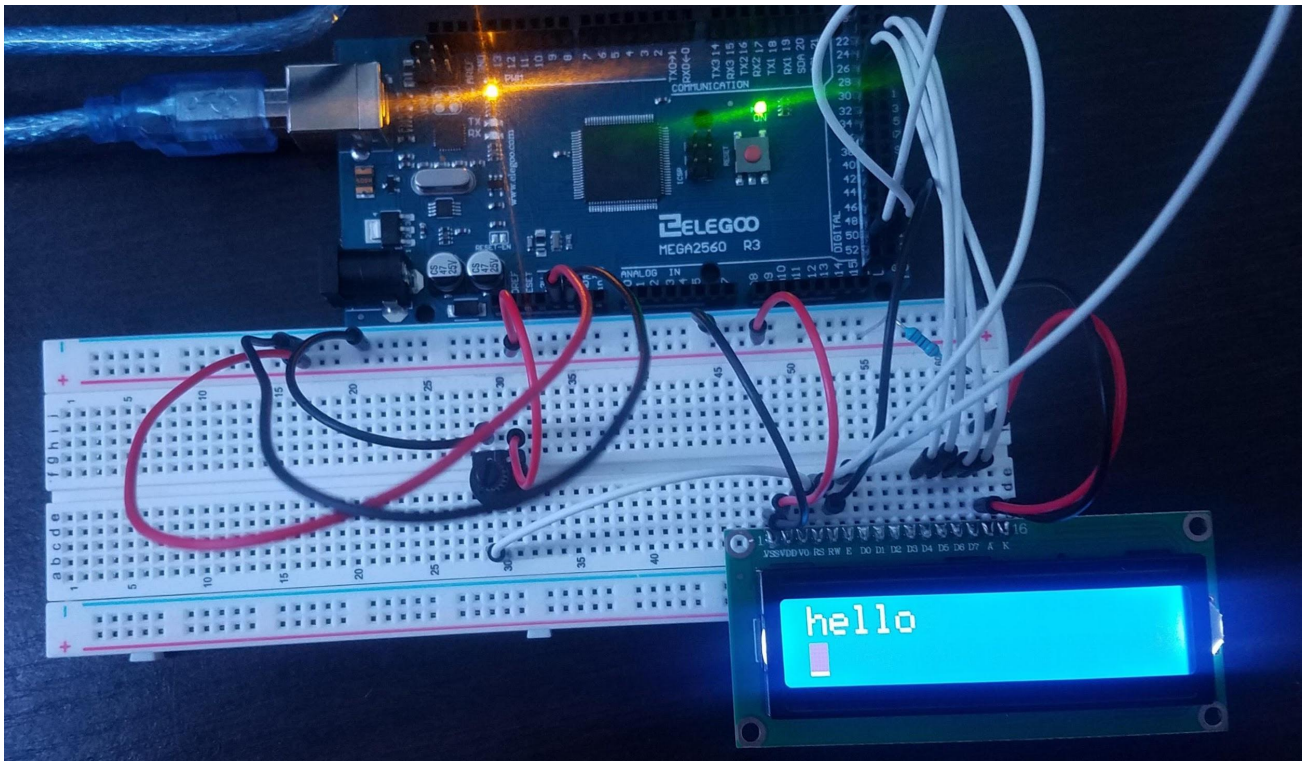
Lab 5 Report

LCD Display

CENG 347 Embedded Intelligent Systems



Samuel Donovan & Samantha Pfeiffer

2021-03-05

OVERVIEW

The purpose of this lab was to learn how to interface with a KS0066U and incorporate that knowledge with our previous knowledge of serial communication. This was also a lesson in timing.

We were assigned to write a program that allows the user to write to an LCD through `USART0`. The user should be able to write string to both lines, clear the display, and scroll the text across the display.

--- KS0066U LCD ---

The KS0066U LCD display has a total of 16 pins, with their functions outlined in **Table 1**.

Table 1 KS0066U Pins

Name	Function	Usage
VSS	Ground	Should be tied to GND (0V)
VDD	Supply Voltage	Should be supplied at +3V ±10% or +5±10%. Note that when supplied with +3V, propagation times will increase.
VO	Contrast Control	0V will yield minimum contrast, while 5V will yield maximum.
RS	Register Select	When set to HIGH, the data bus is tied to the IR (instruction register). When LOW, the data bus is tied to the DR (data register)
RW	Read/Write	When set to HIGH, data is read, when set to LOW, data will be written.
E	Read/Write Enable	Data is read/written out of the unit on this signal's falling edge. (HIGH->LOW)
D[0:7]	Data bus	The data bus that supplies data to/from the unit.
A	Backlight Supply Voltage	Should be tied to +5V with a 330Ω limiting resistor
K	Backlight Ground	Should be tied to GND (0V)

Memory

The KS0066U has a number of memory locations, two of which can be directly written and read from: the **IR** and **DR**. Which register you are accessing is controlled by **RS** (HIGH = IR, LOW = DR).

The **IR** (Instruction Register) holds the instruction that will be executed next. The IR can be written to and read from when the RS pin is LOW.

The **DR** (Data Register) holds the data that the instructions use.

The **AC** (Address Counter) holds the address to which the next read/write operation will be applied.

The **BF** (Busy Flag) is a single bit that is set HIGH when an operation is running.

DDRAM (Display Data RAM) holds the character codes representing what each display character is: In single-line mode, DDRAM spans the addresses 0x00 to 0x4F; in two-line mode, 0x00-0x27 holds the first lines data, while 0x40-0x67 holds the second line's display data.

CGRAM (Character Generator RAM) holds the data used to decode the DDRAM into a 5x8 grid of pixels. CGRAM has a ROM portion, which holds the base character set, and 64 user-writable registers (0x00-0x40). Every 8 bytes defines the mask for character codes 0 through 7. The 3 most significant bits are ignored in each byte. See **Figure 1** for an example of encoding 0 to "A":

Figure 1 CGRAM Encoding Example

Character Code (DDRAM data)								CGRAM Address						CGRAM Data							
D7	D6	D5	D4	D3	D2	D1	D0	A5	A4	A3	A2	A1	A0	P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	×	0	0	0	0	0	0	0	0	0	×	×	×	0	1	1	1	0
											0	0	1				1	0	0	0	1
											0	1	0				1	0	0	0	1
											0	1	1				1	1	1	1	1
											1	0	0				1	0	0	0	1
											1	0	1				1	0	0	0	1
											1	1	0				1	0	0	0	1
											1	1	1				0	0	0	0	0

Instructions

Table 2 Instruction Table

Instruction	Instruction Code										Description	Execution time (fosc= 270 kHz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write "20H" to DDRAM and set DDRAM address to "00H" from AC	1.53 ms
Return Home	0	0	0	0	0	0	0	0	1	-	Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1.53 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	SH	Assign cursor moving direction and enable the shift of entire display.	39 μs
Display ON/ OFF Control	0	0	0	0	0	0	1	D	C	B	Set display(D), cursor(C), and blinking of cursor(B) on/off control bit.	39 μs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data.	39 μs
Function Set	0	0	0	0	1	DL	N	F	-	-	Set interface data length (DL: 8-bit/4-bit), numbers of display line (N: 2-line/1-line) and, display font type (F:5×11dots/5×8 dots)	39 μs
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter.	39 μs
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter.	39 μs
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 μs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43 μs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43 μs

* "-": dont care

Timing

Figure 2 8-bit Mode Timing - Write Data

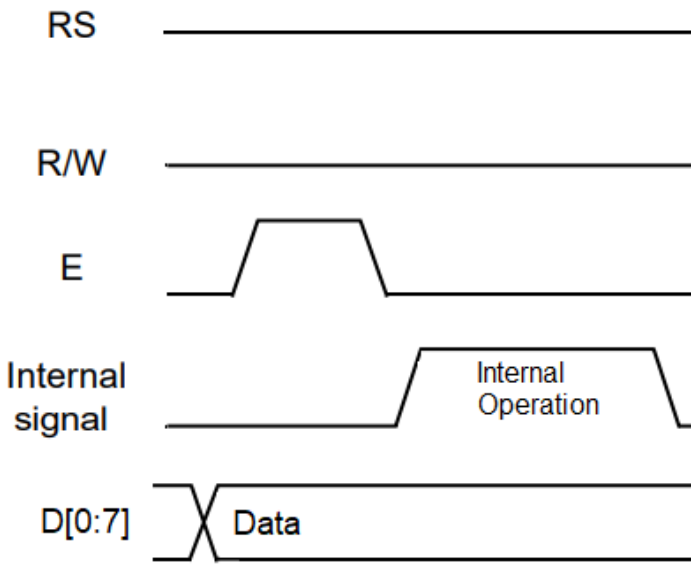


Figure 3 4-bit Mode Timing - Write Data

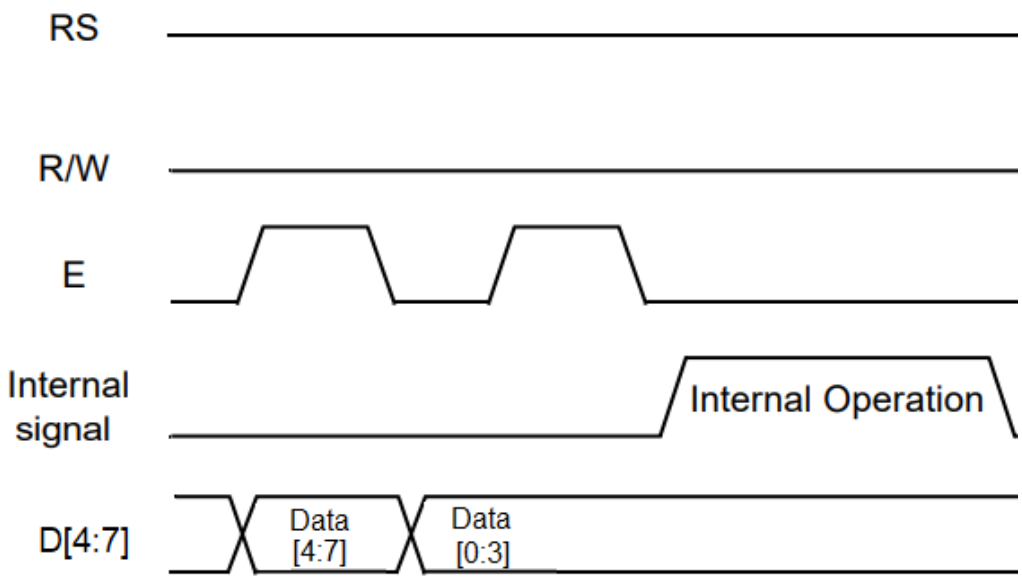


Figure 4 8-bit Mode Timing - Read Data

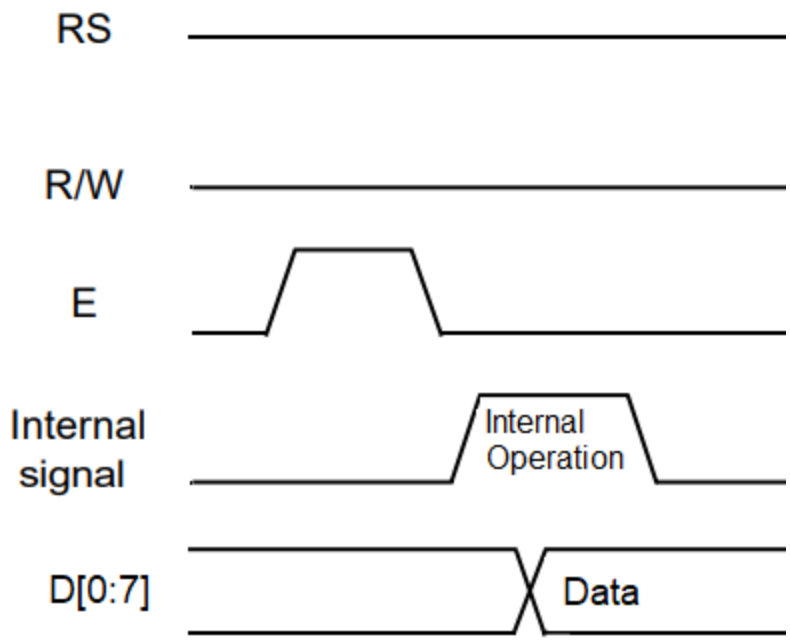
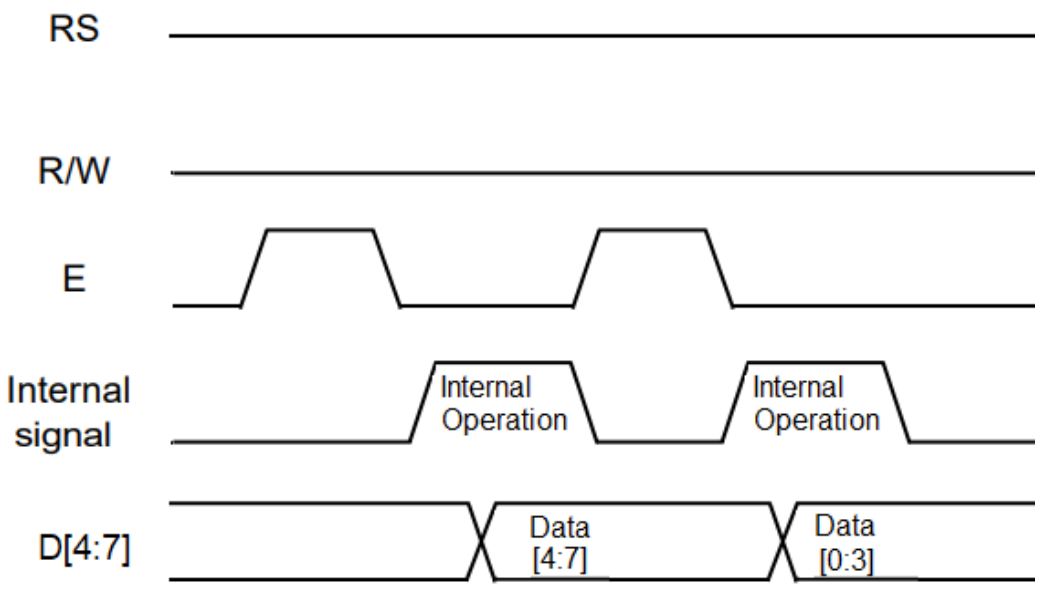


Figure 5 4-bit Mode Timing - Read Data



Setup Sequence

Figure 6 8-bit Mode Typical Setup Sequence

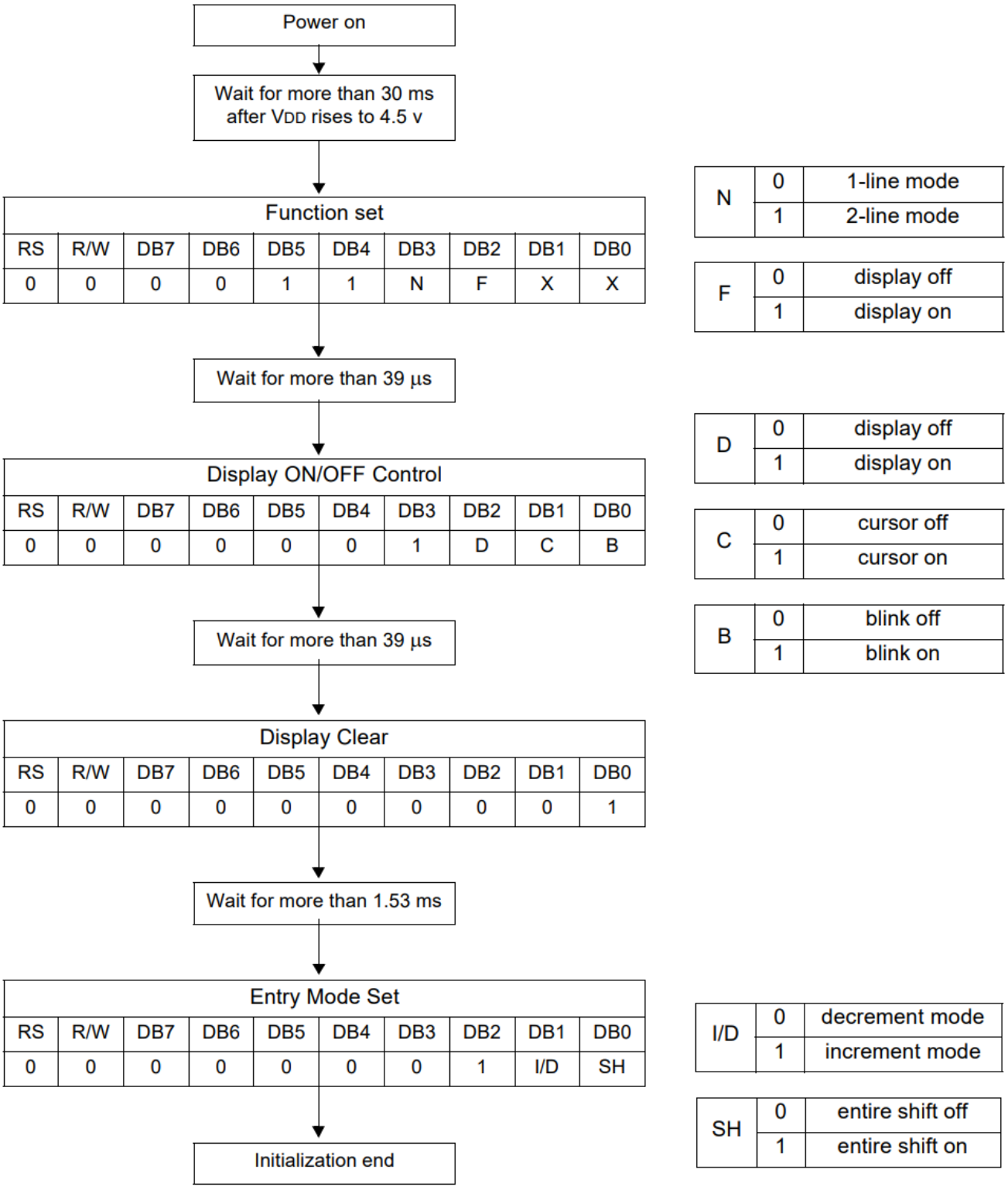
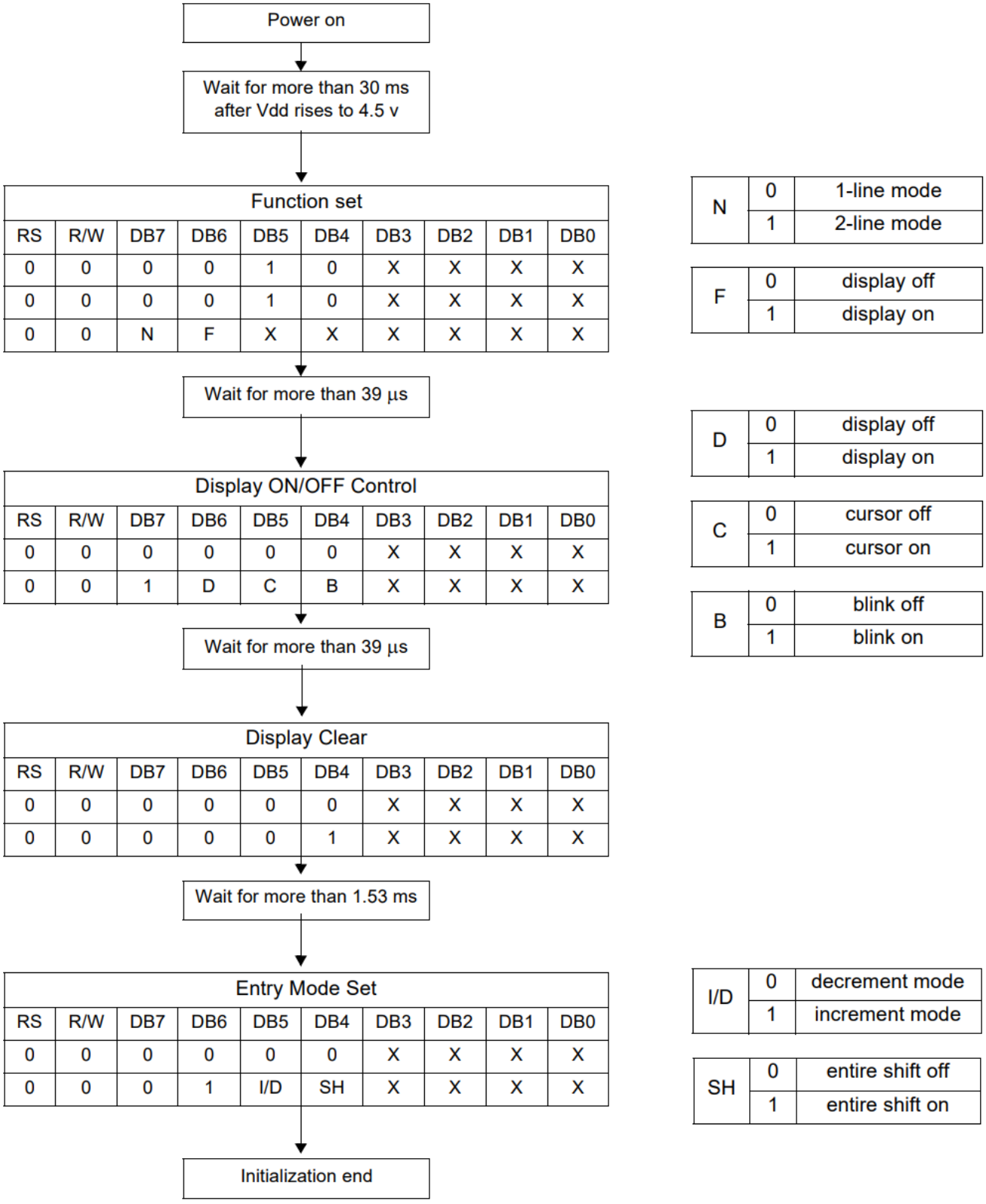


Figure 7 4-bit Mode Typical Setup Sequence



METHODS

--- KS0066 LCD (APPENDIX B & C) ---

The setup sequence for the LCD is accomplished in the constructor function. The function primarily follows the flow chart seen in **Figure** 7 above. Additionally, the constructor function copies the provided `RW`, `RS`, `E` and `D` GPIO ports and sets them to `OUTPUT` mode.

Both `ReadData` and `ReadAddress` are untested functions that are only presumed to work, but are not used by current functionality.

Both `WriteData` and `WriteInstructions` are fundamentally the same, with the only difference being the RS pin state. Both functions follow the timing diagram in **Figure 3**, and pulse `E` for around 1us, which is far over the minimum pulse width of 230ns defined on page 31 of the KS0066U datasheet.

Every other function in LCD is an abstraction of a function from **Table 2**, that implements it's variable bits as input parameters and incorporates it's execution time as a delay.

`SetPos`, `Write`, and `AddCustomChar` are all further abstractions of the base functions, which serve to simplify operations that would otherwise take multiple instructions to accomplish.

--- SERIAL COMMUNICATION (APPENDIX D & E) ---

Interfacing the LCD with serial communication was fairly simple. The `USART` class from Lab 4 only needed a few updates before it was ready to be used with the LCD. These updates include consolidating the `USART` functions into a class as well as adding a function to write strings of characters.

--- MAIN (APPENDIX A) ---

The setup code in `main` enables the `RxC` interrupt for `USART0`, as well as the global interrupt flag so we can handle users sending data over serial. It then initializes `USART0` with the `.begin` function, and prints the menu.

When the `ignoreNext` flag is set, it's simply cleared and then the ISR exits. This is to allow skipping of the newline character resulting from hitting “ENTER” when sending a command (‘`’ or ‘~’).

If the user enters ‘`’ to clear the display, `lcd.Clear()` is called and a status message is written to the serial port. The `ignoreNext` flag is set as well.

If the user enters ‘~’ to enable text scrolling, the boolean `scrollText` is set and a status message is written to the serial port. Additionally, the `ignoreNext` flag is set, same as for ‘`’.

`scrollText` is used during the `while(1)` loop of `main` to conditionally shift the display left every 1000ms.

ISSUES ENCOUNTERED

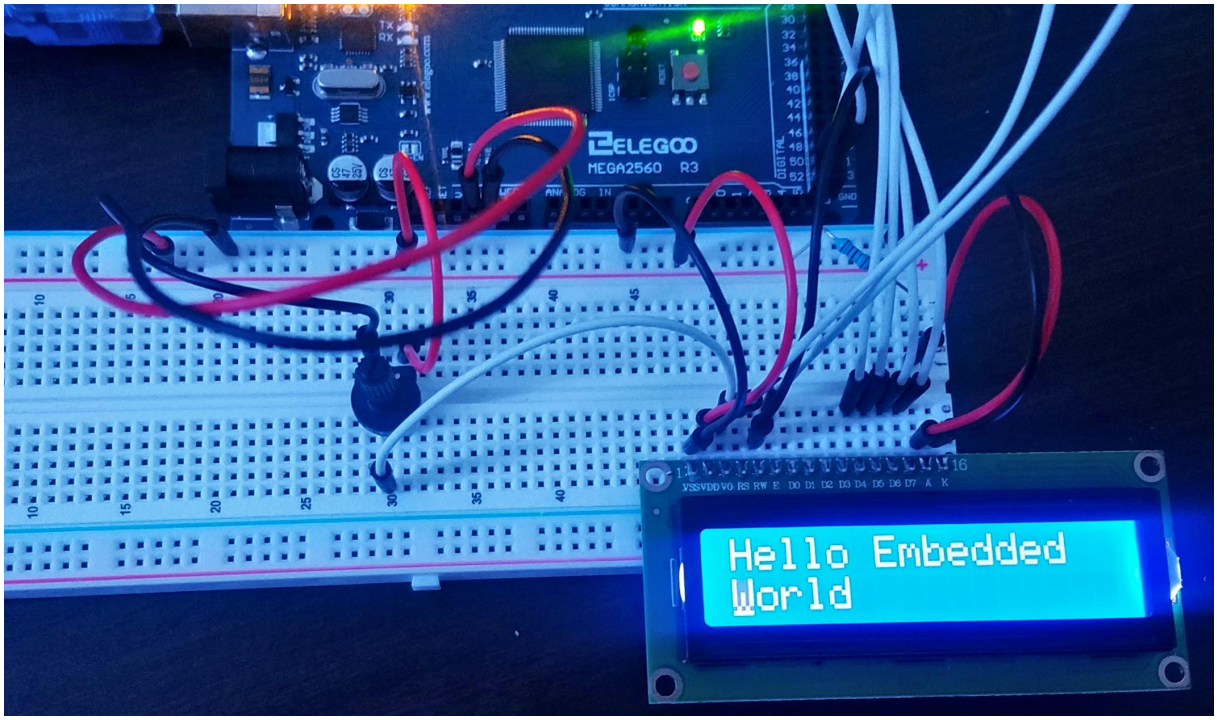
A few minor wiring issues were encountered. Namely, Sam Donovan did not ground his backlight and was confused as to why it was not lighting up. Also, Sam Pfeiffer hooked up the data pins on the LCD to the pins labeled `A4:A7` on the board as opposed to `PA4:PA7`. These mistakes were quickly recognized and corrected.

Another issue consisted of every 4th and 5th character not displaying to the LCD. We believed this to be happening due to the execution time of the ISR. In order to clear a line before printing on it, `lcd.WriteChar()` was being called 16 times (each taking over 50us). When removing this loop, no characters were skipped. Due to the nature of our implementation, the line is not cleared before printing, leaving the old characters if the new message is shorter. We decided that although this does not look the best, implementing a solution that allowed us to clear the display every time a new string was to be written was too tedious, and not defined on the rubric.

The only unresolved issue occurs when resetting the LCD. Even when running the same code, Sam Pfeiffer's LCD switches to `OneLineMode` after the reset button is pressed. This problem persists even if the code is reuploaded to the board. The only apparent way to "reset" Sam Pfeiffer's LCD is to unplug the board from her laptop. Sam Donovan's LCD does not exhibit the same behavior. However, his LCD will occasionally boot up in `OneLineMode` however, his reset button works so the cause is not apparent. It is assumed to be a hardware difference with the LCDs as the code is identical.

RESULTS

Occasional `OneLineModes` aside, the LCD functions as expected. Single characters as well as strings of characters can be printed to the display. Pressing enter wraps the cursor to the leftmost position of the other line. Entering ‘`’ clears the display and entering ‘~’ toggles text scrolling. If a user enters over 16 characters on one line, up to 40 characters are written and viewable when text scrolling is on.



APPENDIX A: Lab5.ino

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

//Vars
GPIO D[4] = {
  GPIO(&DDRA, &PORTA, &PINA, 4),
  GPIO(&DDRA, &PORTA, &PINA, 5),
  GPIO(&DDRA, &PORTA, &PINA, 6),
  GPIO(&DDRA, &PORTA, &PINA, 7)
};

GPIO RS = GPIO(&DDRB, &PORTB, &PINB, 0);
GPIO RW = GPIO(&DDRB, &PORTB, &PINB, 2);
GPIO E  = GPIO(&DDRB, &PORTB, &PINB, 1);

LCD lcd = LCD(RS, RW, E, D);;

//ISR
volatile byte line = 1;
volatile bool ignoreNext = false;
volatile bool scrollText = false;
ISR(USART0_RX_vect)
{
    char c = USART0.Read();

//Commands
    if(ignoreNext) //Ignore char
    {
        ignoreNext = false;
    }
    else if(c == '\n') // New line
    {
        line = line == 1 ? 2 : 1;
        USART0.Write('\n');
        lcd.SetPos(line, 0);
```

```
    }
    else if(c == '`') //Clear
    {
        lcd.Clear();
        USART0.Writeln("Display cleared");
        ignoreNext = true;
//      clearLineOnNext = true;
    }
    else if(c == '~') //Toggle scrolling
    {
        if(scrollText)
        {
            USART0.Writeln("Text scroll disabled");
            scrollText = false;
        }
        else
        {
            USART0.Writeln("Text scroll enabled");
            scrollText = true;
        }
        ignoreNext = true;
    }
    else// if(isprint(c)) //Printable char
    {
//      if(clearLineOnNext)
//      {
//            for(byte i = 0; i < 0x28; i++)
//                lcd.Write(' ');
//          lcd.SetPos(line, 0);
//
//          clearLineOnNext = false;
//      }

        lcd.Write(c);
        USART0.Write(c);
    }
}


int main(void)
```

```
{
    _delay_ms(100);

    USART0.EnableRxCompleteInterrupt();
    sei();
    USART0.begin(9600);

    USART0.Writeln("--------------------");
    USART0.Writeln("Welcome to samsLCD!");
    USART0.Writeln("Send something for it to appear on the
LCD");
    USART0.Writeln("Use your enter key to switch lines");
    USART0.Writeln("Send '`' to clear the display");
    USART0.Writeln("Send '~' to start/stop text scrolling");
    USART0.Writeln("--------------------");

    while(1)
    {   //Text scrolling
        if(scrollText)
        {
            lcd.ShiftDisplayLeft();
            _delay_ms(1000);
        }
    };
    return 1;
}
```

APPENDIX B: LCD Header File (samsLCD.h)

```
#ifndef SAMS_LCD
#define SAMS_LCD

#include "samsGPIO.h"
#include <avr/io.h>
#include <util/delay.h>

class LCD
{
public:
//GPIO Pins
    GPIO RS; //Register Select Pin (1 = Data Input | 0 =
Instruction Input)
    GPIO RW; //Read/Write Select Pin (1 = Read | 0 = Write)
    GPIO E;  //Enable Signal Pin
    GPIO D[4]; //Data Pins

//Class
    LCD();
    LCD(GPIO RS, GPIO RW, GPIO E, GPIO D[]);

//Core
    bool IsBusy();
    byte ReadAddress();
    byte ReadData();
    void WriteData(byte data);
    void WriteInstruction(byte instruction);
//Base Functions
    void Clear();
    void ReturnHome();
    void SetEntryMode(bool increment, bool entireShift=false);
    void DisplayControl(bool on, bool cursor=true, bool
blink=true);
    void ShiftCursorLeft();
    void ShiftCursorRight();
    void ShiftDisplayLeft();
    void ShiftDisplayRight();
```

```
    void SetFunction(bool twoLineMode=true, bool
fontType11=false);
    void SetCGRAMAddress(byte addr);
    void SetDDRAMAddress(byte addr);
//Convience
    void SetPos(byte line = 1, byte pos = 0);
    void Write(byte c, byte line = 0, byte pos = 0);
    void Write(char* buffer, byte line = 0, byte pos = 0, byte
wrapAt = -1);
    void AddCustomChar(byte charCode, byte* bitMap);
};
#endif
```

APPENDIX C: LCD Program File (samsLCD.cpp)

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


//Class
LCD::LCD()
{

}
LCD::LCD(GPIO RS, GPIO RW, GPIO E, GPIO D[])
{
  //Setup GPIO
  this->RS = RS;
  this->RW = RW;
  this->E = E;

  RS.SetMode(OUTPUT);
  RW.SetMode(OUTPUT);
  E.SetMode(OUTPUT);


  for(int i=0; i < 4; i++)
  {
    this->D[i] = D[i];
    D[i].SetMode(OUTPUT);
  }

    //Prepare Instruction Write
    E.Clear();
    RS.Clear();
    RW.Clear();

    //Wait at least 30ms after Vdd rises to 4.5V
    _delay_ms(100);

    //Reset (Function Set to 8-bit mode) //Set D7:D4 to [0011]
```

```
    D[3].Clear();
    D[2].Clear();
    D[1].Set();
    D[0].Set();
    E.Set();
    _delay_us(1);
    E.Clear();

    //Execution Time: 39us
    _delay_us(50);

    //Enable 4-bit (Function Set to 4-bit mode) //Set D7:D4 to
[0010]
    D[3].Clear();
    D[2].Clear();
    D[1].Set();
    D[0].Clear();
    E.Set();
    _delay_us(1);
    E.Clear();

    //Execution Time: 39us
    _delay_us(50);

    //Init Sequence: See documentation pg. 27
    SetFunction(true, false); //Function Set, 2-line mode, font
8

    DisplayControl(true, true, true); //Display ON, Cursor ON,
Blink ON

    Clear();

    SetEntryMode(true, false); //Increment mode, Entire shift
OFF
}

//Core
bool LCD::IsBusy()
{
```

```
    RS.Clear();
    E.Clear();
    RW.Clear();

    D[3].Input();
    bool b = D[3].GetState();
    D[3].Output();

    return b;
}
byte LCD::ReadAddress()
{
  byte b = 0;

  //Prepare Read Address
  E.Clear();
  RS.Clear();
  RW.Set();
  D[3].Input();
  D[2].Input();
  D[1].Input();
  D[0].Input();

  //Execution Time: 0us

  //Read high nibble
  E.Set();
  _delay_us(1);
  if(D[3].GetState())
    b |= 1;
  b = b << 1;
  if(D[2].GetState())
    b |= 1;
  b = b << 1;
  if(D[1].GetState())
    b |= 1;
  b = b << 1;
  if(D[0].GetState())
    b |= 1;
  b = b << 1;
```

```
  E.Clear();
  _delay_us(1);

  //Read low nibble
  E.Set();
  _delay_us(1);
  if(D[3].GetState())
    b |= 1;
  b = b << 1;
  if(D[2].GetState())
    b |= 1;
  b = b << 1;
  if(D[1].GetState())
    b |= 1;
  b = b << 1;
  if(D[0].GetState())
    b |= 1;
  b = b << 1;
  E.Clear();
  _delay_us(1);


  //Cleanup
  D[3].Output();
  D[2].Output();
  D[1].Output();
  D[0].Output();

  return b;
}
byte LCD::ReadData()
{
  byte b = 0;

  //Wait until not busy
  while(IsBusy()){};

  //Prepare Read Data
  E.Clear();
  RS.Set();
```

```
RW.Set();
D[3].Input();
D[2].Input();
D[1].Input();
D[0].Input();

//Execution Time: 43us
_delay_us(50);



//Read high nibble
E.Set();
_delay_us(1);
if(D[3].GetState())
  b |= 1;
b = b << 1;
if(D[2].GetState())
  b |= 1;
b = b << 1;
if(D[1].GetState())
  b |= 1;
b = b << 1;
if(D[0].GetState())
  b |= 1;
b = b << 1;
E.Clear();
_delay_us(1);

//Read low nibble
E.Set();
_delay_us(1);
if(D[3].GetState())
  b |= 1;
b = b << 1;
if(D[2].GetState())
  b |= 1;
b = b << 1;
if(D[1].GetState())
  b |= 1;
```

```
  b = b << 1;
  if(D[0].GetState())
    b |= 1;
  b = b << 1;
  E.Clear();
  _delay_us(1);


  //Cleanup
  D[3].Output();
  D[2].Output();
  D[1].Output();
  D[0].Output();

  return b;
}
void LCD::WriteData(byte data)
{
  //Prepare Write Data
  E.Clear();
  RS.Set();
  RW.Clear();

  //Write high nibble
  D[3].SetState(data & 0b10000000);
  D[2].SetState(data & 0b01000000);
  D[1].SetState(data & 0b00100000);
  D[0].SetState(data & 0b00010000);
  E.Set();
  _delay_us(1);
  E.Clear();
  _delay_us(1);


  //Write low nibble
  D[3].SetState(data & 0b00001000);
  D[2].SetState(data & 0b00000100);
  D[1].SetState(data & 0b00000010);
  D[0].SetState(data & 0b00000001);
  E.Set();
```

```
  _delay_us(1);
  E.Clear();

  _delay_us(50); //Execution Time: 43us
}
void LCD::WriteInstruction(byte instruction)
{
  //Prepare Write Instruction
  E.Clear();
  RS.Clear();
  RW.Clear();

  //Write high nibble
  D[3].SetState(instruction & 0b10000000);
  D[2].SetState(instruction & 0b01000000);
  D[1].SetState(instruction & 0b00100000);
  D[0].SetState(instruction & 0b00010000);
  E.Set();
  _delay_us(1);
  E.Clear();
  _delay_us(1);


  //Write low nibble
  D[3].SetState(instruction & 0b00001000);
  D[2].SetState(instruction & 0b00000100);
  D[1].SetState(instruction & 0b00000010);
  D[0].SetState(instruction & 0b00000001);
  E.Set();
  _delay_us(1);
  E.Clear();
}


//Functions
void LCD::Clear()
{
    WriteInstruction(0b00000001);
    _delay_ms(2); //Execution Time: 1.53ms
}
```

```
void LCD::ReturnHome()
{
    WriteInstruction(0b00000010);
    _delay_ms(2); //Execution Time: 1.53ms
}
void LCD::SetEntryMode(bool increment, bool entireShift)
{
    byte instruction = 0b00000100;

    if(increment)
        instruction |= 0b10;
    if(entireShift)
        instruction |= 0b01;

    WriteInstruction(instruction);

    _delay_us(50); //Execution Time: 39us
}
void LCD::DisplayControl(bool on, bool cursor, bool blink)
{
    byte instruction = 0b00001000;

    if(on)
        instruction |= 0b100;
    if(cursor)
        instruction |= 0b010;
    if(blink)
        instruction |= 0b001;

    WriteInstruction(instruction);

    _delay_us(50); //Execution Time: 39us
}
void LCD::ShiftCursorLeft()
{
    WriteInstruction(0b00010000);

    _delay_us(50); //Execution Time: 39us
}
void LCD::ShiftCursorRight()
```

```
{
    WriteInstruction(0b00010100);


    _delay_us(50); //Execution Time: 39us
}
void LCD::ShiftDisplayLeft()
{
    WriteInstruction(0b00011000);

    _delay_us(50); //Execution Time: 39us
}
void LCD::ShiftDisplayRight()
{
    WriteInstruction(0b00011100);

    _delay_us(50); //Execution Time: 39us
}
void LCD::SetFunction(bool twoLineMode, bool fontType11)
{
    byte instruction = 0b00100000;

    if(twoLineMode)
        instruction |= 0b01000;
    if(fontType11)
        instruction |= 0b00100;

    WriteInstruction(instruction);

    _delay_us(50); //Execution Time: 39us
}
void LCD::SetCGRAMAddress(byte addr)
{
    byte instruction = 0b01000000;

    instruction |= (addr & 0b00111111);

    WriteInstruction(instruction);

    _delay_us(50); //Execution Time: 39us
```

```
}
void LCD::SetDDRAMAddress(byte addr)
{
    byte instruction = 0b10000000;

    instruction |= (addr & 0b01111111);

    WriteInstruction(instruction);

    _delay_us(50); //Execution Time: 39us
}

//Convience
void LCD::SetPos(byte line, byte pos)
{
    if(line == 2)
        pos += 0x40;

    SetDDRAMAddress(pos);
}
void LCD::Write(byte c, byte line, byte pos)
{
    if(line != 0)
        SetPos(line, pos);

    WriteData(c);
}
void LCD::Write(char* buffer, byte line, byte pos, byte wrapAt)
{
    if(line != 0)
        SetPos(line, pos);

    for(byte i = 0; buffer[i] != 0; i++)
    {
        Write(buffer[i]);
        if(i == wrapAt)
            SetPos(2, 0);
    }
}
void LCD::AddCustomChar(byte charCode, byte bitMap[])
```

```
{
    if(charCode > 7)
        return;

    SetCGRAMAddress(charCode * 8); //Each char mask takes up 8
bytes

    for(byte i = 0; i < 8; i++)
        WriteData(bitMap[i]);

    SetPos(1, 0);
}
```

APPENDIX D: USART Header File (samsUSART.h)

```
#ifndef SAMS_SERIAL
#define SAMS_SERIAL
#include <avr/io.h>
typedef unsigned char byte;
typedef volatile byte reg;

class USART
{
public:
    reg* BRRL; //Baud Rate Register Lower//
    reg* BRRH; //Baud Rate Register Higher//
    reg* CSRA; //Control & Status Register A// [RxC,   TxC,
UDRE,  FE,   DOR,   PE,    U2X,   MPCM]
    reg* CSRB; //Control & Status Register B// [RXCIE, TXCIE,
UDRIE, RXEN, TXEN,  UCSZ2, RXB8,  TXB8]
    reg* CSRC; //Control & Status Register C// [URSEL, UMSEL,
UPM1,  UPM0, USBS,  UCSZ1, UCSZ0, UCPOL]
    reg* DR;   //Data Register//               [UDR[7:0]]

//Class
    USART(reg* UBRRL, reg* UBRRH, reg* UCSRA, reg* UCSRB, reg*
UCSRC, reg* UDR);
//CSRA//
    bool RxComplete(); //RxC (R)
    bool TxComplete();  //TxC (R)
    void SetTxComplete(); //TxC (W 1)
    void ClearTxComplete(); //TxC (W 0)
    bool DataRegisterEmpty(); //UDRE (R)
    bool FrameError(); //FE (R)
    bool DataOverrunError(); //DOR (R)
    bool ParityError(); //PE (R)
    bool DoubleTransmissionSpeedIsEnabled(); //U2X (R)
    void EnableDoubleTransmissionSpeed(); //U2X (W 1)
    void DisableDoubleTransmissionSpeed(); //U2X (W 0)
    bool MultiProcessorCommunicationModeIsEnabled(); //MPCM (R)
    void EnableMultiProcessorCommunicationMode(); //MPCM (W 1)
    void DisableMultiProcessorCommunicationMode(); //MPCM (W 0)
```

```
//UCSRB//
    bool RxCompleteInterruptIsEnabled(); //RXCIE (R)
    void EnableRxCompleteInterrupt(); //RXCIE (W 1)
    void DisableRxCompleteInterrupt(); //RXCIE (W 0)
    bool TxCompleteInterruptIsEnabled(); //TXCIE (R)
    void EnableTxCompleteInterrupt(); //TXCIE (W 1)
    void DisableTxCompleteInterrupt(); //TXCIE (W 0)
    bool DataRegisterEmptyInterruptIsEnabled(); //UDRIE (R)
    void EnableDataRegisterEmptyInterrupt(); //UDRIE (W 1)
    void DisableDataRegisterEmptyInterrupt(); //UDRIE (W 0)
    bool RxIsEnabled(); //RXEN (R)
    void EnableRx(); //RXEN (W 1)
    void DisableRx(); //RXEN (W 0)
    bool TxIsEnabled(); //TXEN (R)
    void EnableTx(); //TXEN (W 1)
    void DisableTx(); //TXEN (W 0)
    byte GetRxBit9(); //RXB8 (R)
    byte GetTxBit9(); //TXB8 (R)
    void SetTxBit9(); //TXB8 (W 1)
    void ClearTxBit9(); //TXB8 (W 0)
//UCSRC//
    enum Mode{
       ASYNC = 0b00,
       SYNC = 0b01,
       MSPIM = 0b11
    };
    Mode GetMode(); //UMSEL (R)
    void SetMode(Mode mode=ASYNC); //UMSEL (W)

    enum Parity {
        DISABLED = 0b00,
        RESERVED = 0b01,
        EVEN = 0b10,
        ODD = 0b11
    };
    Parity GetParity(); //UPM (R)
    void SetParity(Parity mode=DISABLED); //UPM (W)
    enum StopBits{
        ONE = 0,
```

```
        TWO = 1
    };
    StopBits GetStopBits(); //USBS (R)
    void SetStopBits(StopBits size=ONE); //USBS (W)
   enum FrameSize {
        FIVE = 0b000,
        SIX = 0b001,
        SEVEN = 0b101,
        EIGHT = 0b011,
        NINE = 0b111
    };
    FrameSize GetFrameSize(); //UCSZ (R)
    void SetFrameSize(FrameSize size = EIGHT); //UCSZ (W)

    enum ClockPolarity{
        RISING_EDGE = 0,
        RALLING_EDGE = 1
    };
    ClockPolarity GetClockPolarity();
    void SetClockPolarity(ClockPolarity polarity=RISING_EDGE);

//Core Funcs
    void SetBaudRate(unsigned long BAUD_RATE);
    void Write(char c);
    void Write(char* buffer);
    void Writeln(char* buffer);
    byte Read();
    void begin(unsigned long BAUD_RATE);
};

//USART Definitions//
extern USART USART0;
extern USART USART1;
extern USART USART2;
extern USART USART3;
#endif
```

APPENDIX E: USART Program File (samsUSART.cpp)

```
#include "samsUSART.h"

//USART Definitions
USART USART0 = USART(&UBRR0L, &UBRR0H, &UCSR0A, &UCSR0B,
&UCSR0C, &UDR0);
USART USART1 = USART(&UBRR1L, &UBRR1H, &UCSR1A, &UCSR1B,
&UCSR1C, &UDR1);
USART USART2 = USART(&UBRR2L, &UBRR2H, &UCSR2A, &UCSR2B,
&UCSR2C, &UDR2);
USART USART3 = USART(&UBRR3L, &UBRR3H, &UCSR3A, &UCSR3B,
&UCSR3C, &UDR3);

//Class
USART::USART(reg* UBRRL, reg* UBRRH, reg* UCSRA, reg* UCSRB,
reg* UCSRC, reg* UDR)
{
    BRRL = UBRRL;
    BRRH = UBRRH;
    CSRA = UCSRA;
    CSRB = UCSRB;
    CSRC = UCSRC;
    DR = UDR;
}

//UCSRA//
//RxC (R)
bool USART::RxComplete()
{
    return *CSRA & 0b10000000; // != 0
}

//TxC (W/R)
bool USART::TxComplete()
{
    return *CSRA & 0b01000000; // != 0
}
void USART::SetTxComplete()
```

```
{
    *CSRA |= 0b01000000;
}
void USART::ClearTxComplete()
{
    *CSRA &= 0b10111111;
}

//UDRE (R)
bool USART::DataRegisterEmpty()
{
    return *CSRA & 0b00100000; //!= 0
}

//FE (R)
bool USART::FrameError()
{
    return *CSRA & 0b00010000; //!= 0
}

//DOR (R)
bool USART::DataOverrunError()
{
    return *CSRA & 0b00001000; //!= 0
}

//PE (R)
bool USART::ParityError()
{
    return *CSRA & 0b00000100; //!= 0
}

//U2X (R/W)
bool USART::DoubleTransmissionSpeedIsEnabled()
{
    return *CSRA & 0b00000010; //!= 0
}
void USART::EnableDoubleTransmissionSpeed()
{
    *CSRA |= 0b00000010;
```

```
}
void USART::DisableDoubleTransmissionSpeed()
{
    *CSRA &= 0b11111101;
}

//MPCM (R/W)
bool USART::MultiProcessorCommunicationModeIsEnabled()
{
    return *CSRA & 0b00000001;
}
void USART::EnableMultiProcessorCommunicationMode()
{
    *CSRA |= 0b00000001;
}
void USART::DisableMultiProcessorCommunicationMode()
{
    *CSRA &= 0b11111110;
}




//UCSRB//
//RXCIE (R/W)
bool USART::RxCompleteInterruptIsEnabled()
{
    return *CSRB & 0b10000000; //!= 0
}
void USART::EnableRxCompleteInterrupt()
{
   *CSRB |= 0b10000000;
}
void USART::DisableRxCompleteInterrupt()
{
    *CSRB &= 0b01111111;
}

//TXCIE (R/W)
bool USART::TxCompleteInterruptIsEnabled()
```

```
{
    return *CSRB & 0b01000000; //!= 0
}
void USART::EnableTxCompleteInterrupt()
{
    *CSRB |= 0b01000000;
}
void USART::DisableTxCompleteInterrupt()
{
    *CSRB &= 0b10111111;
}

//UDRIE (R/W)
bool USART::DataRegisterEmptyInterruptIsEnabled()
{
    return *CSRB & 0b00100000; //!= 0
}
void USART::EnableDataRegisterEmptyInterrupt()
{
    *CSRB |= 0b00100000;
}
void USART::DisableDataRegisterEmptyInterrupt()
{
    *CSRB &= 0b11011111;
}

//RXEN (R/W)
bool USART::RxIsEnabled()
{
    return *CSRB & 0b00010000; //!= 0
}
void USART::EnableRx()
{
    *CSRB |= 0b00010000;
}
void USART::DisableRx()
{
    *CSRB &= 0b11101111;
}
```

```
//TXEN (R/W)
bool USART::TxIsEnabled()
{
    return *CSRB & 0b00001000; //!= 0
}
void USART::EnableTx()
{
    *CSRB |= 0b00001000;
}
void USART::DisableTx()
{
    *CSRB &= 0b11110111;
}

//UCSZ2 (R/W) (See FrameSize funcs)

//RXB8 (R)
byte USART::GetRxBit9()
{
    return (*CSRB & 0b00000010) >> 1;
}
byte USART::GetTxBit9()
{
    return *CSRB & 0b00000001;
}
void USART::SetTxBit9()
{
    *CSRB |= 0b00000001;
}
void USART::ClearTxBit9()
{
    *CSRB &= 0b11111110;
}



//UCSRC//
//UMSEL (R/W)
USART::Mode USART::GetMode()
{
```

```
    return (Mode)((*CSRC & 0b11000000) >> 6);
}
void USART::SetMode(Mode mode)
{
    *CSRC &= 0b00111111; //Clear
    *CSRC |= mode << 6; //Set
}

//UPM (R/W)
USART::Parity USART::GetParity()
{
    return (Parity)((*CSRC & 0b00110000) >> 4);
}
void USART::SetParity(Parity mode)
{
    *CSRC &= 0b11001111; //Clear
    *CSRC |= mode << 4; //Set
}

//USBS (R/W)
USART::StopBits USART::GetStopBits()
{
    return (StopBits)((*CSRC & 0b00001000) >> 3);
}
void USART::SetStopBits(StopBits size)
{
    *CSRC &= 0b11110111; //Clear
    *CSRC |= size << 3; //Set
}

//UCSZ (R/W)
USART::FrameSize USART::GetFrameSize()
{
    return (FrameSize)((*CSRB & 0b00000100) + ((*CSRC &
0b00000110) >> 1));
}
void USART::SetFrameSize(FrameSize size) //size can be 5-9 bits
{
    *CSRC &= 0b11111001; //Clear UCSZ 0 & 1
    *CSRB &= 0b11111011; //Clear UCSZ 2
```

```
    *CSRC |= size << 1; //Set UCSZ 0 & 1
    *CSRC |= size; //Set UCSZ 2
}

//UCPOL
USART::ClockPolarity USART::GetClockPolarity()
{
    return (ClockPolarity)(*CSRC & 0b00000001);
}
void USART::SetClockPolarity(ClockPolarity polarity)
{
    *CSRC &= 0b11111110; //Clear
    *CSRC |= polarity; //Set
}


//Core Funcs
void USART::SetBaudRate(unsigned long baud_rate)
{
    unsigned short val = 0;

    if(GetMode() == SYNC) //SYNC
        val = F_CPU/(2UL*baud_rate)-1;
    else if(DoubleTransmissionSpeedIsEnabled()) //Double ASYNC
        val = F_CPU/(8UL*baud_rate)-1;
    else //ASYNC
        val = F_CPU/(16UL*baud_rate)-1;

    *BRRH = (val >> 8) & 0b00001111; //Set URBBH
    *BRRL = val; //Set UBRRL
}
void USART::Write(char data)
{
    while(!DataRegisterEmpty()){}
    *DR = data;
}
void USART::Write(char* buffer)
{
    for(byte i = 0; buffer[i] != 0; i++)
```

```
    {
        Write(buffer[i]);
    }
}
void USART::Writeln(char* buffer)
{
    Write(buffer);
    Write('\n');
}
byte USART::Read()
{
    return *DR;
}
void USART::begin(unsigned long baud_rate)
{
    SetFrameSize(EIGHT);
    SetStopBits(ONE);
    SetParity(DISABLED);
    SetBaudRate(baud_rate);
    EnableTx();
    EnableRx();
}
```