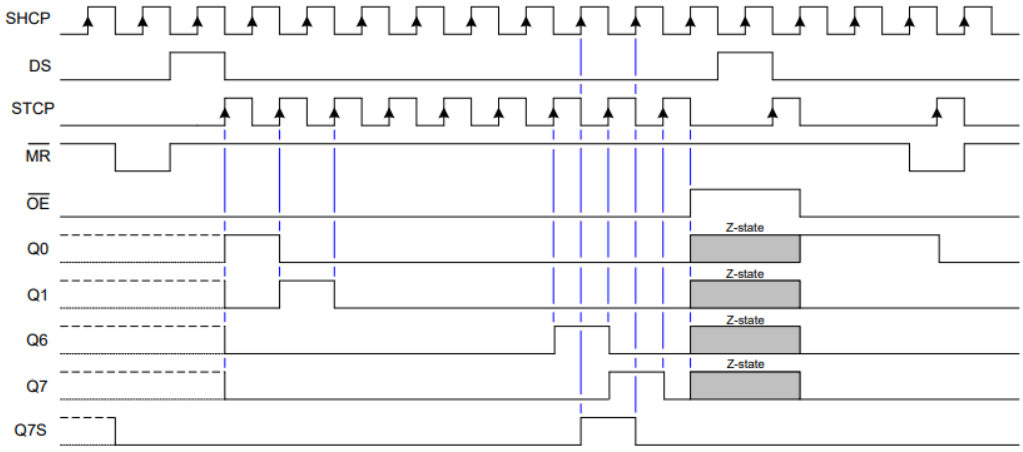


Figure 8: Describe the operation of the 74HC595 shift register

2.3 LED AND SWITCH INTERFACING

To use the switch and LED display blocks, students can use a bus or single wires to connect peripherals to the AVR's port pins. Here is an example program to read the state of a switch connected to pin PB0 and display it on an LED connected to pin PC0.

```
            .cseg
            .org    0x00
start:
             CBI          DDRB,0        ;PB0 is input
             SBI          PORTB,0       ;enable pull up resistor
             SBI          DDRC,0        ;PC0 is output
             CBI          PORTC,0       ;Turn off the LED
MAIN:
             SBIS   PINB,0        ;if swwitch not pressed, skip the jump to release
             RJMP   RELEASE
PRESSED:
             CBI          PORTC,0       ;turn on the LED
             RJMP   MAIN
RELEASE:
             SBI          PORTC,0       ;turn off the LED
              RJMP              MAIN
```

Example 1: Communicate single push button and single LED using assembly

Similarly, you can use C to write a program with the same functionality:

```
#include <avr/io.h>
int main(void)
{
       DDRC = 0X01;
       PORTB |= 0X01;
    /* Replace with your application code */
    while (1)
    {
              if (PINB & (1<<PINB0))      PORTC &= ~(1<<PORTC0);
              else PORTC |= (1<<PORTC0);
    }
}
```

Example 2: Push button and LED interface using C

2.4 DELAYS USING LOOPING STATEMENTS

In practice, controlling peripherals always requires following a specific sequence with a defined time interval between operations. Therefore, programs often include subroutines that create delays, which are essential for proper control.

For instance, if you need to turn an LED on and off to signal that your program is running, blinking the LED too quickly can make it appear continuously on. Hence, it's essential to introduce a delay between LED states, typically around 1 second for a noticeable effect.

There are various ways to create delay subroutines, and this experiment focuses on implementing delays using loop statements. In a microcontroller, each instruction execution takes a certain number of clock cycles, which are typically measured in machine cycles or CPU cycles. Thus, an executed instruction consumes a defined amount of time.

The Atmega 324 microcontroller can be clocked from different sources, depending on how the fuses are configured, as shown in Table 1. Below is an example of creating a DELAY_10ms subroutine with an 8MHz clock frequency.

```
.cseg
.org   0x00
START:
              SBI           DDRC, 0
MAIN:
              CBI           PORTC,0
              CALL          DELAY_10MS
              SBI           PORTC,0
              CALL          DELAY_10MS
              JMP           MAIN
DELAY_10MS:
              LDI           R21,80        ;1MC
L1:           LDI           R20,250       ;1MC
L2:           DEC           R20           ;1MC
              NOP                         ;1MC
              BRNE   L2                   ;2/1MC
```

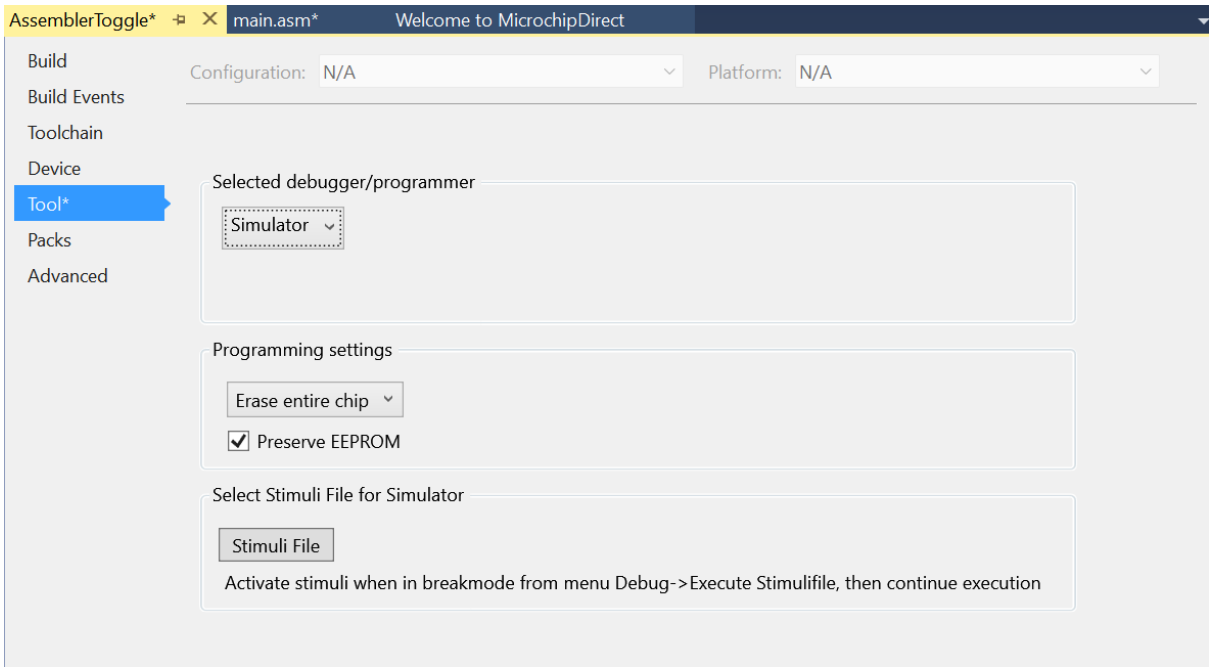
```
        DEC         R21         ;1MC
        BRNE    L1              ;2/1MC
        RET                     ;4MC
```

Example 3: Delay generation using loops (assembly)

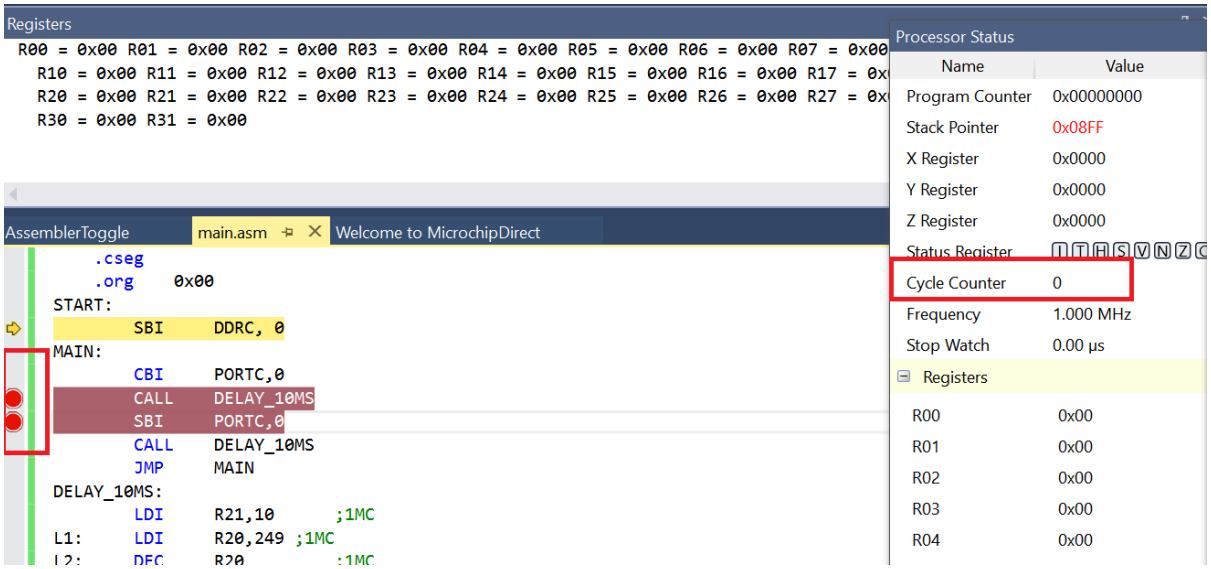
The above is an example of creating a 50Hz frequency signal on pin PC0. The subprogram DELAY_10MS generates a 10ms delay with an 8MHz clock, where 1MC (Machine Cycle) equals 0.125µs, corresponding to 80000MC x 0.125 = 10ms.

To precisely determine how many clock cycles a section of code runs, we use the Cycle Counter feature in the AVR Simulator.

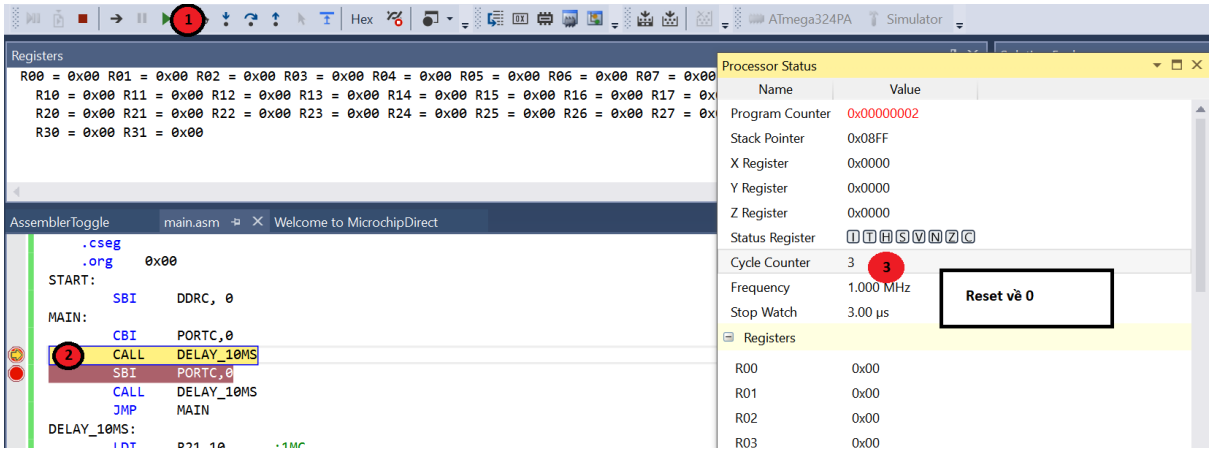
1. Select Simulator Function



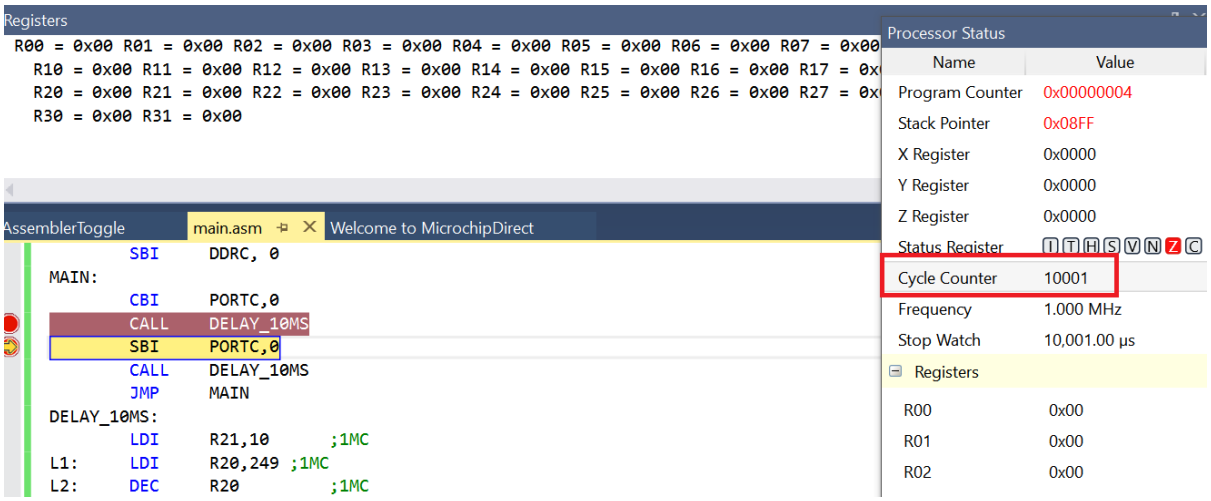
2. Choose Debug-Start Debugging and Break to begin simulation. Select Debug-Windows-Processor Status to open the Processor Status window. Set breakpoints at the beginning and end of the code block you want to measure, in this case, the DELAY_10MS function.



3. Allow the program to run to the first Break Point, click on the Cycle Counter, and reset its value to 0.



4. Continue allowing the program to run until it reaches the second Break Point and make a note of the Cycle Count value. This is the number of clock cycles required to execute this code segment.



If you are using C for programming, you can use simple loops like the following:

```
int main(void)
{
       volatile int i;
    /* Replace with your application code */
       DDRC |= 0x01;
    while (1)
    {
              PORTC ^= 0x01;
              for (i=554;i>0;i--);
    }}
```

Example 4: Delay generation using loops (C)

The count is loaded into the counter variable to adjust the desired delay time using the Simulator as mentioned in the previous section. Additionally, we can use the available library. Note that in order to use the functions in the delay.h library, we need to define the CPU frequency correctly. In the example below, the CPU frequency is set to 1 MHz.

```
#include <avr/io.h>
#define F_CPU 1000000
#include <util/delay.h>

int main(void)
{
volatile int i;
/* Replace with your application code */
DDRC |= 0x01;
while (1)
{
PORTC ^= 0x01;
_delay_ms(10);
}
}
```

Example 5: Delay generation using delay.h library

2.5 DEBOUNCING

With mechanical key presses, the mechanical bounce occurs when the key is pressed or released. Therefore, pressing or releasing a key will generate a sequence of pulses rather than a single pulse as shown in Figure 13.

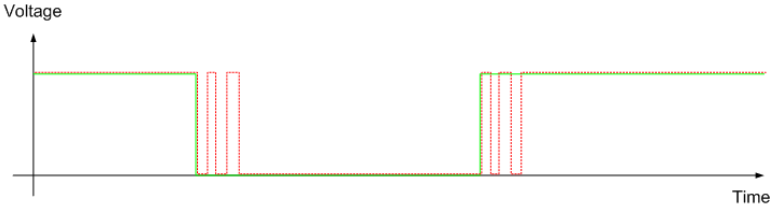


Figure 9: Single key vibrates when pressed or released

When using this signal to determine the number of key presses, the result may be inaccurate. To obtain the correct result, we need to debounce the signal.

The simplest debouncing method is to read the signal twice consecutively after an appropriate delay. If the results of these two reads are the same, we consider it as the correct result. If the result is incorrect, we read it again for the third time, also after a time period T. If the result matches the second read, we accept it as correct, otherwise, we repeat the process. The time period T is typically chosen as 50ms.

We can increase accuracy by reducing the time period T and increasing the number of reads. For example, a key can be considered as returning a value of 1 if we read 20 consecutive times with the value 1, with each read spaced 1ms apart.

2.6 INTERFACE WITH SHIFT REGISTERS

Shift registers are used when we want to expand the display signals without using multiple port pins. These shift registers can be daisy-chained to increase the number of signals.

Data is serially shifted into the register and then output to the output pins of the IC74HC595. The control waveform is described in section 2.2.5.

Here's an example of controlling a shift register to output data using assembly and C. The connections to the AVR microcontroller are as follows, connecting the output header of the shift register to the header of the BARLED as shown in the table below.

PB0	SDI
PB1	LATCH
PB2	CLOCK
PB3	MCLR

```
.include "m324padef.inc"  ; Include Atmega324pa definitions
.def shiftData = r20      ; Define the shift data register
.equ clearSignalPort = PORTB   ; Set clear signal port to PORTB
.equ clearSignalPin = 3        ; Set clear signal pin to pin 0 of PORTB
.equ shiftClockPort = PORTB    ; Set shift clock port to PORTB
.equ shiftClockPin = 2         ; Set shift clock pin to pin 1 of PORTB
.equ latchPort = PORTB       ; Set latch port to PORTB
.equ latchPin = 1              ; Set latch pin to pin 0 of PORTB
.equ shiftDataPort = PORTB     ; Set shift data port to PORTB
.equ shiftDataPin = 0          ; Set shift data pin to pin 3 of PORTB
main:
       call   initport
       ldi            shiftData,0x55
       call   cleardata
       call   shiftoutdata
stop:
       jmp            stop
; Initialize ports as outputs
initport:
```

```
        ldi r24,
(1<<clearSignalPin)|(1<<shiftClockPin)|(1<<latchPin)|(1<<shiftDataPin)
        out DDRB, r24         ; Set DDRB to output
        ret
        ldi    shiftData,0x55
cleardata:
        cbi clearSignalPort, clearSignalPin       ; Set clear signal pin to low
; Wait for a short time
        sbi clearSignalPort, clearSignalPin       ; Set clear signal pin to high
        ret
; Shift out data
shiftoutdata:
        cbi shiftClockPort, shiftClockPin       ;
        ldi r18, 8               ; Shift 8 bits
shiftloop:
          sbrc shiftData, 7      ; Check if the MSB of shiftData is 1
          sbi shiftDataPort, shiftDataPin   ; Set shift data pin to high
          sbi shiftClockPort, shiftClockPin   ; Set shift clock pin to high
          lsl shiftData           ; Shift left
          cbi shiftClockPort, shiftClockPin   ; Set shift clock pin to low
          cbi shiftDataPort, shiftDataPin   ; Set shift data pin to low
          dec r18
          brne shiftloop
; Latch data
        sbi latchPort, latchPin   ; Set latch pin to high
        cbi latchPort, latchPin   ; Set latch pin to low
        ret
```

Example 6: Assembly code to shift data using 74HC595

```
#include <avr/io.h>
#define       F_CPU  1000000UL
#include <util/delay.h>

#define clearSignalPort PORTB
#define clearSignalPin 3

#define shiftClockPort PORTB
#define shiftClockPin 2

#define latchPort PORTB
#define latchPin 1
#define shiftDataPort PORTB
#define shiftDataPin 0

void initializePorts() {
        // Set output pins
        DDRB |= (1 << clearSignalPin)|(1 << shiftClockPin) | (1 << latchPin)|(1 <<
shiftDataPin);
}

void clearShiftRegister() {
        // Set clear signal pin to low
        clearSignalPort &= ~(1 << clearSignalPin);
        _delay_us(1);
        // Set clear signal pin to high
        clearSignalPort |= (1 << clearSignalPin);
}

void shiftOut(uint8_t data) {
```

```
        // Set latch pin to low
        latchPort &= ~(1 << latchPin);
        // Shift out 8 bits
        for (int i = 0; i < 8; i++) {
                if (data & 0x80) {
                        // Set shift data pin to high
                        shiftDataPort |= (1 << shiftDataPin);
                        } else {
                        // Set shift data pin to low
                        shiftDataPort &= ~(1 << shiftDataPin);
                }
                // Set shift clock pin to high
                shiftClockPort |= (1 << shiftClockPin);
                // Set shift clock pin to low
                shiftClockPort &= ~(1 << shiftClockPin);
                // Shift data left by 1 bit
                data <<= 1;
        }

        // Set latch pin to high
        latchPort |= (1 << latchPin);
        _delay_us(1);
        // Set latch pin to low
        latchPort &= ~(1 << latchPin);}
int main() {
        uint8_t data = 0x55;
        initializePorts();
        clearShiftRegister();
        shiftOut(data);
        while (1) {
                // Your main program loop goes here
                _delay_ms(1000);
                data = (data << 1) | (data >> 7);
                shiftOut(data);
        }
        return 0;}
```

Example 7: C code to shift data using 74HC595

CHAPER 3 MATRIX KEYPAD

3.1 BASIC THEORY

When interfacing with multiple keys on a microcontroller, to save the number of port pins, we will connect the keys in a matrix format instead of connecting each key to an individual port pin as discussed in Chapter 2.

In a keypad matrix, the keys are organized in a grid format, where one end of each key is connected to rows, and the other end is connected to columns as shown below:

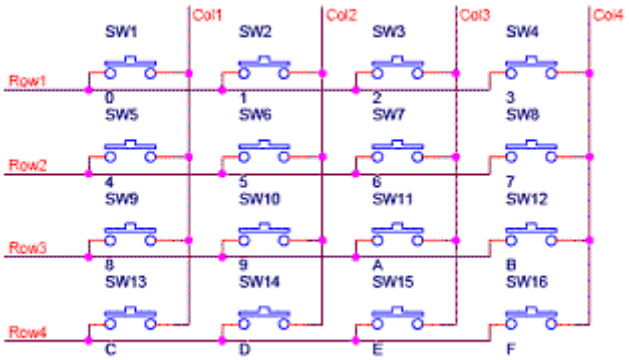


Figure 10: Connection of a 4x4 keypad matrix

With this type of connection, when we press a key, a row and a column will be connected together. Based on the row and column numbers, we can determine the position of the pressed key.

3.2 KEYPAD MATRIX INTERFACE

On the experimental kit, the keypad matrix consists of 16 keys organized into 4 rows and 4 columns, and it's connected to header J17.

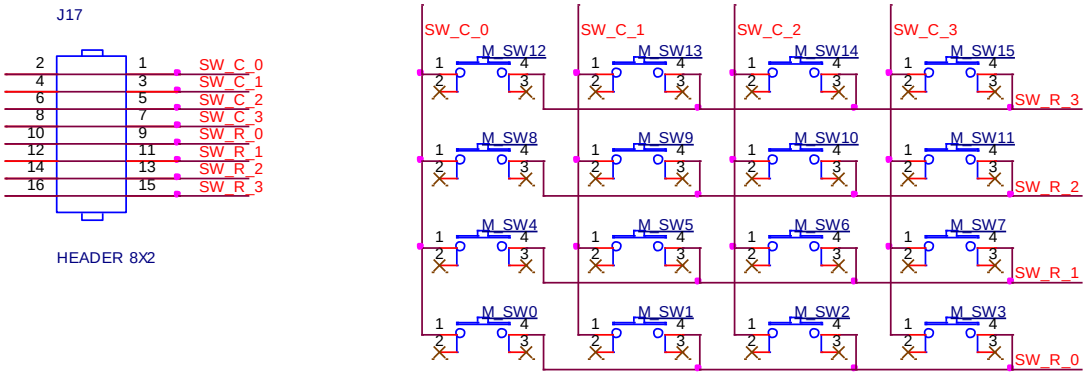


Figure 11: Keypad matrix block diagram

To interface with the keypad matrix, you connect header J17 to one of the AVR's ports. We use key scanning techniques to detect if any key is pressed. The keypad can be scanned either by rows or by columns. Below is the process for scanning the keypad by columns.

Connect header J17 to one of the AVR's ports, for example, PORTC. In this case, the columns SW_C3 to SW_C0 are connected to PC3 to PC0, and the rows SW_R3 to SW_R0 are connected to PC7 to PC4.

To scan by columns, configure the pins interfacing with the columns as outputs and the pins interfacing with the rows as inputs.

Set column 0 to 0 and the other 3 columns to 1. If any key on column 0 is pressed, the corresponding row will be connected to column 0 and will have a value of 0. This way, during this scan, we know which key is pressed in column 0.

Similarly, set column 1 to 0 and the other 3 columns to 1. If any key on column 1 is pressed, the corresponding row will be connected to column 1 and will have a value of 0. This way, during this scan, we know which key is pressed in column 1.

Repeat this process for column 2 and column 3.

In this way, one scanning cycle will sequentially scan all 4 columns and detect if any key is pressed across the entire keypad matrix.

The time it takes to scan all 4 columns is very quick compared to the speed of human key presses and releases, ensuring that the key scanning process will detect any key presses.

Below is a subroutine for scanning a keypad matrix connected to PORT A. The columns from C0 to C3 are connected to PA0 to PA3, and the rows from R0 to R3 are connected to PA4 to PA7.

```
; ATmega324PA keypad scan function
; Scans a 4x4 keypad connected to PORTA
;C3-C0 connect to PA3-PA0
;R3-R0 connect to PA7-PA4
; Returns the key value (0-15) or 0xFF if no key is pressed

keypad_scan:
    ldi r20, 0b00001111 ; set upper 4 bits of PORTD as input with pull-up, lower 4
bits as output
    out DDRA, r20
       ldi r20, 0b11111111  ; enable pull up resistor
    out PORTA, r20

    ldi r22, 0b11110111 ; initial col mask
    ldi r23, 0 ; initial pressed row value
       ldi    r24,3  ;scanning col index

keypad_scan_loop:
    out PORTA, r22 ; scan current col
       nop                          ;need to have 1us delay to stablize
```

```
    sbic PINA, 4 ; check row 0
    rjmp keypad_scan_check_col2
    rjmp keypad_scan_found ; row 0 is pressed
keypad_scan_check_col2:
    sbic PINA, 5 ; check row 1
    rjmp keypad_scan_check_col3
    ldi r23, 1 ; row1 is pressed
    rjmp keypad_scan_found
keypad_scan_check_col3:
    sbic PINA, 6 ; check row 2
    rjmp keypad_scan_check_col4
    ldi r23, 2 ; row 2 is pressed
    rjmp keypad_scan_found
keypad_scan_check_col4:
    sbic PINA, 7 ; check row 3
    rjmp keypad_scan_next_row
    ldi r23, 3 ; row 3 is pressed
    rjmp keypad_scan_found

keypad_scan_next_row:
    ; check if all rows have been scanned
    cpi r24,0
    breq keypad_scan_not_found

    ; shift row mask to scan next row
    ror r22
       dec    r24           ;increase row index
    rjmp keypad_scan_loop

keypad_scan_found:
    ; combine row and column to get key value (0-15)
       ;key code = row*4 + col
    lsl r23 ; shift row value 4 bits to the left
       lsl    r23
    add r23, r24 ; add row value to column value
    ret

keypad_scan_not_found:
    ldi r23, 0xFF ; no key pressed
    ret
```

Example 8: Scans keys assembly program

```
//ATmega324PA keypad scan function
//Scans a 4x4 keypad connected to PORTA
//C3-C0 connect to PA3-PA0
//R3-R0 connect to PA7-PA4
//Returns the key value (0-15) or -1 if no key is pressed

int8_t keypad_scan() {

       uint8_t col_mask = 0b11110111; // initial row mask
       int8_t scan_col;                  //scanning col index

       DDRA = 0x0F; // set upper 4 bits of PORTA as input with pull-up, lower 4 bits
as output
       PORTA = 0xFF;//enable pullup resistor on the input pin

       for (scan_col=3;scan_col>=0;scan_col--) {
              PORTA = col_mask; // scan current col
              _delay_us(1); // wait for stable input
```

```
            if ((PINA & (1 << PINA4)) == 0) { // check row 0
                  return  (scan_col) ; // calculate key value as (row index * 4)
+ column index
                  }
            else if ((PINA & (1 << PINA5)) == 0) { // check row 1
                  return  (4 + scan_col) ; // calculate key value as (row index *
4) + column index
                  }
            else if ((PINA & (1 << PINA6)) == 0) { // check row 2
                  return  (8 + scan_col) ; // calculate key value as (row index *
4) + column index
                  }
             else if ((PINA & (1 << PINA7)) == 0) { // check row 3
                  return  (12 + scan_col) ; // calculate key value as (row index
* 4) + column index
            }

            col_mask = ((col_mask >> 1) | (1<<7)) ; // shift row mask to scan next
row
            }

      return -1;
}
```

Example 9: Scans keys C program

3.3 USING 'AND' GATES TO GENERATE INTERRUPTS ON KEY PRESS

To detect the key presses event and handle them, we can continuously scan the keys. The following program segment will scan the keys and display the key code on a Bar LED using the keypad_scan and shiftoutdata subroutines.

```
call   initport
       call   cleardata
main:
       call   keypad_scan
       mov           shiftData,r23

       call   shiftoutdata

       jmp           main
```

However, it can be observed that the CPU has to continuously scan the keys to avoid missing any keypress events. This would consume CPU operating time, leaving no time for other tasks. Additionally, when the CPU is executing a time-consuming task, it can lead to unresponsiveness to keypresses because the keypad scanning routine isn't being called.

To address this issue, we can design the system to generate an interrupt when a key is pressed or released. The keypad scanning program will be called immediately upon an interrupt.

The experimental kit has 2 AND gates designed as shown in Figure 16. To set up the experiment, we use a wire bus to connect header J17 of the keypad block to header J55, and another wire bus to connect J53 to a port of the AVR for scanning the keys.

In this configuration, the signals C0-C3 will be connected to signals A, B, C, D, and R0-R3 will be connected to signals E, F, G, H of the AND gate block. If we scan the keys by columns, the signals R0-R3 will be inputs, and the signal OUT1 = R0 & R1 & R2 & R3.

Connect the OUT1 signal to a port pin that supports external interrupts (PD3, PD2, PB2). Configure the interrupt to trigger on a low level if you want continuous key scanning while a key is pressed. If you only want to scan once when a key is pressed or released, configure the external interrupt to trigger on the falling edge.

Normally, we set the column signals COL3..0 to logic 0. When no key is pressed, the row signals R3..R0 will be logic 1 due to pull-up resistors, making OUT1 logic 1.

When any key is pressed, one of the row signals R3..R0 will be pulled down to logic 0, making OUT1 logic 0 and triggering the interrupt.

The program will then jump to the interrupt service routine. After scanning and processing the keys in the interrupt.

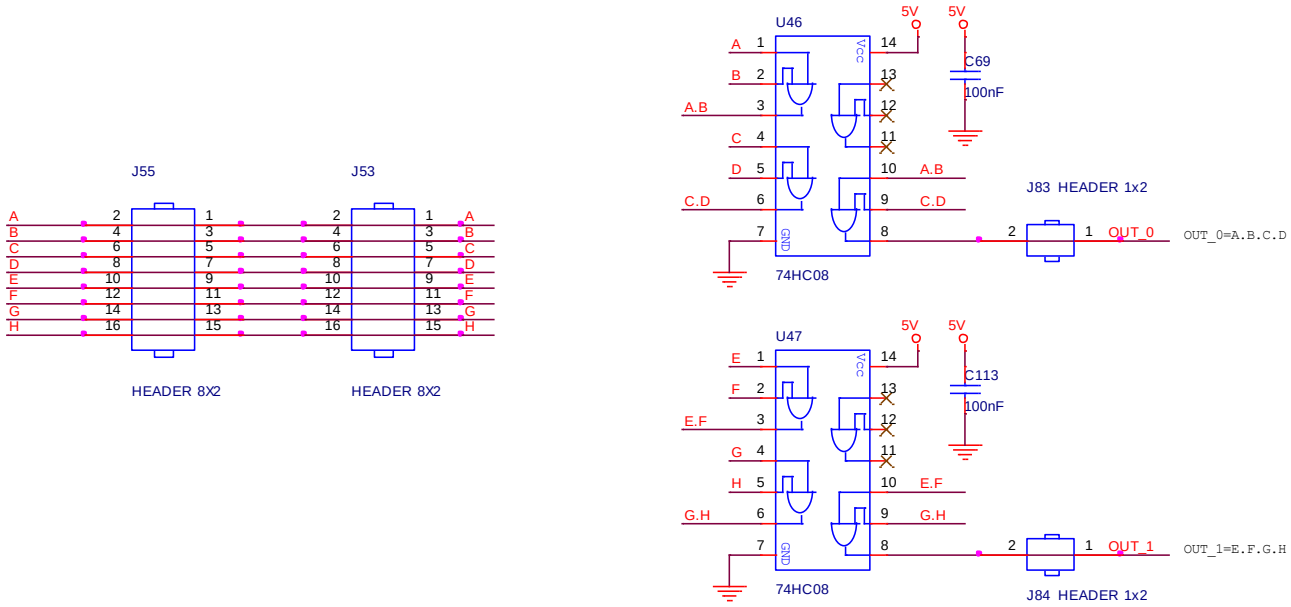


Figure 12: AND Gate

Note that the port pins of the Atmega324 support PINCHANGE interrupts; we can use these interrupts without having to use the AND gate. The following program scans the keys and displays them on the LED bar using the PINCHANGE interrupt on pins PA4, PA5, PA6, PA7.

```
.equ BUTTON_MASK = (1<<PA4)|(1<<PA5)|(1<<PA6)|(1<<PA7) ; mask for checking button
state

.org 0x0000 ; interrupt vector table
rjmp reset_handler ; reset

.org 0x001C ; pin change interrupt vector
```

```
rjmp pinchange_handler

reset_handler:
    ; initialize stack pointer
    ldi r16, high(RAMEND)
    out SPH, r16
    ldi r16, low(RAMEND)
    out SPL, r16
       call   initport
       call   cleardata
       call   keypad_prepare_port
    ; enable pin change interrupt for PA3, PA2, PA1, and PA0
    ldi r16, BUTTON_MASK
    sts PCMSK0, r16
    ldi r16, (1<<PCIE0)
    sts PCICR, r16
    ; enable global interrupts
    sei
main:
       jmp           main
; Set PA7..PA4 as input, PA3..PA0 as output
;set PA3..PA0 to 0
keypad_prepare_port:
    ldi r20, 0b00001111 ; set upper 4 bits of PORTD as input with pull-up, lower 4
;bits as output
    out DDRA, r20
       ldi r20, 0b11110000  ; enable pull up resistor
    out PORTA, r20
       ret
pinchange_handler:
    ; check if PA7, PA6, PA5, or PA4 has changed
    in r16, PINA
    andi r16, BUTTON_MASK
    breq pinchange_handler_exit ; exit if none of the buttons have changed

       call   keypad_scan                ;scan and shiftout keycode
       mov    shiftData,r23
       call   shiftoutdata

pinchange_handler_exit:
       call   keypad_prepare_port
    reti ; return from interrupt
```

Example 10: Key scanning and display using PORTCHANGE (Assembly) interrupt

```
#define BUTTON_MASK ((1 << PA7) | (1 << PA6) | (1 << PA5) | (1 << PA4))
void portIntPrepare()
{
              DDRA = 0x0F; // set upper 4 bits of PORTA as input with pull-up, lower
4 bits as output
              PORTA = 0xF0;//enable pullup resistor on the input pin, clear 4 lower
bit
              return;
}
void portchangeIntEna()
{
           // enable pin change interrupt for PA7, PA6, PA5, and PA4
           PCMSK0 |= BUTTON_MASK;
           PCICR |= (1 << PCIE0);
           // enable global interrupts
```

```
            sei();
}

int main() {

        initializePorts();
        portIntPrepare();
        clearShiftRegister();
        portchangeIntEna();
        while (1) {
                //shiftOut(keypad_scan());
        }
        return 0;
}
ISR(PCINT0_vect) {
        shiftOut(keypad_scan());
        portIntPrepare();
}
```

Example 11: Key scanning and display using PORTCHANGE (C) interrupt

CHAPER 4 7-SEGMENT LED

4.1 BASIC THEORY

7-segment LEDs are commonly used for displaying simple numerical digits or letters. This form of communication allows individuals without technical expertise to interact with a system by reading information displayed on the LEDs. For example, multiple 7-segment LEDs can be used to display phone numbers at public telephone booths or time values at traffic signal intersections.

A 7-segment LED display consists of 8 individual LEDs connected with a common anode (common anode type) or common cathode (common cathode type). These individual LED segments are named a, b, c, d, e, f, g, and the decimal point (dp). To display a value on a 7-segment LED, you need to apply voltage to the common pin and control the segment pins in a manner similar to controlling individual LEDs.

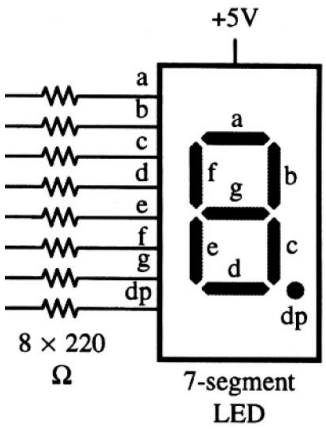


Figure 17: Common anode 7-segment LED display

To display a numerical value on a 7-segment LED, programmers need to output values to control the segments a, b, ..., g, and dp. However, data in microcontroller systems is typically in binary format, which cannot be directly displayed on a 7-segment LED. Therefore, a conversion from binary representation to 7-segment LED representation is required. This conversion can be done through hardware with a code converter IC or through software using methods like table lookup.

4.2 HARDWARE DESIGN

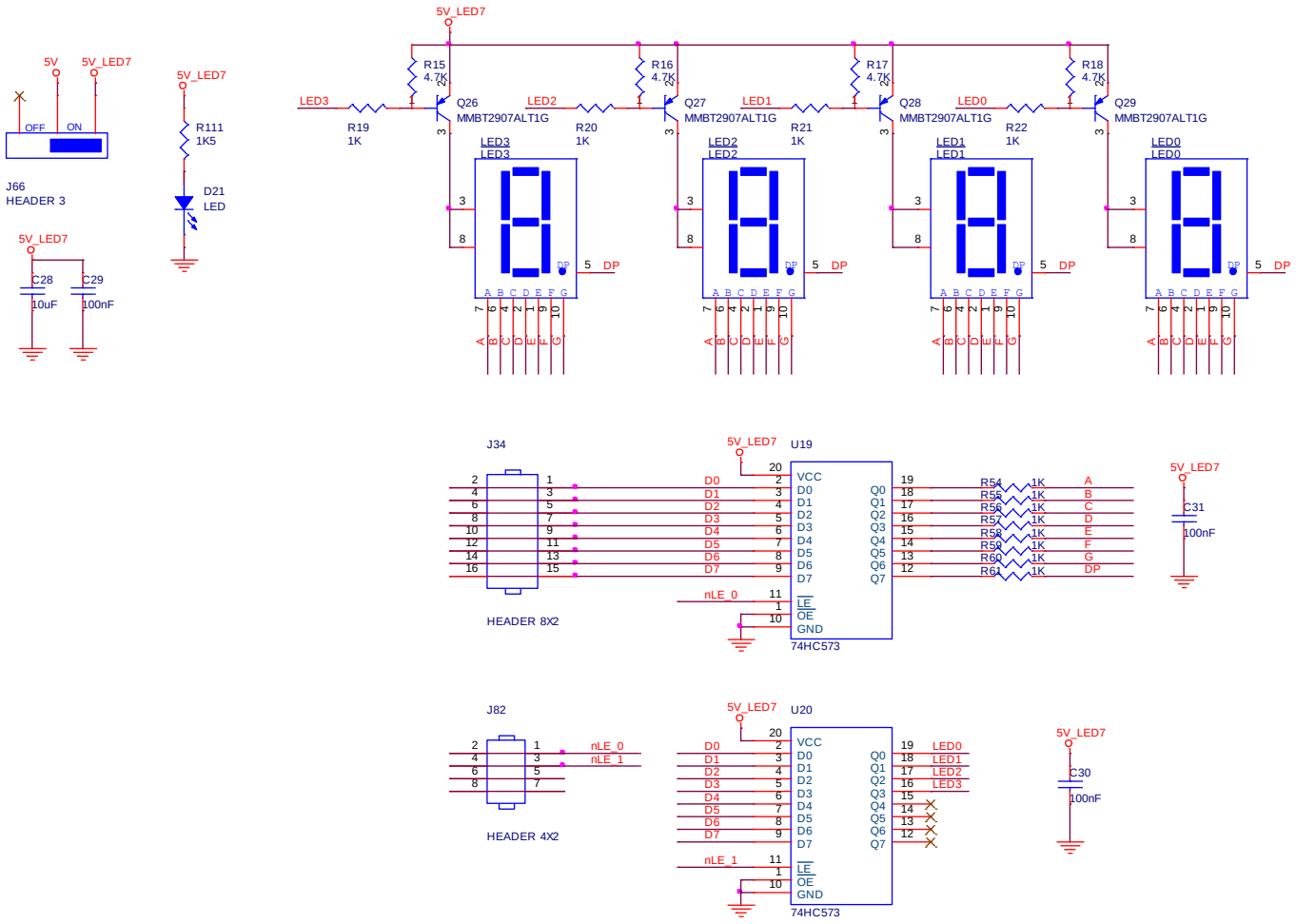


Figure 13: 7-segment LED block diagram

The 7-segment LED block consists of 4 common anode LEDs designed for LED scanning. Two ICs, 74HC573 (U19) and 74HC573 (U20), share input data and connect to header J34. The control signals nLE0 and nLE1 are available on header J82.

The segment signals A to G and DP of the 7-segment LEDs are shared and connected to the output of the 74HC573 latch (U19). This latch is controlled by the nLE0 signal on header J82. **To illuminate segments, you must output logic low to the corresponding segment pins and logic high to turn them off.**

The 7-segment LEDs can be turned on by enabling BJTs to provide power to the LEDs. A logic low signal at the base (B) of BJTs Q25, Q26, Q27, Q28 will saturate the BJTs, causing the collector (C) of the BJTs to have a value near 5V (close to the voltage at the emitter, E), supplying 5V power to the 7-segment LEDs. The enable signals for the BJT LED3, LED2,

LED1, LED0 are connected to the output of latch IC U20. This IC is controlled by the nLE1 signal on header J82.

The LED block is powered through Jumper J66. Before programming, set the Jumper to the ON position to supply power to the 7-segment LED block.

4.3 DISPLAY ON 7-SEGMENT LEDs

4.3.1 Displaying on a Single LED:

As seen in the hardware design, to display a number on a 7-segment LED, programmers need to output the 7-segment code for the desired number or character to the input of the '573 U19 IC. To latch this data to the output of the 74HC573 IC, first, raise the nLE0 signal to 1 to bring the data out, then bring it down to 0 to latch it. Then, provide the appropriate data to latch into U23 by setting the data on header J34 and the positive edge of the nLE1 signal.

INPUTS			OUTPUTQ
$\overline{OE}$	LE	D	
L	H	H	H
L	H	L	L
L	L	X	Q_0
H	X	X	Z

Table 1 Truth Table of IC 74HC573

Suppose we want to display the number 1 on LED 3. First, we need to output the 7-segment code for the number 1, which is F9, to IC U19. Then, we turn on LED 3 by outputting the value 0x07 to IC U23 to activate LED3.

Số	Số nhị phân								HEX
	7	6	5	4	3	2	1	0	
	dp	g	f	e	d	c	b	a	
0	1	1	0	0	0	0	0	0	C0
1	1	1	1	1	1	0	0	1	F9
2	1	0	1	0	0	1	0	0	A4
3	1	0	1	1	0	0	0	0	B0
4	1	0	0	1	1	0	0	1	99
5	1	0	0	1	0	0	1	0	92
6	1	0	0	0	0	0	1	0	82
7	1	1	1	1	1	0	0	0	8F
8	1	0	0	0	0	0	0	0	80
9	1	0	0	1	0	0	0	0	90
A	1	0	0	0	1	0	0	0	88
B	1	0	0	0	0	0	1	1	83
C	1	1	0	0	0	1	1	0	C6
D	1	0	1	0	0	0	0	1	A1
E	1	0	0	0	0	1	1	0	86
F	1	0	0	0	1	1	1	0	8E

Table 2: 7-Segment Code Table for Common Anode LED

The following program initializes the port pins and sends values to the 7-segment LED. J34 is connected to PORTD, nLE0 is connected to PB4, and nLE1 is connected to PB5.

```
; Lookup table for 7-segment codes
table_7seg_data:     .DB 0XC0, 0XF9,0XA4,0XB0,0X99,0X92,0X82,0XF8,0X80,0X90,0X88,0X8
                     .DB           0XC6,0XA1,0X86,0X8E
; Lookup table for LED control
table_7seg_control:
                     .DB           0b00001110,0b00001101, 0b00001011, 0b00000111
;             J34 connect to PORTD
;             nLE0 connect to PB4
;             nLE1 connect to PB5
; Output: None
.equ   LED7SEGPORT = PORTD
.equ   LED7SEGDIR =  DDRD
.equ   LED7SEGLatchPORT = PORTB
.equ   LED7SEGLatchDIR =  DDRB
.equ   nLE0Pin                     =      4
.equ   nLE1Pin                     =      5
led7seg_portinit:
       push   r20
       ldi r20, 0b11111111 ; SET led7seg PORT as output
       out LED7SEGDIR, r20
       in r20, LED7SEGLatchDIR     ; read the Latch Port direction register
    ori r20, (1<<nLE0Pin) | (1 << nLE1Pin)
       out LED7SEGLatchDIR,r20
       pop    r20
       ret
; Display a value on a 7-segment LED using a lookup table
; Input: R27 contains the value to display
;             R26 contain the LED index (3..0)
;             J34 connect to PORTD
;             nLE0 connect to PB4
;             nLE1 connect to PB5
; Output: None
display_7seg:
    push r16 ; Save the temporary register

    ; Look up the 7-segment code for the value in R18
```

```
    ; Note that this assumes a common anode display, where a HIGH output turns OFF
the segment
    ; If using a common cathode display, invert the values in the table above
       ldi          zh,high(table_7seg_data<<1) ;
       ldi          zl,low(table_7seg_data<<1)  ;
       clr          r16
    add             r30, r27
       adc          r31,r16

    lpm             r16, z
       out          LED7SEGPORT,r16
       sbi          LED7SEGLatchPORT,nLE0Pin
       nop
       cbi          LED7SEGLatchPORT,nLE0Pin
       ldi          zh,high(table_7seg_control<<1)     ;
       ldi          zl,low(table_7seg_control<<1)      ;
       clr          r16
    add             r30, r26
       adc          r31,r16
       lpm          r16, z
       out          LED7SEGPORT,r16
       sbi          LED7SEGLatchPORT,nLE1Pin
       nop
       cbi          LED7SEGLatchPORT,nLE1Pin
    pop             r16 ; Restore the temporary register
    ret ; Return from the function
```

Example 12: Program to display 7-segment LED using assembly

```
#define LED7SEGPORT  PORTD
#define LED7SEGDIR  DDRD

#define LED7SEGLatchPORT  PORTB
#define LED7SEGLatchDIR   DDRB

#define nLE0Pin                   4
#define nLE1Pin                   5

const uint8_t led7seg_code[16] = {0XC0,
0XF9,0XA4,0XB0,0X99,0X92,0X82,0XF8,0X80,0X90,0X88,0X8,0XC6,0XA1,0X86,0X8E};
const uint8_t led7seg_control[4] = {0b00000111,0b00001011,0b00001101, 0b00001110};

void led7segInitializePorts() {
       // Set output pins
       LED7SEGDIR = 0xFF;
       LED7SEGLatchDIR |= (1<<nLE0Pin) | (1 << nLE1Pin);
       return;
```

```
}
void    led7segDisplay(uint8_t      value, uint8_t ledIndex)
{
        LED7SEGPORT = led7seg_code[value];
        LED7SEGLatchPORT |= (1<<nLE0Pin);
        LED7SEGLatchPORT &= ~(1<<nLE0Pin);

        LED7SEGPORT = led7seg_control[ledIndex];
        LED7SEGLatchPORT |= (1<<nLE1Pin);
        LED7SEGLatchPORT &= ~(1<<nLE1Pin);
        return;
}
```

Example 13: Program to display 7-segment LED using C

4.3.2 Display Simultaneously on Multiple 7-Segment LEDs

The design of the experimental kit does not allow for displaying four different numbers simultaneously on all four LEDs. To achieve this, we use LED scanning.

For example, if we want to display the number 1234 on four LEDs, we'll have LED0 display the number 4 for a period of time T, then turn off LED0 and have LED1 display the number 3, also for the duration of T. Similarly, we do this for LED2 and LED3.

As a result, the LEDs will light up and turn off sequentially. In one second, the number of times each LED lights up will be 1s/4T.

Thanks to the persistence of vision effect, if the LEDs blink rapidly enough, the human eye won't perceive the flickering. In theory, if the number of times the LED lights up and turns off is greater than 24 times in 1 second, the human eye will perceive the LED as continuously lit. Therefore, the smaller the time T, the more stable the image appears to the human eye.

On the experimental kit with 4 LEDs, if we choose a scanning frequency of 25Hz, the lighting time for each LED will be 10ms.

The following program will display the number 1234 on the four 7-segment LEDs on the experimental kit.

In practice, with a lighting time of 10ms for each LED (meaning a scanning frequency of 25Hz), you may still perceive the LED blinking. Typically, to ensure stable LED illumination, a scanning frequency of 50Hz is used.

```
        clr     r26                     ;start index = 0
        ldi     r27,1           ;start value = 1
main:

        call    display_7seg
        call    DELAY_10MS
        inc     r26
        inc     r27
```

```
        andi    r26,0x0f
        cpi     r26,4
        brne    main
        ldi     r27,1
        clr     r26

        jmp     main
```

Example 14: Scanning LEDs using a delay loop

4.4 Scanning LEDs Using Timer Interrupts

The method of scanning LEDs based on delay functions, as discussed in section 4.3.2, has the advantage of simplicity but comes with a significant drawback: it utilizes the entire CPU time to create delays, leaving no resources for other tasks. If you try to insert other tasks within the LED display loop, the timing of the display process will be disrupted, causing uneven illumination of LEDs or even flickering.

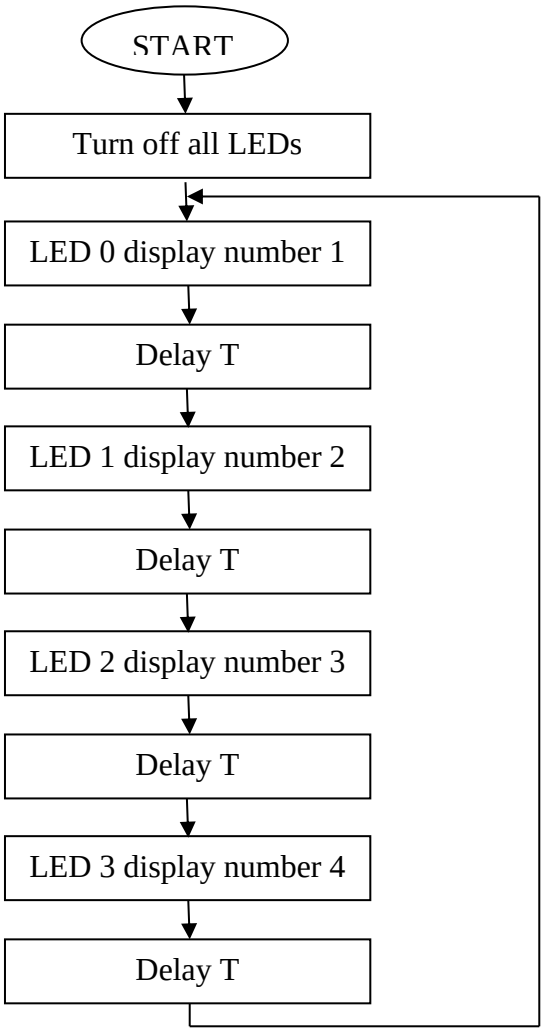


Figure 14: Flowchart of LED scanning program

To address this issue, we will use timer interrupts to generate interrupts at suitable intervals. When an interrupt occurs, we will update the data and activate the corresponding LED.

The following example program displays four values stored in memory locations 0x0100, 0x0101, 0x0102, and 0x0103 on the four 7-segment LEDs on the kit.Timer 1 is configured to generate an interrupt every 10 ms. When the first interrupt occurs, LED3 will display the value at memory location 0x0100, and the other 3 LEDs will be turned off. When the second interrupt occurs, LED2 will display the value at memory location 0x0101. This process will continue for the remaining LEDs and repeat in a loop.

The LED's current position is stored in the **LED7segIndex** memory location. This counter will be automatically updated during the scanning process.

With this approach, the total CPU time used for LED scanning is minimal, ensuring that LEDs are displayed correctly according to the specified requirements.

The values displayed on the LEDs are changed by writing values to four memory locations in RAM at **LED7segValue**. These memory locations act as a "frame buffer" for the display.

For example, if you want to display the numbers 1-2-3-4 on the four LEDs, you would sequentially write the values 1-2-3-4 to four consecutive memory locations starting from **LED7segValue.**

```
.include "m324padef.inc"  ; Include Atmega324pa definitions
.org 0x0000 ; interrupt vector table
rjmp reset_handler ; reset
.org 0x001A
rjmp timer1_COMP_ISR
reset_handler:
    ; initialize stack pointer
    ldi r16, high(RAMEND)
    out SPH, r16
    ldi r16, low(RAMEND)
    out SPL, r16
    ldi r16, (1<<PCIE0)
    sts PCICR, r16
    call initTimer1CTC
    call      led7seg_portinit
    call      Led7seg_buffer_init

    ; enable global interrupts
    sei
main:
       jmp           main
; Lookup table for 7-segment codes
table_7seg_data:
    .DB              0XC0,0XF9,0XA4,0XB0,0X99,0X92,0X82,0XF8,0X80,0X90,0X88,0X83
    .DB              0XC6,0XA1,0X86,0X8E
; Lookup table for LED control
table_7seg_control:
    .DB              0b00001110,0b00001101, 0b00001011, 0b00000111
;             J34 connect to PORTD
;             nLE0 connect to PB4
```

```
;             nLE1 connect to PB5
; Output: None

.equ   LED7SEGPORT = PORTD
.equ   LED7SEGDIR =  DDRD
.equ   LED7SEGLatchPORT = PORTB
.equ   LED7SEGLatchDIR =  DDRB
.equ   nLE0Pin                    =      4
.equ   nLE1Pin                    =      5
.dseg
.org   SRAM_START                 ;starting address is 0x100
              LED7segValue: .byte  4 ;store the BCD value to display
              LED7segIndex: .byte  1
.cseg
.align 2
;init the Led7seg buffer
Led7seg_buffer_init:
       push          r20
       ldi           r20,3                              ;LED index start at 3
       ldi           r31,high(LED7segIndex)
       ldi           r30,low(LED7segIndex)
       st            z,r20
       ldi           r20,0
       ldi           r31,high(LED7segValue)
       ldi           r30,low(LED7segValue)
       st            z+,r20                             ;display value is 0-1-2-3
       inc           r20
       st            z+,r20
       inc           r20
       st            z+,r20
       inc           r20
       st            z+,r20
       pop           r20
       ret

led7seg_portinit:
       push   r20
       ldi r20, 0b11111111 ; SET led7seg PORT as output
       out LED7SEGDIR, r20
       in r20, LED7SEGLatchDIR     ; read the Latch Port direction register
    ori r20, (1<<nLE0Pin) | (1 << nLE1Pin)
       out LED7SEGLatchDIR,r20
       pop    r20
       ret;

;Display a value on a 7-segment LED using a lookup table
; Input: R27 contains the value to display
;             R26 contain the LED index (3..0)
;             J34 connect to PORTD
;             nLE0 connect to PB4
;             nLE1 connect to PB5
; Output: None
display_7seg:
    push r16 ; Save the temporary register

    ; Look up the 7-segment code for the value in R18
    ; Note that this assumes a common anode display, where a HIGH output turns OFF
the segment
    ; If using a common cathode display, invert the values in the table above
       ldi           zh,high(table_7seg_data<<1) ;
       ldi           zl,low(table_7seg_data<<1)  ;
```

```
        clr             r16
    add                 r30, r27
        adc             r31,r16

    lpm                 r16, z
        out             LED7SEGPORT,r16
        sbi             LED7SEGLatchPORT,nLE0Pin
        nop
        cbi             LED7SEGLatchPORT,nLE0Pin
        ldi             zh,high(table_7seg_control<<1)     ;
        ldi             zl,low(table_7seg_control<<1)      ;
        clr             r16
    add                 r30, r26
        adc             r31,r16
        lpm             r16, z
        out             LED7SEGPORT,r16
        sbi             LED7SEGLatchPORT,nLE1Pin
        nop
        cbi             LED7SEGLatchPORT,nLE1Pin
    pop                 r16 ; Restore the temporary register
    ret ; Return from the function

initTimer1CTC:
         push r16
         ldi r16, high(10000)  ; Load the high  yte into the temporary register
         sts OCR1AH, r16       ; Set the high byte of the timer 1 compare value
         ldi r16, low(10000)   ; Load the low byte into the temporary register
         sts OCR1AL, r16       ; Set the low byte of the timer 1 compare value
         ldi r16,  (1 << CS10)| (1<< WGM12) ; Load the value 0b00000101 into the
temporary register
         sts TCCR1B, r16       ;
         ldi r16, (1 << OCIE1A); Load the value 0b00000010 into the temporary
register
         sts TIMSK1, r16       ; Enable the timer 1 compare A interrupt
         pop  r16
         ret

timer1_COMP_ISR:
        push   r16
        push   r26
        push   r27
        ldi             r31,high(LED7segIndex)
        ldi             r30,low(LED7segIndex)
        ld              r16,z
        mov             r26,r16
        ldi             r31,high(LED7segValue)
        ldi             r30,low(LED7segValue)
        add             r30,r16
        clr             r16
        adc             r31,r16
        ld              r27,z
        call    display_7seg

        cpi             r26,0
        brne    timer1_COMP_ISR_CONT
        ldi             r26,4  ;if r16 = 0, reset to 3
timer1_COMP_ISR_CONT:
        dec             r26                     ;else, decrease
        ldi             r31,high(LED7segIndex)
        ldi             r30,low(LED7segIndex)
        st              z,r26
```

```
        pop         r27
        pop         r26
        pop         r16
        reti
```

CHAPER 5 LED MATRIX

5.1 BASIC THEORY

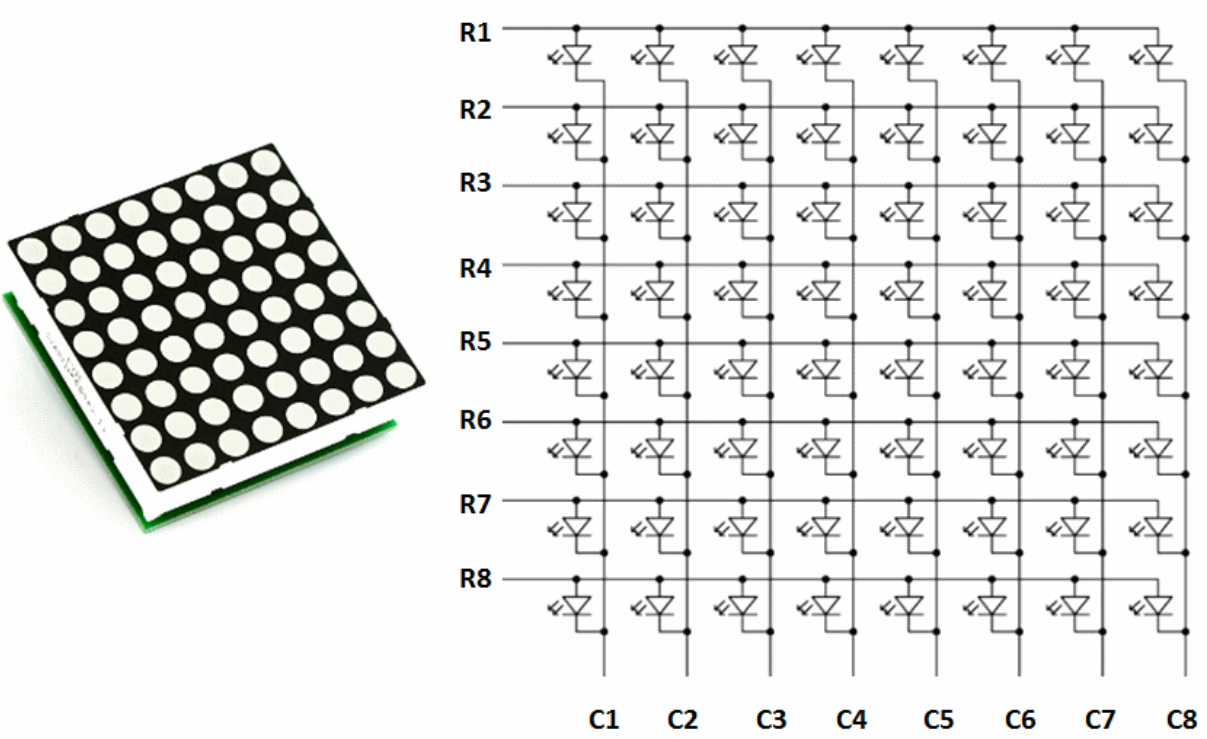


Figure 15: Matrix LED structure

A LED matrix is a collection of LEDs arranged in a grid pattern. In Figure 19, we see an example of an 8x8 single-color LED matrix consisting of 64 LEDs organized into 8 rows and 8 columns. LEDs in the same row share a common anode, while LEDs in the same column share a common cathode. To illuminate an LED at row R and column C, a positive voltage is applied to row R, and a negative voltage (or ground) is applied to column C.

LED matrices are capable of displaying information pixel by pixel, making them suitable for displaying numbers, letters, and various images. LED matrices can be single-color (often red) or multi-color. In multi-color matrices, each pixel can contain two LEDs of different colors (e.g., red and green), allowing for intermediate colors by adjusting the control pulse width of each color (i.e., the on-time duration of each LED). For LED matrices, both the anode and cathode pins of the LEDs are exposed for control.

5.2 HARDWARE DESIGN

The LED matrix on the kit is an 8x8 matrix. The row signals (ROW0-ROW7) are connected to the output of a 74HC595 shift register, and the columns are connected to the outputs of the ULN2803 IC. The design uses the ULN2803 IC to handle currents up to 500mA on the column side. To display a pixel, the data output on the row must be at logic level 1, and VCC is supplied to the anode of the LED. The corresponding column is pulled to GND by setting the appropriate input to the ULN2803 IC to logic 1.

For example, to illuminate the LED at row 0 and column 0, you would set the MATRIX_ROW_0 signal to 1 and the IN0 signal to 1 (causing the MATRIX_COL0 signal to be 0).

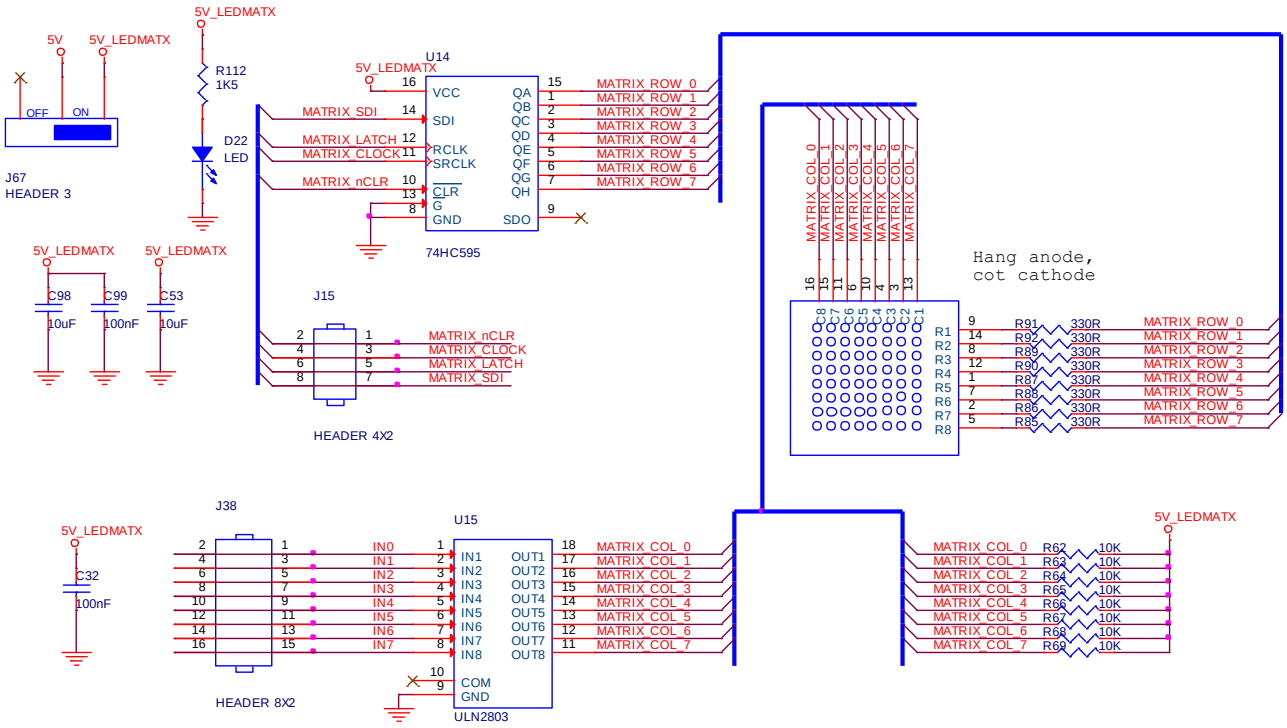


Figure 16: LED Matrix Schematic

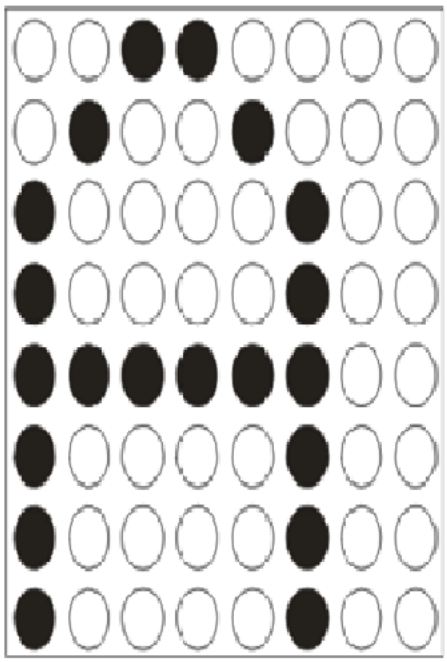
5.3 INTERFACE WITH LED MATRIX

Similar to the 7-segment LED display, displaying on a LED matrix requires sequentially showing data for each column. Each column will be displayed for a specific duration before moving on to the next column. Just like the 7-segment LED display, columns need to be scanned at least 25 times per second, but in practice, a frequency of 50Hz is chosen to ensure that the LEDs do not appear to flicker.

To display a single column, the programmer needs to output one byte to the row (1 byte out of 8 bytes looked up from the font table) to enable that column, wait for a period of time, and then switch to displaying the next column using a similar process. After all 8 columns have been displayed, the display process starts over from the beginning.

The LED matrix being used has 8 columns, with each column having 8 rows. The data required to display each column consists of 8 bits or 1 byte. Therefore, you'll need 8 bytes to store the information you want to display.

For example, let's say you want to display the character 'A' on the LED matrix.



C0	C1	C2	C3	C4	C5	C6	C7
0	0	1	1	0	0	0	0
0	1	0	0	1	0	0	0
1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0
1	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0

Figure 1: 'A' on Led Matrix 8x8

The table describes the values of the 8 bytes needed to display the letter 'A'. This can be referred to as the font for the letter 'A'. These fonts can be stored in program memory as follows:

```
ledmatrix_Font_A:
.DB    0b11111100,  0b00010010,  0b00010001,  0b00010001,  0b00010010,  0b11111100,
0b00000000, 0b00000000
```

Using software tools like "LCD Font Maker" can indeed simplify the process of creating custom fonts or images for LED matrix displays. These tools typically allow users to design fonts or images pixel by pixel, and then generate the corresponding data bytes that can be directly used in microcontroller programs to display the desired content on the LED matrix.

5.3.1 Hardware Connection on the Kit

- **Connect the Column Driver Circuit**

There are several ways to connect the hardware on the kit to facilitate communication with the LED matrix. The simplest method is to connect the J38 header (input of ULN2803) to one of AVR's ports using a bus wire and connect the signals on J15 to other port pins to control the shift register.

Alternatively, you can use the shift register U9 in the 07-SHIFT REGISTER block and connect its output (J35 header) to J38. This way, you can control both rows and columns using two shift registers to save port pins.

- **Connect the Row Driver Circuit**

Choose a 4-bit Port, for example, PB0..PB3, and connect them respectively to the DI, SCLK, SRCLK, and /CLR pins of shift register U4 through J15.

After making these connections, power up the LED matrix block by setting jumper J67 to the ON position.

5.3.2 Programming Display on the LED Matrix

The following program segment displays the character 'A' using the previously created font on the LED matrix, utilizing timer 1 interrupts, with a time interval between two timer overflows of 2500 clock cycles.

The font is stored in ROM and copied to RAM at the address **`LedMatrixBuffer`**.

```
.include "m324padef.inc"  ; Include Atmega324pa definitions
.org 0x0000 ; interrupt vector table
rjmp reset_handler ; reset

.org 0x001A
rjmp timer1_COMP_ISR
reset_handler:
    ; initialize stack pointer
    ldi r16, high(RAMEND)
    out SPH, r16
    ldi r16, low(RAMEND)
    out SPL, r16
       call   shiftregister_initport
       call   shiftregister_cleardata
       call initTimer1CTC
    ; enable global interrupts
    sei
       call   ledmatrix_portinit
main:
       jmp           main

.equ clearSignalPort = PORTB   ; Set clear signal port to PORTB
.equ clearSignalPin = 3        ; Set clear signal pin to pin 3 of PORTB
```

```
.equ shiftClockPort = PORTB    ; Set shift clock port to PORTB
.equ shiftClockPin = 2         ; Set shift clock pin to pin 2 of PORTB
.equ latchPort = PORTB       ; Set latch port to PORTB
.equ latchPin = 1              ; Set latch pin to pin 1 of PORTB
.equ shiftDataPort = PORTB     ; Set shift data port to PORTB
.equ shiftDataPin = 0          ; Set shift data pin to pin 0 of PORTB

; Initialize ports as outputs
shiftregister_initport:
       push   r24
       ldi           r24,
(1<<clearSignalPin)|(1<<shiftClockPin)|(1<<latchPin)|(1<<shiftDataPin);
       out           DDRB, r24        ; Set DDRB to output
       pop           r24
       ret

shiftregister_cleardata:
       cbi clearSignalPort, clearSignalPin       ; Set clear signal pin to low
; Wait for a short time
       sbi clearSignalPort, clearSignalPin       ; Set clear signal pin to high
       ret
; Shift out data
;shift out R27 to bar led
shiftregister_shiftoutdata:
       push   r18
       cbi shiftClockPort, shiftClockPin       ;
       ldi r18, 8               ; Shift 8 bits
shiftloop:
         sbrc r27, 7      ; Check if the MSB of shiftData is 1
         sbi shiftDataPort, shiftDataPin   ; Set shift data pin to high
         sbi shiftClockPort, shiftClockPin   ; Set shift clock pin to high
         lsl r27           ; Shift left
         cbi shiftClockPort, shiftClockPin   ; Set shift clock pin to low
         cbi shiftDataPort, shiftDataPin   ; Set shift data pin to low
         dec r18
         brne shiftloop
; Latch data
       sbi latchPort, latchPin   ; Set latch pin to high
       cbi latchPort, latchPin   ; Set latch pin to low
       pop    r18
       ret

;Lookup table for collumn control
ledmatrix_col_control: .DB  0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01
; Lookup table for font
ledmatrix_Font_A:    .DB           0b11111100, 0b00010010, 0b00010001, 0b00010001,
0b00010010, 0b11111100, 0b00000000, 0b00000000
;             J38 connect to PORTD
; clear signal pin to pin 0 of PORTB
; shift clock pin to pin 1 of PORTB
; latch pin to pin 0 of PORTB
; shift data pin to pin 3 of PORTB
; Output: None

.equ   LEDMATRIXPORT = PORTD
.equ   LEDMATRIXDIR =  DDRD
.dseg
.org   SRAM_START                  ;starting address is 0x100
              LedMatrixBuffer      :      .byte  8
              LedMatrixColIndex    :      .byte  1
.cseg
```

```
.align 2
ledmatrix_portinit:
       push   r20
       push   r21
       ldi            r20, 0b11111111 ; SET port as output
       out            LEDMATRIXDIR, r20

       ldi            r20,0                              ;col index start at 0
       ldi            r31,high(LedMatrixColIndex)
       ldi            r30,low(LedMatrixColIndex)
       st             z,r20
       ldi            r20,0
       ldi            r31,high(ledmatrix_Font_A << 1)    ;Z register point to fontA
value
       ldi            r30,low(ledmatrix_Font_A << 1)
       ldi            r29,high(LedMatrixBuffer)   ;   Y register point to fontA value
       ldi            r28,low(LedMatrixBuffer)
       ldi            r20,8
ledmatrix_portinit_loop:                          ;copy font to display buffer
       lpm            r21,z+
       st             y+,r21
       dec            r20
       cpi            r20,0
       brne   ledmatrix_portinit_loop
       pop            r21
       pop            r20
       ret
; Display a Collumn of Led Matrix
; Input: R27 contains the value to display
;             R26 contain the Col index (3..0)
; Output: None
ledmatrix_display_col:
    push r16 ; Save the temporary register
       push   r27
       clr            r16
       out            LEDMATRIXPORT,r16
       call   shiftregister_shiftoutdata

       ldi            r31,high(ledmatrix_col_control << 1)
       ldi            r30,low(ledmatrix_col_control << 1)
       clr            r16
       add            r30,r26
       adc            r31,r16
       lpm            r27,z
       out            LEDMATRIXPORT,r27
       pop            r27
    pop               r16 ; Restore the temporary register
    ret ; Return from the function

initTimer1CTC:
         push r16
         ldi r16, high(2500)  ; Load the high  yte into the temporary register
         sts OCR1AH, r16       ; Set the high byte of the timer 1 compare value
         ldi r16, low(2500)   ; Load the low byte into the temporary register
         sts OCR1AL, r16       ; Set the low byte of the timer 1 compare value
         ldi r16,  (1 << CS10)| (1<< WGM12) ; Load the value 0b00000101 into the
temporary register
         sts TCCR1B, r16       ;
         ldi r16, (1 << OCIE1A); Load the value 0b00000010 into the temporary
register
         sts TIMSK1, r16       ; Enable the timer 1 compare A interrupt
```

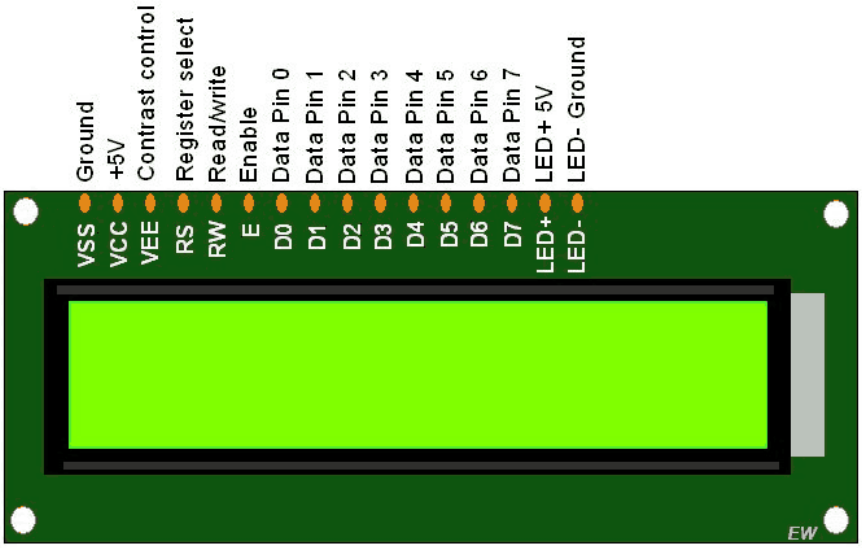
```
        pop  r16
        ret

timer1_COMP_ISR:
       push   r16
       push   r26
       push   r27
       ldi            r31,high(LedMatrixColIndex)
       ldi            r30,low(LedMatrixColIndex)
       ld             r16,z
       mov            r26,r16
       ldi            r31,high(LedMatrixBuffer)
       ldi            r30,low(LedMatrixBuffer)
       add            r30,r16
       clr            r16
       adc            r31,r16
       ld             r27,z
       call   ledmatrix_display_col

       inc            r26
       cpi            r26,8
       brne   timer1_COMP_ISR_CONT
       ldi            r26,0  ;if r26 = 8, reset to 0
timer1_COMP_ISR_CONT:
       ldi            r31,high(LedMatrixColIndex)
       ldi            r30,low(LedMatrixColIndex)
       st             z,r26

       pop            r27
       pop            r26
       pop            r16
       reti
```

CHAPER 6 CHARACTER LCD

6.1 BASIC THEORY

To display information flexibly and efficiently, the system can use an LCD module. There are many types of LCD modules, with the most common being the 2-line 16-character display. LCD modules can be used to display character-based information. Due to the integrated LCD driver, controlling an LCD module is relatively simple.

LCD modules are designed to allow communication with LCDs produced by any manufacturer as long as they use the same HD44780 controller IC. Most LCD modules use a 14-pin interface, including 8 data lines, 3 control lines, and 3 power supply lines. The connections can be arranged as either a single row of 14 pins or two rows of 7 pins.



Pins 1 and 2 are power supply pins, Vss and Vdd. Pin 3, Vee, controls the screen contrast. Pin 4 is the RS line, which controls commands. When RS = 0, data written to the LCD is interpreted as commands, and data read from the LCD is its status. Pin 5 is the R/nW control line, with a low level allowing writing to the LCD and a high level allowing reading from the LCD. Pin 6 is the Enable (E) control line. The remaining pins carry 8-bit data to or from the LCD.

Pin num	Name	Funstion
1	V_{SS}	Ground
2	V_{DD}	'+' pole of power supply
3	V_{EE}	Constrast

4	RS	Register Select
5	R/W	Read/Write
6	E	Enable
7	D0	Bit 0 of data
8	D1	Bit 1 of data
9	D2	Bit 2 of data
10	D3	Bit 3 of data
11	D4	Bit 4 of data
12	D5	Bit 5 of data
13	D6	Bit 6 of data
14	D7	Bit 7 of data

6.2 HARDWARE DESIGN

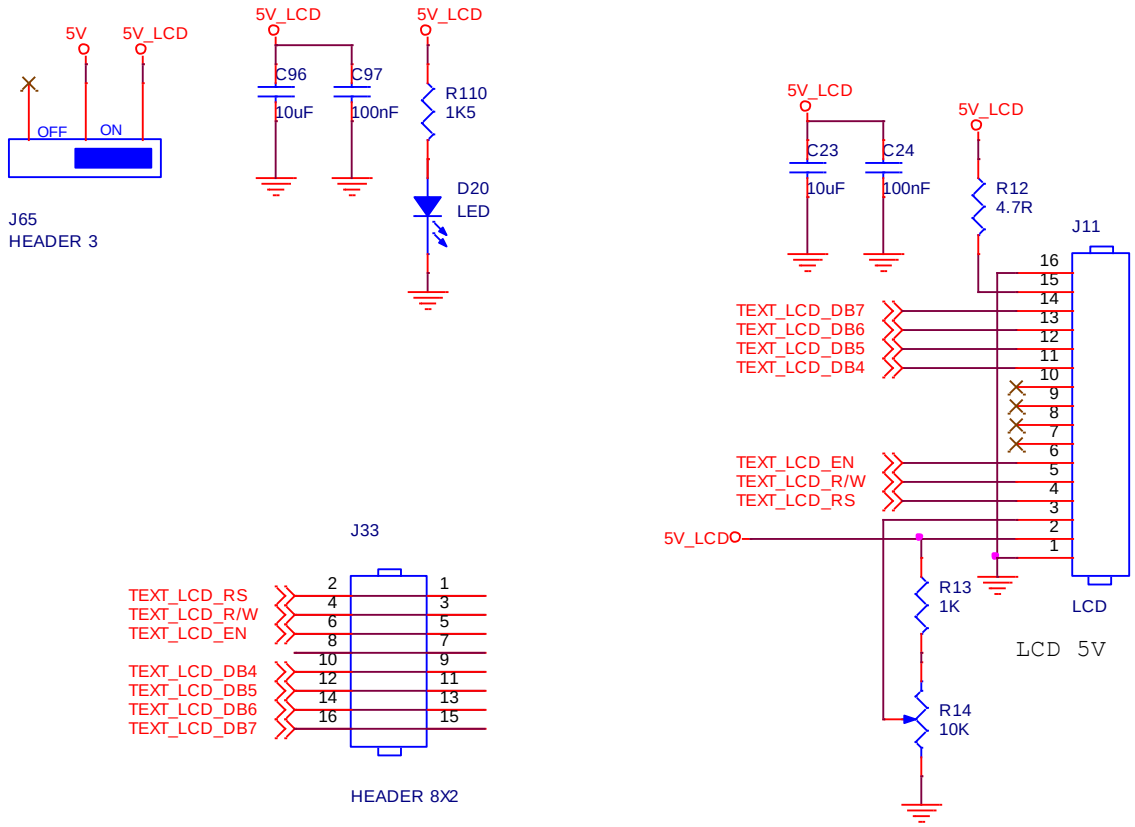


Figure 17: LCD Connection diagram

The LCD is designed for parallel communication with a 4-bit data width. To control the LCD, you can connect the J33 header to one of the AVR's ports using either a bus wire or individual wires, and provide power by setting jumper J65 to the ON position.

6.3 CONTROL THE LCD

Here is a table summarizing the standard control commands for an LCD module:

Command	Description
0F	Display on, cursor blinking
01	Clears the screen and returns the cursor to the initial position.
02	Returns the cursor to the initial position without clearing the screen.
04	Decrement cursor (shift cursor to left)
06	Increment cursor (shift cursor to right)
0E	Display on, cursor blinking
80	Force cursor to the beginning (1st line)
C0	Force cursor to the beginning (2nd line)
38	2 lines and 5×7 matrix
83	Cursor on row 1, position 3
3C	Activate line 2
08	Display off, cursor off
C1	Cursor on row 2, position 1
0C	Display on, cursor off
C2	Cursor on row 2, position 2

Table 3: LCD Commands

To send a command to the LCD, you set the command code on the DATA signals, set RS signal to 0 to select the command register, set RW signal to 0 to select the write mode, and initially set the EN signal to 0. Then, you create a pulse on EN by toggling the EN signal from 0 to 1 and back to 0 to write the command into the command register.

To write data to the LCD for display, you set the data on the DATA signals, set RS signal to 1 to select the data register, set RW signal to 0 to select the write mode, and initially set the EN signal to 0. Then, you create a pulse on EN by toggling the EN signal from 0 to 1 and back to 0 to write the data into the data register.

To check if the LCD is ready to receive a command, you can read the command register. Set RS signal to 0 to select the command register, set RW signal to 1 to select the read mode, and initially set the EN signal to 0. Then, you create a pulse on EN by toggling the EN signal from 0 to 1 and back to 0. At this point, the LCD data will be available on the data pins.

In 4-bit mode (as connected in the experimental kit), data is transferred in 4-bit nibbles, with the high nibble being sent/received first. Therefore, each data write/read process requires 2 EN pulses. The timing diagram below illustrates the LCD write/read process.

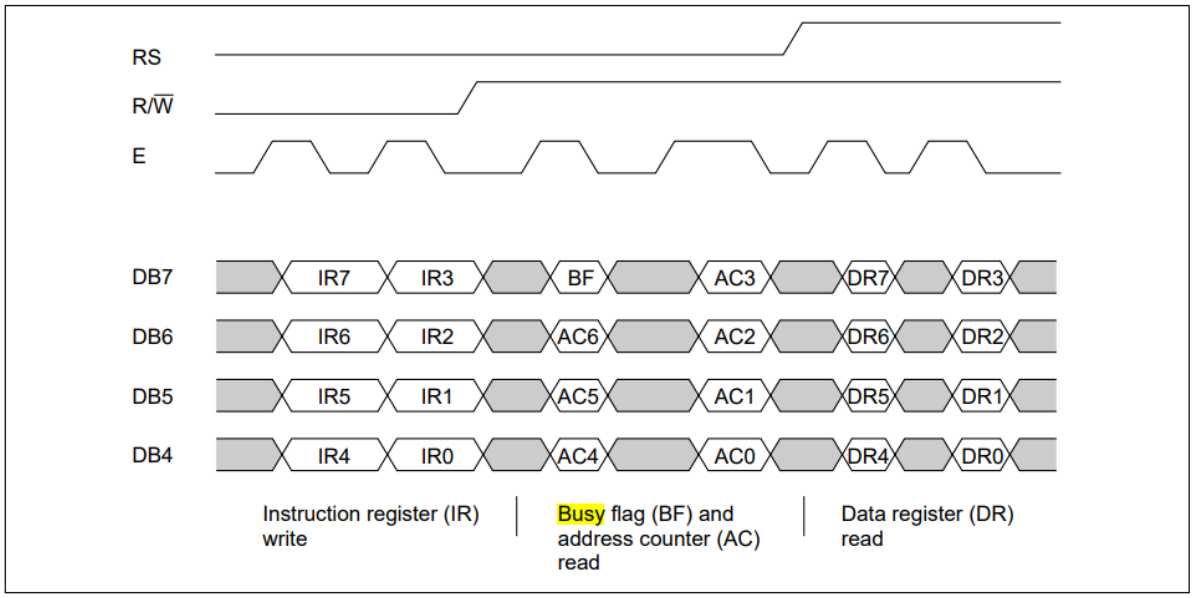


Figure 18: LCD control pulse diagram

The following code snippet performs writing a command to the LCD:

```
; Subroutine to send command to LCD
;Command code in r16
;LCD_D7..LCD_D4 connect to PA7..PA4
;LCD_RS connect to PA0
;LCD_RW connect to PA1
;LCD_EN connect to PA2
LCD_Send_Command:
       push   r17
       call   LCD_wait_busy ; check if LCD is busy
       mov    r17,r16              ;save the command
    ; Set RS low to select command register
    ; Set RW low to write to LCD
       andi   r17,0xF0
    ; Send command to LCD
    out LCDPORT, r17
       nop
       nop
    ; Pulse enable pin
    sbi LCDPORT, LCD_EN
    nop
    nop
    cbi LCDPORT, LCD_EN
    swap      r16
```

```
        andi    r16,0xF0
    ; Send command to LCD
    out LCDPORT, r16
    ; Pulse enable pin
    sbi LCDPORT, LCD_EN
        nop
        nop
        cbi     LCDPORT, LCD_EN
        pop     r17
    ret
```

The following code snippet performs writing a data to the LCD:

```
LCD_Send_Data:
        push    r17
        call    LCD_wait_busy ;check if LCD is busy
        mov             r17,r16             ;save the command
    ; Set RS high to select data register
    ; Set RW low to write to LCD
        andi    r17,0xF0
        ori             r17,0x01
    ; Send data to LCD
    out LCDPORT, r17
        nop
    ; Pulse enable pin
    sbi LCDPORT, LCD_EN
    nop
    cbi LCDPORT, LCD_EN
    ; Delay for command execution
        ;send the lower nibble
        nop
    swap        r16
        andi    r16,0xF0
        ; Set RS high to select data register
    ; Set RW low to write to LCD
        andi    r16,0xF0
        ori             r16,0x01
    ; Send command to LCD
    out LCDPORT, r16
        nop
    ; Pulse enable pin
    sbi LCDPORT, LCD_EN
    nop
    cbi LCDPORT, LCD_EN
        pop     r17
    ret
```

As shown in Figure 23, after each command or data write operation, the LCD requires some time to process the command or data. During this time, the LCD will not accept new commands or data, so when sending data or commands to the LCD, it's important to ensure that the LCD is not in a busy state.

The processing time of the LCD depends on the type of command or operation being executed. The following table illustrates the time required for some typical commands:

Instruction	Code RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs

Figure 19: LCD delay time

The second method is a better approach to ensure higher performance in communicating with the LCD. Instead of waiting for a fixed period of time after each command, you can check the status of the LCD before sending new commands or data.

By checking the seventh bit (BUSY FLAG) in the command register, which is cleared to 0 when the LCD is ready, you can avoid unnecessary delays and ensure that the LCD is ready to accept new instructions or data before proceeding.

```
LCD_wait_busy:
       push   r16
       ldi r16, 0b00000111  ; set PA7-PA4 as input, PA2-PA0 as output
       out LCDPORTDIR, r16
       ldi    r16,0b11110010       ; set RS=0, RW=1 for read the busy flag
       out    LCDPORT, r16
       nop
 LCD_wait_busy_loop:
      sbi LCDPORT, LCD_EN
       nop
       nop
       in     r16, LCDPORTPIN
       cbi    LCDPORT, LCD_EN
       nop
       sbi LCDPORT, LCD_EN
    nop
       nop
    cbi LCDPORT, LCD_EN
       nop
       andi   r16,0x80
```

```
        cpi           r16,0x80
        breq    LCD_wait_busy_loop
        ldi r16, 0b11110111  ; set PA7-PA4 as output, PA2-PA0 as output
    out LCDPORTDIR, r16
        ldi     r16,0b00000000       ; set RS=0, RW=1 for read the busy flag
        out     LCDPORT, r16
        pop     r16
        ret
```

When the power is initially turned on, the LCD needs to wait for a period of approximately 15 ms to stabilize its operation. After that, you can start the initialization process to put the LCD into the appropriate operating mode by sending commands to it. The following subroutine accomplishes the task of initializing an LCD with the mentioned connections:

```
;init the LCD
;LCD_D7..LCD_D4 connect to PA7..PA4
;LCD_RS connect to PA0
;LCD_RW connect to PA1
;LCD_EN connect to PA2

.equ    LCDPORT = PORTA    ; Set signal port reg to PORTA
.equ    LCDPORTDIR = DDRA    ; Set signal port dir reg to PORTA
.equ    LCDPORTPIN = PINA    ; Set clear signal port pin reg to PORTA
.equ    LCD_RS = PINA0
.equ    LCD_RW = PINA1
.equ    LCD_EN = PINA2
.equ    LCD_D7 = PINA7
.equ    LCD_D6 = PINA6
.equ    LCD_D5 = PINA5
.equ    LCD_D4 = PINA4
LCD_Init:
    ; Set up data direction register for Port A
    ldi r16, 0b11110111  ; set PA7-PA4 as outputs, PA2-PA0 as output
    out LCDPORTDIR, r16
    ; Wait for LCD to power up
    call      DELAY_10MS
        call    DELAY_10MS

    ; Send initialization sequence
        ldi r16, 0x02    ; Function Set: 4-bit interface
    call LCD_Send_Command
    ldi r16, 0x28    ; Function Set: enable 5x7 mode for chars
    call LCD_Send_Command
        ldi r16, 0x0E    ; Display Control: Display OFF, Cursor ON
    call LCD_Send_Command
    ldi r16, 0x01    ; Clear Display
    call LCD_Send_Command
    ldi r16, 0x80    ; Clear Display
    call LCD_Send_Command
    ret
```

6.4 DISPLAY CHARACTERS ON AN LCD

To display a character on an LCD, you move the cursor to the desired position and write the ASCII code of the character you want to display on the LCD.

The following example is a main program that initializes the LCD and displays some characters on it:

```
.include "m324padef.inc"  ; Include Atmega324pa definitions
.org 0x0000 ; interrupt vector table
rjmp reset_handler ; reset
;****************************** Program ID ******************************
.org    INT_VECTORS_SIZE
course_name:
.db         "TN Vi Xu Ly-AVR",0
course_time:
.db         "HK2 2022-2023",0


reset_handler:
       call   LCD_Init
; display the first line of information
    ldi     ZH, high(course_name)        ; point to the information that is to be
displayed
    ldi     ZL, low(course_name)
    call    LCD_Send_String
       ldi          r16,1
       ldi          r17,0
       call   LCD_Move_Cursor
       ldi     ZH, high(course_time)        ; point to the information that is to be
displayed
    ldi     ZL, low(course_time)
    call    LCD_Send_String
start:
    rjmp start
```

Example 15: Display characters on LCD

In the example above, the LCD_Move_Cursor function is used, with the source code as shown below:

```
; Function to move the cursor to a specific position on the LCD
; Assumes that the LCD is already initialized
; Input: Row number in R16 (0-based), Column number in R17 (0-based)
LCD_Move_Cursor:

    cpi       r16,0  ;check if first row
       brne   LCD_Move_Cursor_Second
       andi   r17, 0x0F
       ori          r17,0x80
       mov          r16,r17
    ; Send command to LCD
    call LCD_Send_Command
       ret
LCD_Move_Cursor_Second:
        cpi   r16,1  ;check if second row
        brne   LCD_Move_Cursor_Exit      ;else exit
        andi  r17, 0x0F
        ori          r17,0xC0
        mov          r16,r17
    ; Send command to LCD
    call LCD_Send_Command
LCD_Move_Cursor_Exit:
    ; Return from function
    ret
```

Example 16: Moving the Cursor on an LCD

To send a string to an LCD, you can define the string in ROM and send the characters to the LCD one by one until the end is reached (encountering the character '0'). The following code snippet accomplishes this:

```
       ; Subroutine to send string to LCD
       ;address of the string on ZH-ZL
       ;string end with Null
.def   LCDData = r16
LCD_Send_String:
    push    ZH                              ; preserve pointer registers
    push    ZL
       push   LCDData

; fix up the pointers for use with the 'lpm' instruction
    lsl     ZL                              ; shift the pointer one bit left for the
lpm instruction
    rol     ZH
; write the string of characters
LCD_Send_String_01:
    lpm     LCDData, Z+                        ; get a character
    cpi     LCDData,  0                        ; check for end of string
    breq    LCD_Send_String_02          ; done

; arrive here if this is a valid character
    call    LCD_Send_Data          ; display the character
    rjmp    LCD_Send_String_01          ; not done, send another character

; arrive here when all characters in the message have been sent to the LCD module
LCD_Send_String_02:
       pop          LCDData
    pop     ZL                              ; restore pointer registers
    pop     ZH
    ret
```

CHAPER 7 SERIAL PORT

7.1 BASIC THEORY

Serial communication is a critical component in processing systems. For microcontrollers, serial communication is accomplished through the serial port peripheral. This peripheral allows programmers to communicate with external devices using an 8-bit data, 1 stop bit serial communication protocol. The transmission speed can be programmable through software.

To transmit data over long distances, TTL voltage levels from the microcontroller are converted to RS232 or 485 levels using level-shifting circuits. In the experimental kit, the AVR's serial port can communicate with the RS-232 port on a computer through a MAX232 chip, which converts TTL to RS. Communication typically requires only three wires: TXD, RXD, and GND, unless hardware handshaking is used.

7.2 HARDWARE DESIGN

In the experimental kit, the AVR's serial port can communicate with the RS-232 port on a computer through a MAX232 chip, which converts TTL to RS. Communication typically requires only three wires: TXD, RXD, and GND, unless hardware handshaking is used.

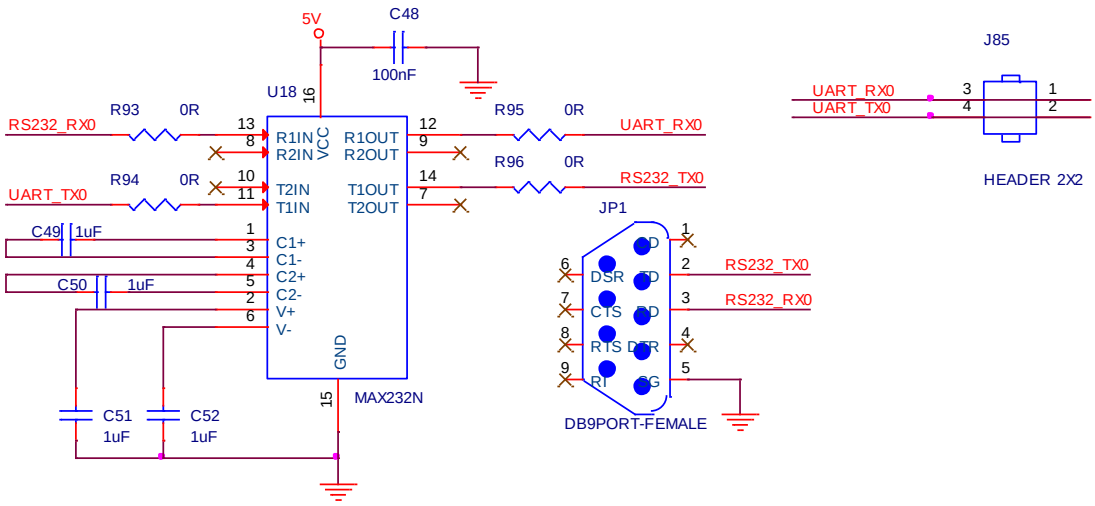


Figure 20: UART-RS232 Interface Block Design

7.3 HARDWARE CONNECTION AND SOFTWARE CONFIGURATION

To begin programming, connect the TX and RX signals from header J85 of the RS232 block to the PD1 (TXD0) and PD0 (RXD0) pins of the microcontroller. Connect a USB-RS232 converter cable to the COM port on the experimental kit. Connect the USB connector of this

cable to the USB port on your computer. Use the Putty software to communicate with the COM port.

1. Open Device Manager on your computer and note the virtual COM port number that the computer recognizes. This will be the COM port for communicating with the microcontroller's UART0.

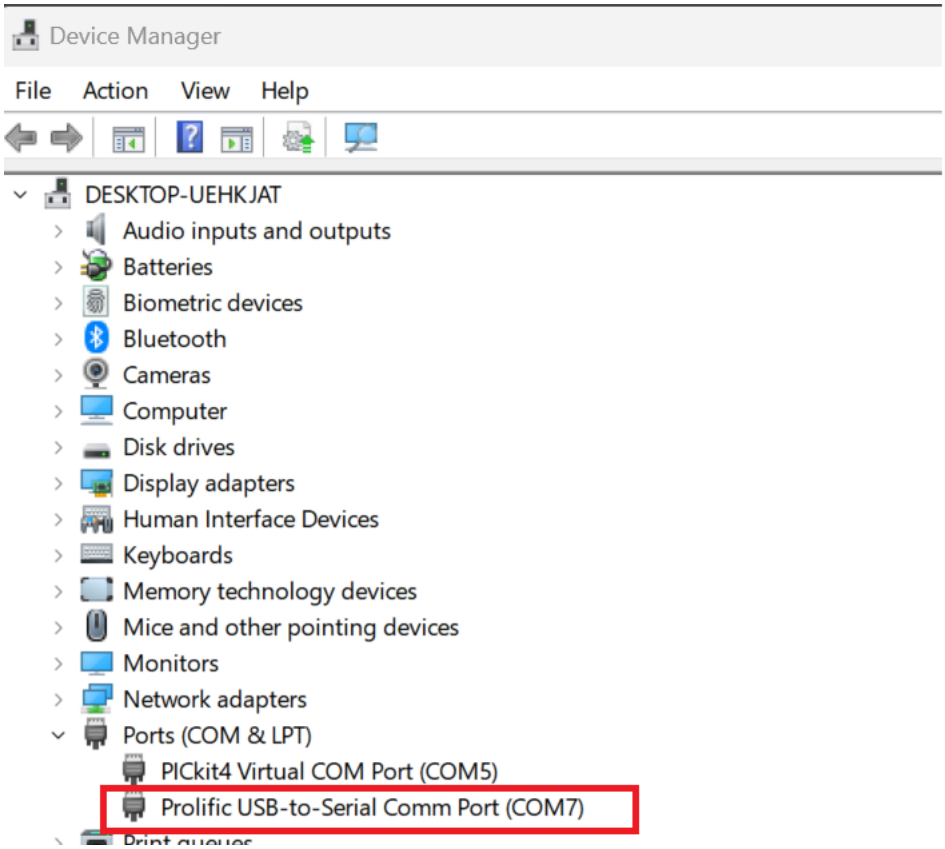


Figure 21: Note the Virtual COM Port on Your Computer

2. Open Putty. Select the Serial tab, enter the correct Serial Line field with the COM port number, then configure the baud rate, data bits, parity bits, and stop bits. Choose Flow control as None:

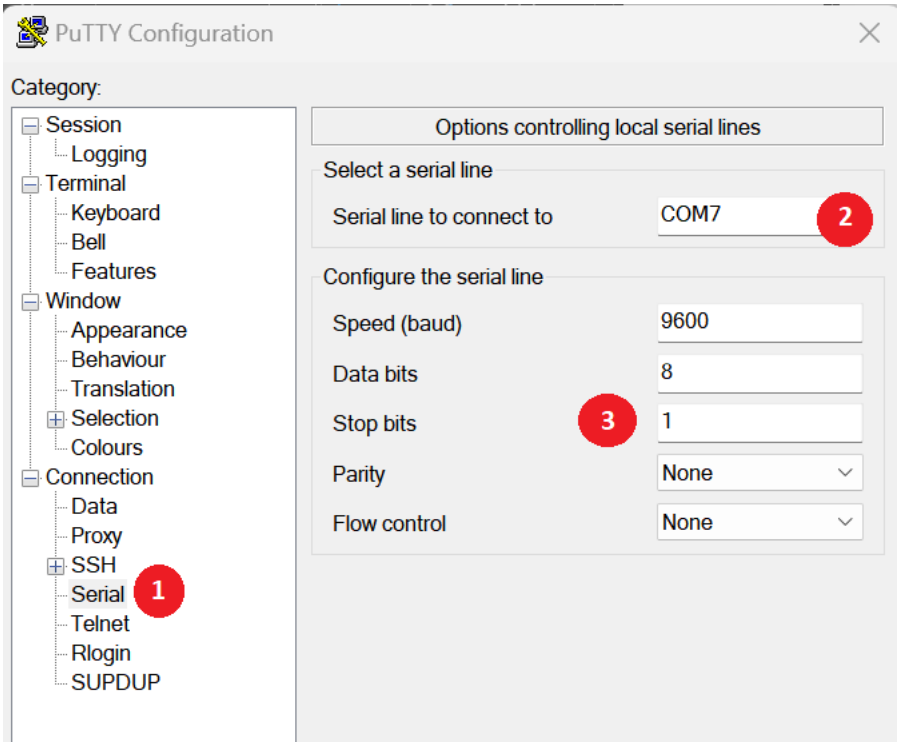


Figure 22: Configure the COM Port

3. Select the Session tab, click Serial, and click Open to open the console.

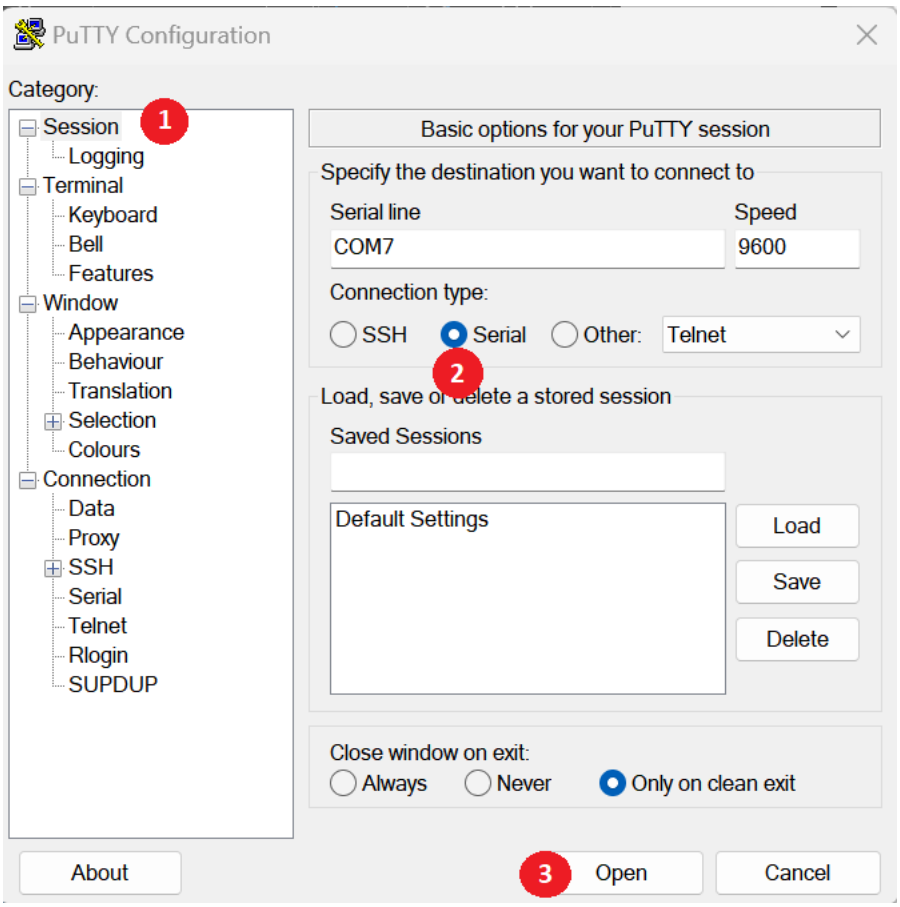


Figure 23: Open the Console

7.4 PROGRAMMING FOR ATMEGA324 UART

Depending on the fuse configuration on the microcontroller, you will have different clock frequencies to calculate the baud rate. By default, the kit uses an external crystal oscillator with a frequency of 8 MHz. You can refer to the datasheet of the Atmega324 for the values to calculate the baud rate and corresponding error.

Figure 28 describes the values to be loaded into the UBRR register corresponding to CPU frequencies of 1 MHz, 1.8432 MHz, and 2 MHz. For a 1 MHz CPU frequency and a baud rate of 9600, to minimize error, you can select double-speed mode and configure the UBRR register with a value of 12.

Baud Rate [bps]	f_{osc} = 1.0000MHz				f_{osc} = 1.8432MHz				f_{osc} = 2.0000MHz			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error
2400	25	0.2%	51	0.2%	47	0.0%	95	0.0%	51	0.2%	103	0.2%
4800	12	0.2%	25	0.2%	23	0.0%	47	0.0%	25	0.2%	51	0.2%
9600	6	-7.0%	12	0.2%	11	0.0%	23	0.0%	12	0.2%	25	0.2%
14.4k	3	8.5%	8	-3.5%	7	0.0%	15	0.0%	8	-3.5%	16	2.1%
19.2k	2	8.5%	6	-7.0%	5	0.0%	11	0.0%	6	-7.0%	12	0.2%
28.8k	1	8.5%	3	8.5%	3	0.0%	7	0.0%	3	8.5%	8	-3.5%
38.4k	1	-18.6%	2	8.5%	2	0.0%	5	0.0%	2	8.5%	6	-7.0%
57.6k	0	8.5%	1	8.5%	1	0.0%	3	0.0%	1	8.5%	3	8.5%
76.8k	–	–	1	-18.6%	1	-25.0%	2	0.0%	1	-18.6%	2	8.5%
115.2k	–	–	0	8.5%	0	0.0%	1	0.0%	0	8.5%	1	8.5%
230.4k	–	–	–	–	–	–	0	0.0%	–	–	–	–
250k	–	–	–	–	–	–	–	–	–	–	0	0.0%
Max.(1)	62.5kbps		125kbps		115.2kbps		230.4kbps		125kbps		250kbps	

Figure 24: UBRRn Values Table for Baud Rate Configuration

Example 17 describes the process of configuring UART0 at a baud rate of 9600 and waiting to receive a byte on UART0, then transmitting it back on UART0.

```
.include "m324padef.inc"
; Replace with your application code
    call      USART_Init
start:
    call      USART_ReceiveChar
    call      USART_SendChar
    rjmp start
;init UART 0
;CPU clock is 1Mhz
USART_Init:
    ; Set baud rate to 9600 bps with 1 MHz clock
    ldi r16, 12
    sts UBRR0L, r16
       ;set double speed
```

```
    ldi r16, (1 << U2X0)
    sts UCSR0A, r16
    ; Set frame format: 8 data bits, no parity, 1 stop bit
    ldi r16, (1 << UCSZ01) | (1 << UCSZ00)
    sts UCSR0C, r16
    ; Enable transmitter and receiver
    ldi r16, (1 << RXEN0) | (1 << TXEN0)
    sts UCSR0B, r16
    ret
;send out 1 byte in r16
USART_SendChar:
       push   r17
    ; Wait for the transmitter to be ready
    USART_SendChar_Wait:
        lds   r17,   UCSR0A
        sbrs r17, UDRE0     ;check USART Data Register Empty bit
        rjmp USART_SendChar_Wait
       sts UDR0, r16        ;send out
       pop    r17
    ret
;receive 1 byte in r16
USART_ReceiveChar:
       push   r17
    ; Wait for the transmitter to be ready
    USART_ReceiveChar_Wait:
        lds   r17,   UCSR0A
        sbrs r17, RXC0      ;check USART Receive Complete bit
        rjmp USART_ReceiveChar_Wait
        lds r16, UDR0              ;get data
        pop   r17

    ret
```

Example 17: Configuring UART0 and Transmitting Data

CHAPER 8 EEPROM AND SDCARD

8.1 HARDWARE DESIGN FOR EEPROM BLOCK

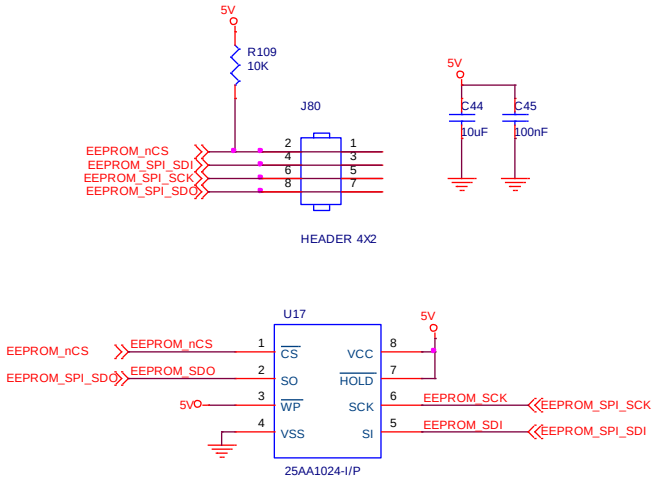


Figure 25: illustrates the hardware design for the EEPROM block

The 25AA1024 memory chip is a 1 Mbit EEPROM chip that communicates using the SPI standard. In designs where it is necessary to retain information without data loss during power outages, such as configuration settings or passwords, chips of this type are commonly used.

Name	Pin Number	Function
$\overline{CS}$	1	Chip Select Input
SO	2	Serial Data Output
$\overline{WP}$	3	Write-Protect Pin
Vss	4	Ground
SI	5	Serial Data Input
SCK	6	Serial Clock Input
$\overline{HOLD}$	7	Hold Input
Vcc	8	Supply Voltage

Table 4 describes the signal pins of the EEPROM

The control signals of the memory chip are connected to header J80. These signals can be connected to the microcontroller's port pins to perform read/write operations on the chip.

In the design, the /WP and /HOLD signals are tied to VCC, making them inactive.

8.2 HARDWARE CONNECTION AND PROGRAMMING

The ATmega32 has on-chip SPI hardware, capable of operating in both master and slave modes. The EEPROM chip operates as an SPI Slave device. To communicate with it, you will configure the AVR microcontroller in master mode.

Connect the AVR pins and control signals on header J80. The SPI signals include MOSI, MISO, SCK, and SS.

32-pin TQFP/ QFN/ MLF Pin #	40-pin PDIP Pin #	DRQFN Pin#	VFBGA Pin#	PAD	EXTINT	PCINT	ADC/AC	OSC	T/C # 0	T/C # 1	USART	I2C	SPI
1	6	A1	B2	PB[5]		PCINT13							MOSI
2	7	B1	B1	PB[6]		PCINT14							MISO
3	8	A2	C3	PB[7]		PCINT15							SCK

For the Slave Select (SS) signal, if you are working in SPI Slave mode, on the ATMEGA324, it is connected to pin PB4. In master mode, you can use any port pin and control it through software.

If multiple SPI Slave chips are connected to the same SPI bus, they will share the MOSI, MISO, and SCK signals. The SS signals of the slaves will be connected to AVR's I/O port pins.

The connections of the EEPROM signals to the SPI signals are as follows:

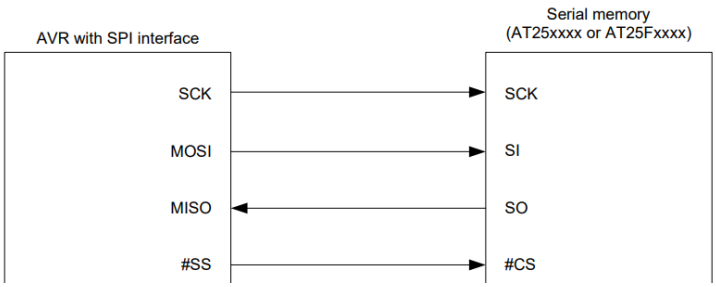


Figure 26: SPI Signal Connections between AVR and EEPROM

For detailed read/write commands and timing diagrams, refer to the 25AA1024 datasheet.

8.3 HARDWARE DESIGN FOR SDCARD BLOCK

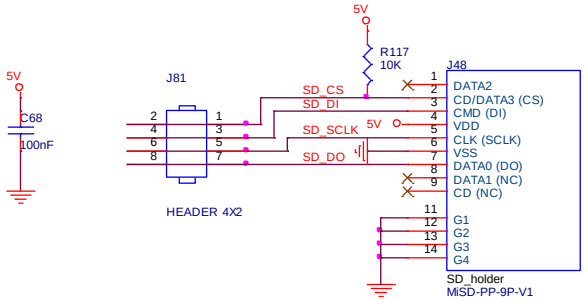


Figure 27: Illustrates the hardware design for the SDCARD block

To store large-sized data such as images, audio, or long-term data, SDCARDs are commonly used.

SDCARDS have two control interfaces: SDIO and SPI. With the SDIO interface, data is transmitted and received in 4-bit chunks, allowing for high-speed communication. The SPI interface transmits data sequentially, resulting in a slower data transfer rate. However, most microcontrollers only support SPI and lack SDIO support, so SPI is commonly used to communicate with SDCARDs.

8.4 HARDWARE CONNECTION AND PROGRAMMING

Similar to the SPI EEPROM connection, the SDCARD signals are connected to the SPI signals as follows. There are no external pull-up resistors for the SDCARD's DO pin in the design. You need to enable the internal pull-up resistor for the MISO pin of the AVR.

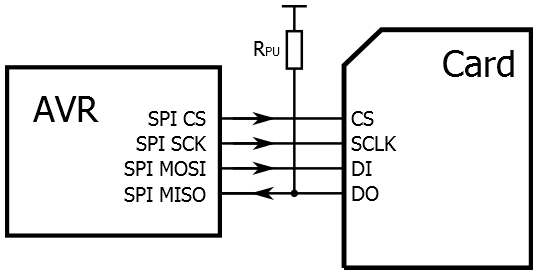


Figure 28: SPI Signal Connections between AVR and SDCARD

CHAPER 9 TOUCH SCREEN LCD

9.1 HARDWARE DESIGN FOR LCD BLOCK

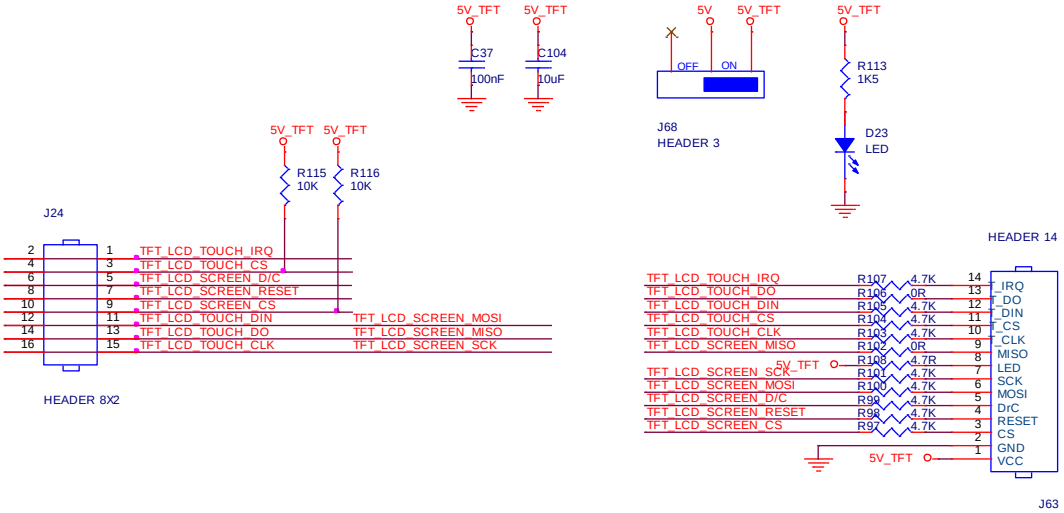


Figure 29: Hardware design for the Graphic LCD block

To display graphic interfaces and use a touchscreen, the experimental kit integrates a 3.2-inch TFT LCD screen. This LCD screen has an ILI9341 controller chip and an XPT2046 touch screen controller.

Both the LCD and touch screen communicate via SPI. Header J63 in Figure 39 is a connector used to connect the signals on the LCD touch module.

Signals from the LCD module are connected to header J24, from which they can be connected to the microcontroller.

The SCK, MOSI, and MISO signals of the ILI9341 and XPT2046 chips are shared at header J24. The chip select signals for these two chips are controlled separately.

In addition to the mentioned signals, the LCD also has a command/data select signal (D/C) and a reset signal. For the touch screen, there is a TOUCH_IRQ signal that goes active when a touch event occurs. These signals are connected to J24.

9.2 HARDWARE CONNECTION AND PROGRAMMING

You will connect the MOSI, MISO, and SCK signals from the AVR to their corresponding signals on J24.

The chip select signals, LCD_D/C, LCD_RESET, will be connected to port pins. The TOUCH_IRQ signal will be connected to an external interrupt pin.

For interaction with the LCD and touch screen, please refer to the documentation for detailed information: http://www.lcdwiki.com/3.2inch_SPI_Module_ILI9341_SKU:MSP3218

CHAPER 10 REAL-TIME CLOCK (RTC)

10.1 HARDWARE DESIGN FOR RTC BLOCK

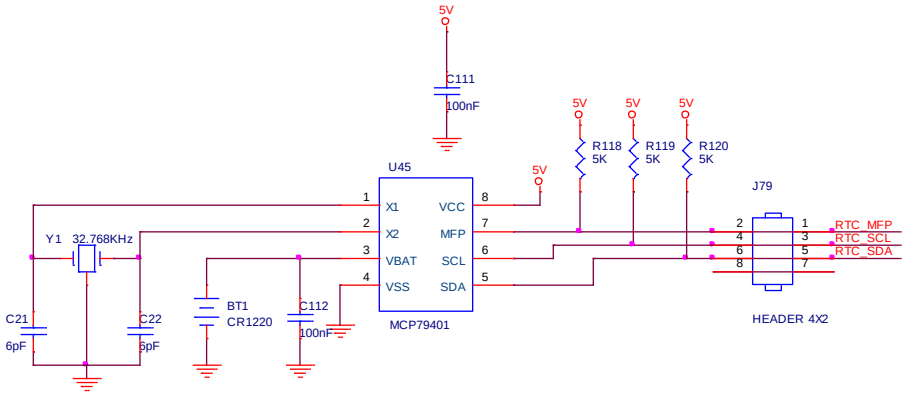


Figure 30: hardware design for the RTC (Real-Time Clock) block

MCP79401 chip is a real-time clock circuit from Microchip, featuring a real-time clock/calendar, automatic date, time, and year updating, leap year compensation, and more. In designs requiring real-time capabilities, chips like this one are often used instead of relying on the CPU to count time.

The MCP79401 is an I2C Slave device and communicates with the AVR microcontroller via the I2C bus. Additionally, the MFP (Multipurpose Pin) signal can be programmed to signal an event to the microcontroller.

The control signals of the RTC are connected to header J79.

10.2 HARDWARE CONNECTION AND PROGRAMMING

TWI (Two-Wire Serial Interface) is a synchronous serial communication module on AVR chips based on the I2C communication standard. On the ATMEGA324, the TWI functional pins are connected to port pins as follows:

PC[0]		PCINT16						SCL		
PC[1]		PCINT17						SDA		

To communicate with the MCP79401 IC, connect pin PC0 to the RTC_SCL signal and pin PC1 to RTC_SDA.

You can connect the RTC_MFP pin to an external interrupt pin or a port pin that supports PINCHANGE interrupts. Then, program this pin to trigger the AVR when an event occurs.

In case you want to continuously update the time from the RTC in an application like a clock, you can program the MFP pin to output a 1Hz frequency signal, and use this signal to trigger

an interrupt. This way, every time the interrupt is triggered, you can update the data from the RTC without having to poll continuously.

For details about the registers and addresses of the MCP79401 chip, refer to the MCP79401 datasheet.

CHAPER 11 ADC COMMUNICATION AND ANALOG SENSING

11.1 ADC COMMUNICATION DESIGN

11.1.1 OVERVIEW OF ADC ON ATMEGA324PA

The ATMEGA324PA has a 10-bit ADC with 8 single-ended input channels ADC7..ADC0.
These single-ended channels can be used to form 16 differential input channels.
After each ADC conversion, the result is stored in the ADCH-ADCL register pair.
For single-ended ADC conversion, the ADC reading result is calculated as follows:

$$\boldsymbol{ADCVal} = \frac{\boldsymbol{V_{IN} * 1024}}{\boldsymbol{V_{REF}}} \quad (1)$$

With differential ADC conversion, the ADC reading will be:

$$\boldsymbol{ADCVal} = \frac{(\boldsymbol{V_{POS} - V_{NEG}}) * \boldsymbol{GAIN} * 512}{\boldsymbol{V_{REF}}} \quad (2)$$

For differential ADC conversions, the ADC result is in two's complement format, ranging from 0x200 (-512) to 0x1FF (+511).
The reference voltage V_{REF} can be provided externally by applying a voltage to the AV_{REF}, pin or can be selected from the AV_{CC} voltage or the internal 2.56V reference voltage. It's important to note that if an external voltage is connected to the AV_{REF}, you cannot select the AV_{CC} or the internal reference voltage because it would short-circuit the external and internal references, potentially damaging the chip..
Another important note is that in differential ADC mode, the AV_{REF} voltage must be between 2V (minimum) and AV_{CC} - 0.5V (maximum).
Please refer to the Atmega324pa datasheet, specifically the ADC - Analog to Digital Converter chapter and the Electrical Characteristics chapter under ADC characteristics, to fully understand the features and parameters of the ADC.

11.1.2 HARDWARE DESIGN

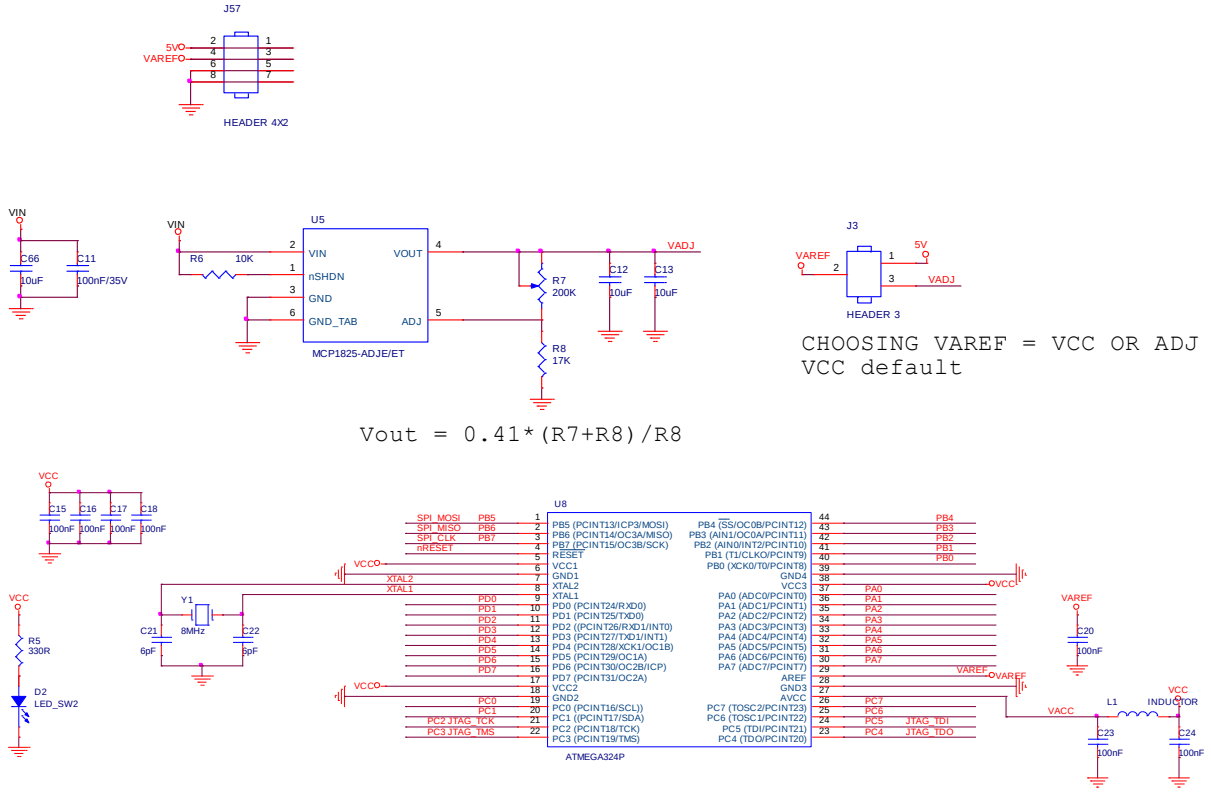


Figure 31: Hardware design for the CPU and reference voltage supply for the ADC

The ATMEGA324 microcontroller is powered by 5V through the V_{CC} pins. The 5V supply is filtered using a PI filter and supplied to the AV_{CC} pin. Therefore, AV_{CC} is also powered by 5V.

The MCP1825-ADJ voltage regulator is used to generate a variable voltage according to the formula.

$$V_{OUT} = \frac{(R_7 + R_8) * 0.41}{R_8}$$

Header J3 is used to select the reference voltage for the ADC. Depending on the jumper configuration, the reference voltage can be either V_{CC} (5V) or a voltage from an external source. Before selecting an external reference voltage, it's essential to ensure that this voltage does not exceed 5V in the case of single-ended ADC or 4.5V in the case of differential ADC. If you are using the internal reference voltage (2.56V) or AV_{CC}, you must remove this jumper to ensure that no external voltage is applied to the AV_{REF} pin. Failure to do so could potentially damage the microcontroller.

11.2 RHEOSTAT

11.2.1 HARDWARE DESIGN

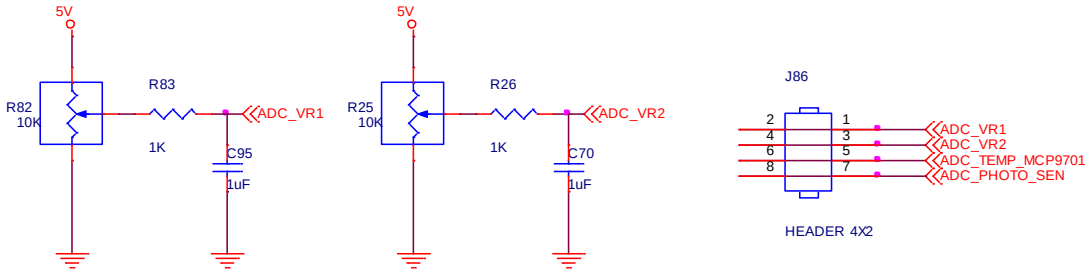


Figure 32: Rheostat Block

The experimental kit includes two potentiometers, VR1 and VR2, which are used to generate variable voltages. The voltage outputs are connected to header J86..

11.2.2 HARDWARE CONNECTION AND PROGRAMMING

11.2.2.1 PROGRAMMING ADC IN SINGLE-ENDED MODE

Single-ended ADC mode is used to measure the voltage between a signal and ground. For experimentation, you can apply voltage from the voltage dividers created by VR1 and VR2 to the ADC input pins for programming. Connect the ADC_VR1 or ADC_VR2 signal to the AVR's ADC7..ADC0 port pins using individual wires.

To measure the voltage value, read the value from the ADC and calculate the input voltage using the formula:

$$V_{IN} = \frac{ADCVal * V_{REF}}{1024} \quad (3)$$

In the formula, ADCVal represents the value read from the ADC, which is an unsigned value ranging from 0x00 to 0x3FF in 10-bit ADC mode.

The value $\frac{V_{REF}}{1024}$ is the value of one LSB, which is the range of input voltage change that causes the ADC value to increase by 1 bit. This is the resolution of the measurement. For example, with V_{REF} = 5V, the resolution of a 10-bit ADC would be: 1 LSB = $\frac{V_{REF}}{1024}$ = 0.0048828125 (V). In measurements, we often want to display with resolutions like 1, 0.1, 0.01... or multiples of such values, for example, 0.2, 0.02... In such cases, the reference voltages chosen are typically powers of 2, such as 1.024V, 2.56V, 2.048V, 4.096V.

For example, if we choose a reference voltage of 2.56V, it allows us to measure voltages above it, and the resolution of the measurement would be: $1LSB = \frac{2.56}{1024} = 0.0025\,V$. This value is convenient for calculations and display.

To perform calculations using assembly language, we need libraries and instructions for 8-bit and 16-bit multiplication. These libraries are described in the document 'Atmel AVR201: Using the AVR Hardware Multiplier.' The source code is contained in the avr201.asm file.

In the formula $\boldsymbol{V_{IN} = \frac{ADCVal * V_{REF}}{1024}}$, after multiplication, we also need to perform a division by 1024. This division is simple and involves shifting the number being divided to the right by 10 bits. This is integer division, resulting in an integer quotient, ignoring the remainder, with no decimal part.

To display the result with decimal precision, we choose the number of decimal places and multiply the result of ADCVal * V_{REF} by a power of 10 before performing the division.

For example, if we want to display 4 decimal places, we must calculate the display value using the formula:

$$\boldsymbol{V_{IN} * 10000 = \frac{10000 * ADCVal * V_{REF}}{1024}}$$

And display this number with 4 decimal places after the decimal point.

If we use V_{REF} = 2.56V, the formula becomes:

$$\boldsymbol{V_{IN} * 10000 = \frac{10000 * ADCVal * 2.56}{1024} = \frac{100 * ADCVal * 256}{1024} = 25 * ADCVal.}$$

To easily verify the result, we can connect this voltage signal to the TEST STATION block and use a VOM (Volt-Ohm Meter) or oscilloscope to measure the actual value.

The following program segment performs the initialization and reads ADC channel 0

```
.include "m324padef.inc"  ; Include Atmega324pa definitions

call   init_adc
start:
    call      read_adc_16bit
    rjmp start
;init the ADC with reference voltage to AVCC,  select ADC channel 0
;set ADC clock to 125Khz with CPU clock 1Mhz
init_adc:
    ldi   r16, (1<<REFS0)    ; set reference voltage to AVCC, ADC channel 0
    sts   ADMUX, r16         ; store the value in ADMUX register
    ldi   r16, (1<<ADEN) | (1<<ADPS1)|(1<<ADPS0) ; set ADC prescaler to 8, enable
ADC
    sts   ADCSRA, r16        ; write to ADCSRA register
       nop
    ret

;start a single conversion
;read ADC and store H-L byte to r17-r16
read_adc_16bit:
  push r18

  lds  r18, ADCSRA
  ori  r18, (1<<ADSC)
  sts   ADCSRA, r18
```

```
read_adc_16bit_wait:
      lds           r18,ADCSRA
      andi    r18, (1<<ADSC)
   cpi              r18, (1<<ADSC)
      breq    read_adc_16bit_wait
 lds    r16, ADCL              ; read ADCL first
 lds    r17, ADCH              ; read ADCH second

 pop  r18
 ret
```

Example 18: ADC communication in single end mode

11.2.2.2 *PROGRAMMING ADC IN DIFFERENTIAL MODE*

In differential mode, the ADC is used to measure the voltage between two signals. Voltage signals are applied to pairs of inputs, P (positive) and N (negative), and the voltage difference is amplified and fed into the ADC. The pairs of signals and gain are selected by configuring the ACDMUX register, with the parameters provided in Table 25-4, page 319, of the document: "Atmel-42743B-ATmega324P/V_Datasheet_Complete-08/2016."

Please note that in differential mode, the reference voltage must be greater than 2V and less than AVCC – 0.5. Therefore, you must use the on-chip 2.56V reference voltage or provide an external reference voltage and connect it to AVREF accordingly.

It's important to note that in differential mode, the result will be a signed number in two's complement form.

$$\boldsymbol{ADCVal} = \frac{(V_{POS} - V_{NEG}) * GAIN * 512}{V_{REF}}$$

The following program segment measures the voltage difference between two ADC channels, ADC1 and ADC0:

```
.include "m324padef.inc"  ; Include Atmega324pa definitions

call   init_adc
start:
    call      read_adc_16bit
    rjmp start
init_adc:
    ldi   r16, (1<<REFS0)|(1<<REFS1)  | 0b00001001   ; set reference voltage to
2.56, , ADC1 (P), ADC0 (N) GAIN=10
    sts   ADMUX, r16          ; store the value in ADMUX register
    ldi   r16, (1<<ADEN) | (1<<ADPS1)|(1<<ADPS0) ; set ADC prescaler to 8, enable
ADC
    sts   ADCSRA, r16         ; write to ADCSRA register
    nop
    ret
;start a single conversion
;read ADC and store H-L byte to r17-r16
read_adc_16bit:
```

```
  push r18

  lds  r18, ADCSRA
  ori  r18, (1<<ADSC)
  sts   ADCSRA, r18

read_adc_16bit_wait:
      lds           r18,ADCSRA
      andi   r18, (1<<ADSC)
      cpi           r18, (1<<ADSC)
      breq   read_adc_16bit_wait
  lds   r16, ADCL             ; read ADCL first
  lds   r17, ADCH             ; read ADCH second

  pop  r18
  ret
```

11.3 TEMPERATURE SENSOR MCP9701

11.3.1 HARDWARE DESIGN

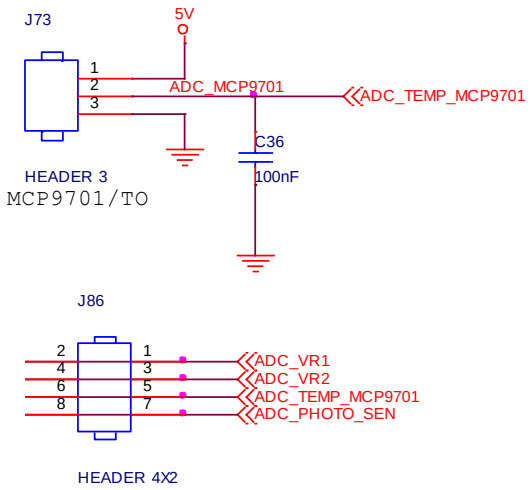


Figure 33: Block diagram of the analog temperature sensor

The MCP9701 temperature sensor from Microchip is a voltage output sensor. This output is connected to header J86.

11.3.2 HARDWARE CONNECTION AND PROGRAMMING

To measure temperature, you connect the ADC_TEMP_ADC9701 signal on header J86 to an analog input channel of the microcontroller and use the ADC to measure this voltage, thus calculating the temperature.

To easily verify the results, you can connect this voltage signal to the TEST STATION block and use a VOM or oscilloscope to measure the actual value.

The output of MCP9701 is calculated using the following formula:

Vout = Tc * Ta + V0c

- Ta: Ambient temperature
- Tc: temperature coefficient, for MCP9701, Tc = 19.53 mV. So, for every 1 degree Celsius change, the output increases by 19.53 mV.
- V0c: voltage at 0 degrees Celsius. For MCP9701, V0c = 400 mV.

Please note that the value of 400 mV is a typical value. In practical applications, you would need to determine the precise value through calibration, often done by immersing the sensor in melting ice and measuring the output voltage.

11.4 LIGHT SENSOR

11.4.1 HARDWARE DESIGN

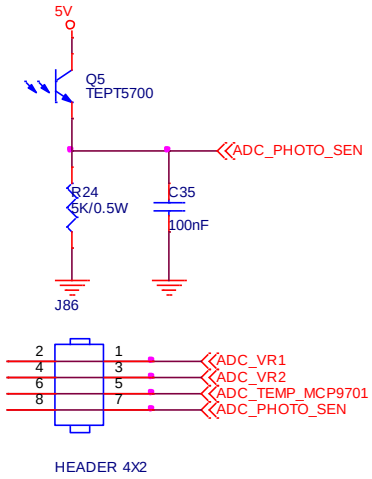


Figure 34: Block diagram of the light sensor

To measure light intensity, we use a photo transistor TEPT5700. This sensor has a current output IC that depends on the received light. This current is passed through resistor R24 to convert it into voltage. This voltage signal is connected to header J86.

11.4.2 HARDWARE CONNECTION AND PROGRAMMING

With R24 = 5Kohm, the output voltage of the sensor will be:

Vout = 5K * Ipce **[1]**

When the light intensity increases, the current Icpe also increases, causing the output voltage to gradually rise to 5V. As shown in Figure 33, when the brightness is 900 lux, the output current is 800 µA. At that point, Vout = 4V.

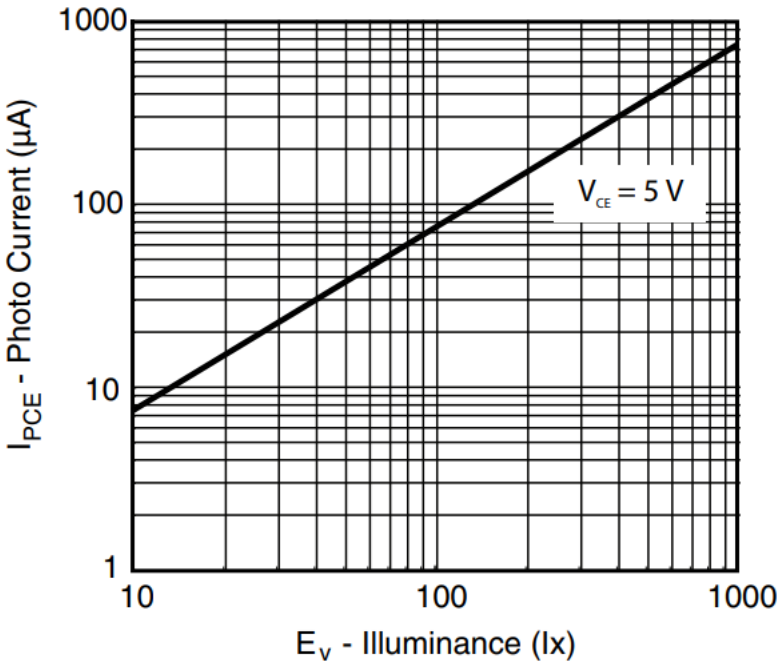


Figure 35: Output characteristics of TEPT5700

When the light intensity increases further, Vce of TEPT5700 decreases to Vceo (saturated voltage), and the sensor goes into a saturated state, no longer functioning correctly.

To convert the measured voltage value to light intensity, we use the characteristic curve in Figure 33. This is a log-log characteristic, and when represented by a linear line, it means that the relationship between the vertical and horizontal axes will have the form:

$$n \log I_{CPE} + \log k = \log E_v \quad [2]$$

From the characteristic curve above, we have the following reference points:

$E_v = 10$, $I_{CPE} = 7$

$E_v = 900$, $I_{CPE} = 700$

To measure light, we connect the ADC_PHOTO_SEN signal on header J86 to an analog input channel of the microcontroller and use the ADC to measure this voltage, thereby calculating the light intensity.

To easily verify the results, you can connect this voltage signal to the TEST STATION block and use a VOM or oscilloscope to measure the actual value.

Since the correlation between the variables is an exponential function, it's challenging to implement calculations in assembly language. Even with C libraries, due to hardware limitations, computational functions can be slow.

To calculate light intensity from the ADC value, you can use a lookup table method. With a 10-bit ADC, you will create a table of 1024 light intensities corresponding to ADC readings

from 0 to 1023. For light intensity in the range of 0-1000 lux, each value in the table will require 2 bytes, so you'll need 2KB for this table.

11.5 TEMPERATURE SENSOR DS18B20

DS18S20 is a 9-bit digital temperature sensor with programmable high and low temperature alarms. It communicates over a 1-Wire bus, which means it requires only one port pin for communication between the sensor and the MCU. The temperature range of the sensor is from -55°C to +125°C with a maximum accuracy of 0.5°C. Additionally, the DS18S20 can be powered directly through the data line.

Each DS18S20 contains a unique 64-bit serial code, allowing multiple sensors to operate on the same 1-Wire bus. The DS18S20 has three pins: GND, DQ (data), and VDD (power), where DQ is the data input and output.

The 1-Wire bus protocol is defined by Dallas Semiconductor. The control line needs a pull-up resistor because all devices on the bus are connected through a 3-state or open-drain port (DQ pin). In this bus system, the MCU (master) identifies and addresses devices on the bus using their 64-bit address code. Since each device has a unique 64-bit identification code, there is virtually no limit to the number of devices that can be connected to the bus.

11.5.1 HARDWARE DESIGN

On the experimental kit, J72 is designed to connect to the DS18B20 sensor, and the signal output is connected to J74.

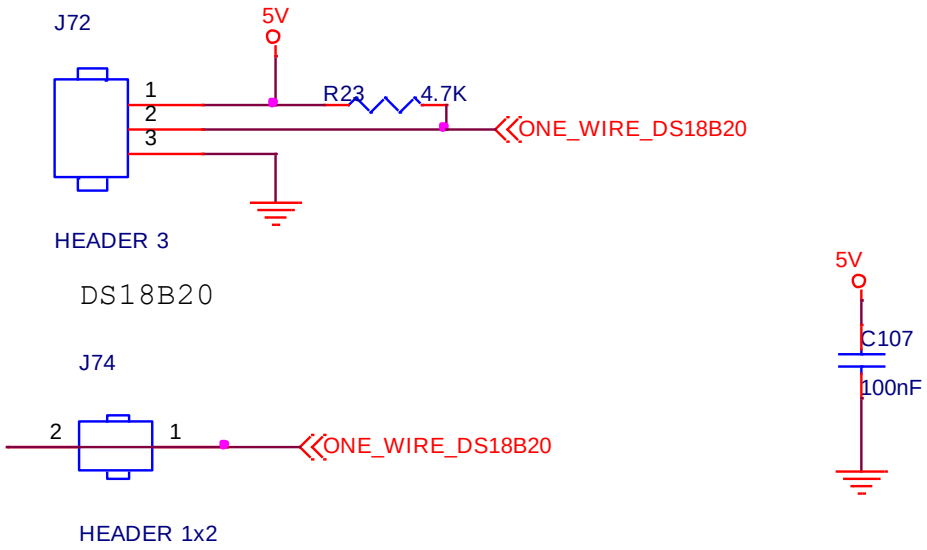


Figure 36: Block diagram of the DS18B20 design

11.5.2 CONNECTION AND COMMUNICATION PROGRAMMING

To communicate with the DS18B20 sensor, you should connect the DS18B20 sensor to J12 and connect J74 to one of the port pins of the microcontroller.

For reading values from the DS18B20 sensor, students can refer to the datasheet of this sensor. The methods of communication between AVR microcontrollers and this sensor can be found in the document titled "AVR318: Dallas 1-Wire Master on tinyAVR and megaAVR."

CHAPER 12 DC MOTORS

12.1 OVERVIEW AND THEORY

DC motors are commonly encountered devices in control systems. They are controlled by applying a direct current (DC) to the stator windings. The direction of rotation and speed are manipulated by changing the direction and magnitude of the current. Additionally, DC motors can be torque-controlled when dealing with varying loads.

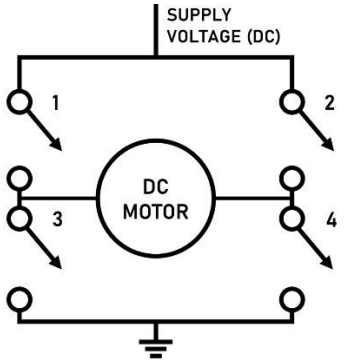


Figure 37: H-Bridge

The most common method of control involves using an H-bridge. The switches in the H-bridge open and close to change the direction of current flow through the motor, thus altering its rotation. If switches 1 and 4 are closed, the motor rotates clockwise; if switches 2 and 3 are closed, it rotates counterclockwise. When all switches are open, no current flows through the motor, causing it to slow down and stop.

To adjust the speed, the switches are not continuously closed but rather opened and closed using a pulse-width modulation (PWM) signal with a fixed frequency and variable duty cycle, which changes the average current flow through the motor. The motor can be rapidly stopped (braked) by simultaneously closing switches 1-2 or 3-4.

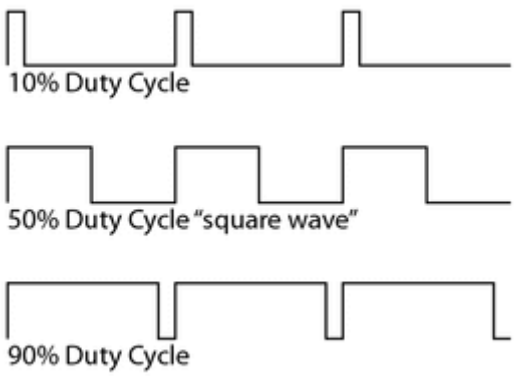
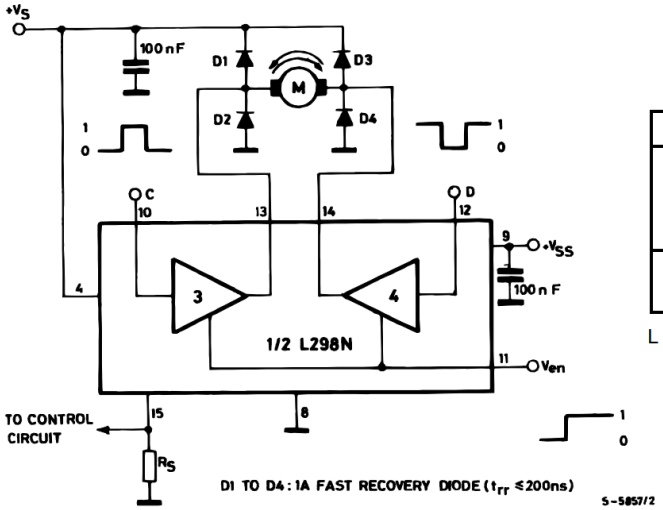


Figure 38: DC Motor Speed Control Using PWM

In the experimental kit, an L298N H-bridge module is used.

The switches are integrated inside the module and are controlled by providing appropriate signals to the control pins EN, C, and D. The L298N IC integrates two H-bridges.



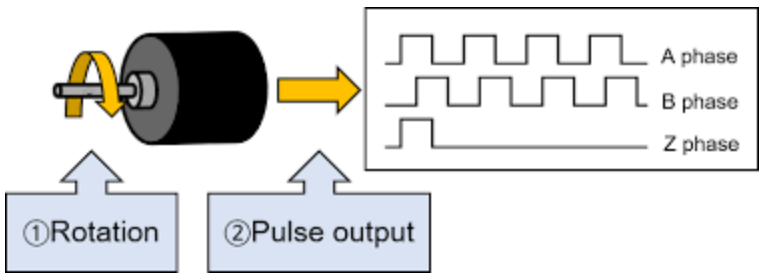
Inputs		Function
V_{en} = H	C = H ; D = L	Forward
	C = L ; D = H	Reverse
	C = D	Fast Motor Stop
V_{en} = L	C = X ; D = X	Free Running Motor Stop

L = Low H = High X = Don't care

Figure 39: Operating Principle of the L298N H-Bridge

To measure the actual speed of the motor, an encoder is attached to the motor shaft.

Depending on the type of encoder, there will be a certain number of pulses generated for each revolution of the motor. Encoders often provide two phased signals to determine the direction and position of the motor. Some encoders also feature a Z pulse to indicate when the motor passes through the zero position, useful for counting revolutions.



12.2 HARDWARE DESIGN

CAP NGUON 12V

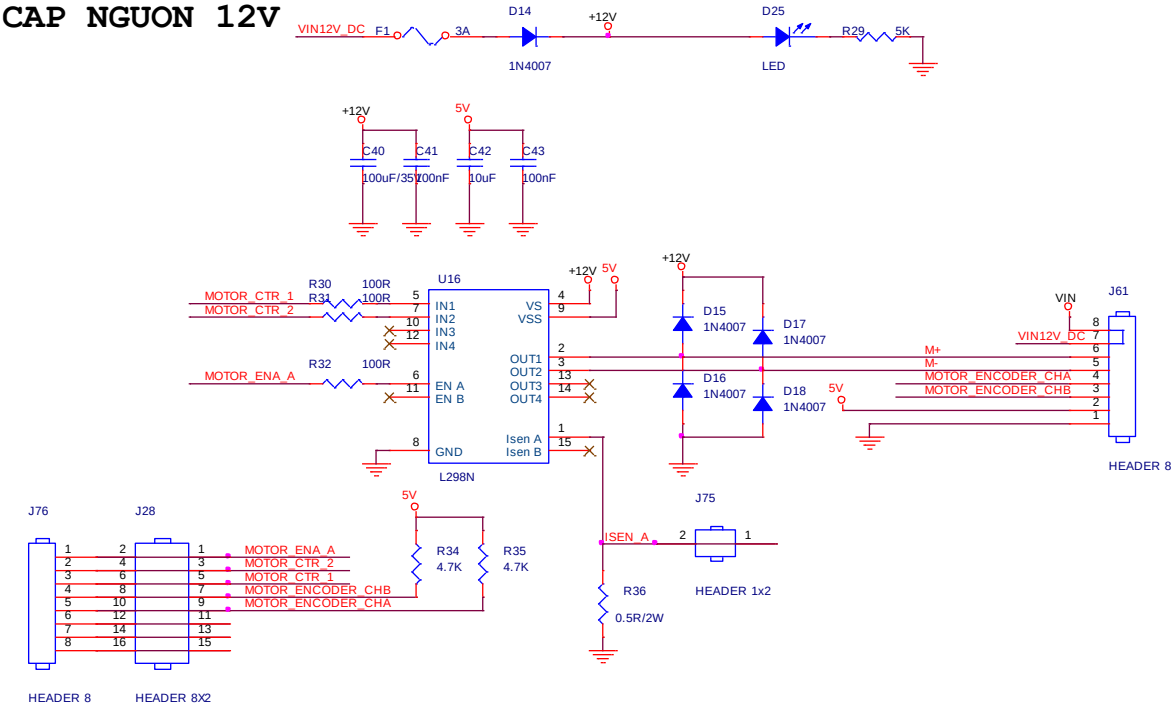


Figure 40: Block diagram of the motor control design

The design utilizes the L298 IC, which operates as an H-bridge. Control signals for opening/closing the switches are fed into MOTOR_CTRL_1 and MOTOR_CTRL_2, which are connected to the inputs IN1 and IN2 of the IC. The MOTOR_ENABLE signal allows the bridge to operate. The outputs OUT1 and OUT2 are connected to the two ends of the motor.

The current flowing through the motor passes through resistor R36, resulting in a voltage on the ISEN_A signal. This signal is connected to an ADC channel to measure the current passing through the motor.

The motor has integrated 2-channel encoder signals, A and B, which are connected to header J28. These signals are used to calculate the actual speed and direction of the motor.

12.3 HARDWARE CONNECTION AND PROGRAMMING

Connect the MOTOR_CTRL1 and MOTOR_CTRL2 signals to two port pins for direction control. Connect the MOTOR_ENABLE signal to a PWM output pin to control the motor speed.

Route one encoder signal into an input for counting pulses using a timer to count the number of pulses. You can use two separate port pins on the AVR to determine the direction of rotation.

12.3.1 MOTOR CONTROL

The motor is controlled using the MOTOR_ENABLE, MOTOR_CTRL1, and MOTOR_CTRL2 signals. The operating modes are described in Figure 43.

If MOTOR_ENABLE is set to 0, the motor will not be powered, and it will slowly decelerate due to inertia.

To brake the motor, set MOTOR_ENABLE to 1, and set both MOTOR_CTRL1 and MOTOR_CTRL2 to either 1 or 0.

To make the motor run clockwise, set MOTOR_ENABLE and MOTOR_CTRL1 to 1 and MOTOR_CTRL2 to 0. To make it run counterclockwise, set MOTOR_ENABLE and MOTOR_CTRL2 to 1 and MOTOR_CTRL1 to 0.

To control the speed, apply a PWM signal to the MOTOR_ENABLE and select one of the two direction signals (forward/backward) as the active polarity.

The PWM frequency may vary depending on the specific DC motor. For the motor used in the experiment, a PWM frequency of 1 kHz is suitable.

The PWM signal can be generated using a timer. For AVR microcontrollers, if you want to adjust the frequency, you may need to use Fast PWM in mode 7 or Phase Correct in Mode 5.

12.3.2 MEASURING MOTOR SPEED

The specifications of the motor are as follows:

Rotation speed	333 rpm (revolutions per minute)
Operating voltage	12 VDC
Type	Equipped with an encoder (which is a device used for measuring speed and position)
Shaft diameter	6mm
Motor length	47mm
Motor diameter	37mm
Transmission ratio	1:30
Pulses per revolution	11 pulses
Series	GB37

This motor already has a gearbox with a transmission ratio of 1:30. The encoder emits 11 pulses per revolution. Therefore, when the motor shaft completes one full revolution, there will be 330 pulses generated, corresponding to 30 revolutions of the encoder disc.

The speed of 333 rpm (revolutions per minute) in the datasheet is the speed of the motor shaft after going through the gearbox reduction.

To measure the motor speed, there are two common methods:

1. Counting the number of pulses from the encoder within 1 second to calculate the number of revolutions in 1 second.
2. Measuring the time it takes to complete one revolution and calculating the speed from that.

For the first method, the encoder signal is fed into an input of a timer. Another timer is used to count the time for 1 second. After 1 second, the timer counts the number of pulses received. If you count 330 pulses, it means the speed for that second was 1 revolution per second or 60 rpm (revolutions per minute).

For the second method, the pulse-counting timer is configured so that it triggers an interrupt when it reaches 330 counts. Another timer is used to measure the time from when the pulse count is 0 to when it reaches 330. From this time measurement, you can calculate the motor speed.

Instead of using a timer input to count pulses, you can also use an external interrupt pin. Each time there is a rising (or falling) edge in the signal, you increment the pulse count by 1. With this method, you don't need a timer to count pulses, and you can use that timer for other tasks.