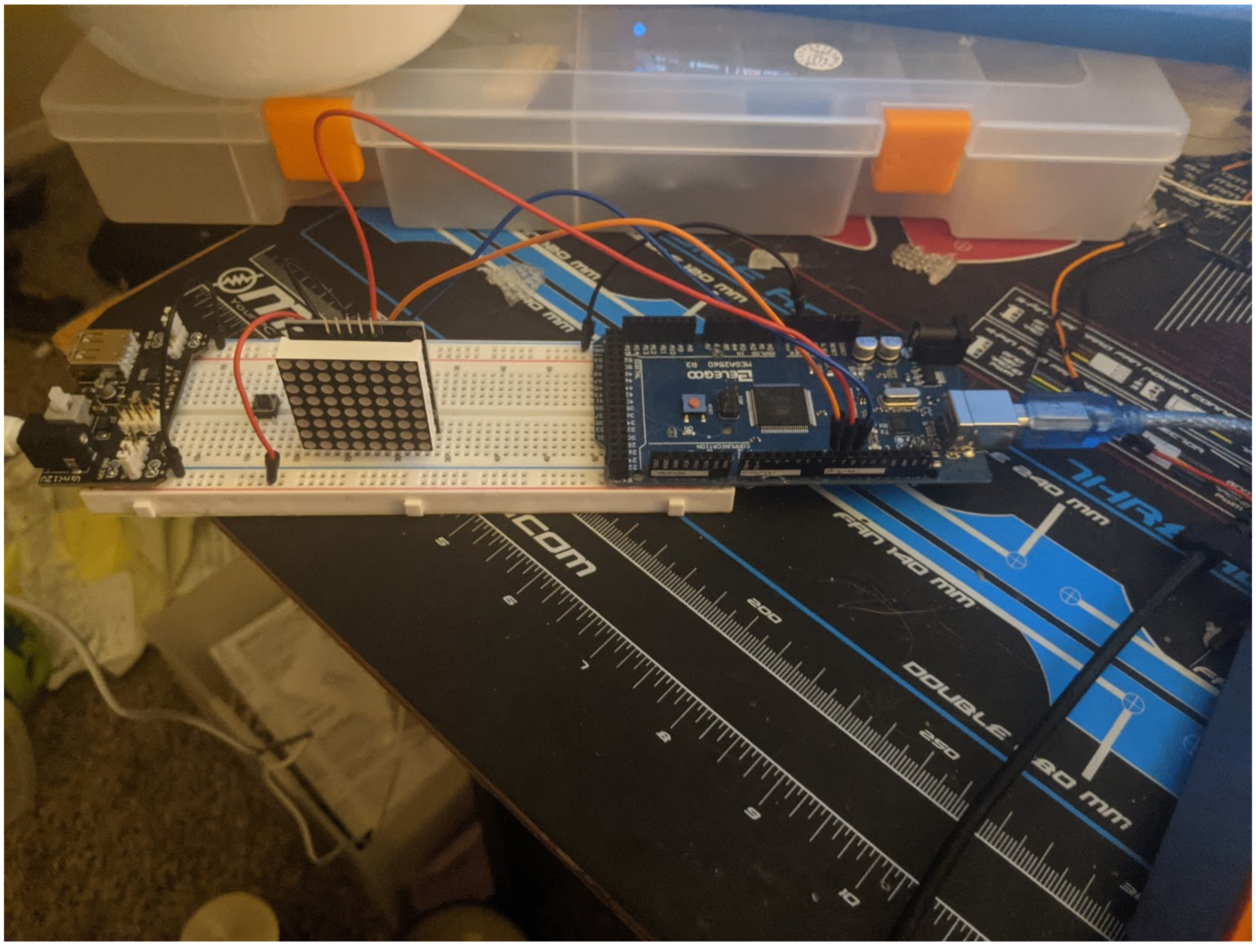
Lab 4 Report

USART & 8x8 LED Matrix

CENG 347 Embedded Intelligent Systems



Samuel Donovan & Samantha Pfeiffer

2021-02-21

TABLE OF CONTENTS

OVERVIEW

--- ATMega2560 USART ---

The ATMega2560 has four Universal Synchronous and Asynchronous serial Receiver and Transmitters, USART0, USART1, USART2, and USART3. USART0 also tied directly to the USB port on the board, and in this lab was used to receive and transmit data to another computer.

Each USART has two Baud Rate Registers (BRR), three Control and Status Registers (CSRA, CSRB, CSRC), and one Data Register (DR).

Following are descriptions of the USART's registers from the ATMega2560 Data sheet, sourced from the official ATmega640/1280/1281/2560/2561 datasheet which can be found at https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

-- DATA REGISTER (UDRn) --

Table 1 UDR Register Description

Bit	7	6	5	4	3	2	1	0
UDRn (Read)	RxB[7:0]							
UDRn (Write)	TxB[7:0]							
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

The USART data is transmitted from, and received into the same register, UDRn.

When writing data to the data register for transmission, it will first be copied into a shift register in the transmission circuit before it is sent out.

For 5-bit, 6-bit, or 7-bit frame sizes the upper unused bits will be ignored when transmitting, and set to zero when receiving.

For 9-bit frame sizes, the 9th bit should be written to **USRnB[0]**, and can be received at **USRnB[1]**.

The data register can only be written when Rx is enabled (**UCSRnA[4]**) .

Data written to the data register when Tx is enabled (**UCSRnB[3]**) will be transmitted as soon as possible.

-- CONTROL & STATUS REGISTER A (UCSRnA) --

Table 2 UCSRnA Register Description

Bit	7	6	5	4	3	2	1	0
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn
Read/Write	R	R/W	R	R	R	R	R/W	R/W
Initial Value	0	0	1	0	0	0	0	0

• Bit 7 – RXCn: USART Receive Complete
This flag bit is set when there is unread data in the data register.

• Bit 6 – TXCn: USART Transmit Complete
This flag bit is set once data written into the data register has been fully transmitted.

• Bit 5 – UDREn: USART Data Register Empty
This flag is set either once received data has been read from the data register, or data written to the register has been transferred to the transmission circuit shift register.

• Bit 4 – FEn: Frame Error
This bit is set if the next character in the receive buffer had a Frame Error when received, that is, when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDRn) is read. The FEn bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRnA.

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

• Bit 2 – UPEn: USART Parity Error
This bit is set if the next character in the receive buffer had a Parity Error when received, and Parity Checking was enabled at that point (**UCSnC[5:4]**). This bit is cleared when the data register is read. Always set this bit to zero when writing to UCSRnA.

• Bit 1 – U2Xn: Double USART Transmission Speed
This bit only has effect when in asynchronous operation (**UCSnC[7:6]**). Write this bit to zero when using synchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

• Bit 0 – MPCMn: Multi-processor Communication Mode
This bit enables the Multi-processor Communication mode. When the MPCMn bit is written to one, all the incoming frames received that do not contain address information will be ignored. Transmission is unaffected by this setting. MPCM must also be enabled in **UCSnC[7:6]**.

-- CONTROL & STATUS REGISTER B (UCSRnB) --

Table 3 UCSRnB - USART Control and Status Register B

Bit	7	6	5	4	3	2	1	0
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Initial Value	0	0	0	0	0	0	0	0

• Bit 7 – RXCIEn: RX Complete Interrupt Enable n
This bit enables the USARTn_RXC interrupt when the RXCn bit (**UCSnA[7]**) is 1. The global interrupt flag must also be set using `sei()`.

• Bit 6 – TXCIEn: TX Complete Interrupt Enable n
This bit enables the USARTn_TXC interrupt when the TXCn bit (**UCSnA[6]**) is 1. The global interrupt flag must also be set using `sei()`.

• Bit 5 – UDRIEn: USART Data Register Empty Interrupt Enable n
This bit enables the USARTn_DRE interrupt when the UDREn bit (**UCSnA[5]**) is 1. The global interrupt flag must also be set using `sei()`.

• Bit 4 – RXENn: Receiver Enable n
Writing this bit to one enables the USART Receiver.

• Bit 3 – TXENn: Transmitter Enable n
Writing this bit to one enables the USART Transmitter.

• Bit 2 – UCSZn2: Character Size n
The UCSZn2 bits combined with the UCSZn1:0 bits (**UCSRnC[2:1]**), determine the frame size for both Rx and Tx.

• Bit 1 – RXB8n: Receive Data Bit 8 n
The ninth data bit of the received data when operating with a frame size of 9-bits

Must be read before reading the low bits from UDRn.

• Bit 0 – TXB8n: Transmit Data Bit 8 n
The ninth data bit of the data to transmit when operating with a frame size of 9-bits. Must be written to before writing the low bits to UDRn

-- CONTROL & STATUS REGISTER C (UCSRnC) --

Table 4 UCSRnC - USART Control and Status Register C

Bit	7	6	5	4	3	2	1	0
UCSRnC	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Initial Value	0	0	0	0	0	1	1	0

• Bits 7:6 – UMSELn1:0 USART Mode Select

These bits select the mode of operation of the USARTn as shown in Table

Table 5 UMSELn Bits Settings

UMSELn1	UMSELn0	Mode
0	0	Async USART
0	1	Sync USART
1	0	(Reserved)
1	1	MSPIM

• Bits 5:4 – UPMn1:0: Parity Mode

These bits enable and set type of parity generation and check. This is handled automatically by the hardware. If data with bad parity is received, the UPEn Flag (**UCSRnA[2]**) will be set.

Table 6 UPMn Bits Settings

UPMn1	UPMLn0	Parity Mode
0	0	Disabled
0	1	(Reserved)
1	0	Enabled, Even
1	1	Enabled, Odd

• Bit 3 – USBSn: Stop Bit Select

This bit selects the number of stop bits to be inserted when transmitting, which is then handled by hardware.

Table 7 USBS Bits Settings

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

• Bit 2:1 – UCSZn1:0: Character Size

These bits combined with the UCSZn2 bit (**UCSRnB[2]**) sets the frame size for both Rx and Tx.

Table 8 UCSZn Bits Settings

UCSZn2	UCSZn1	UCSZn0	Character Size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	X	(Reserved)
1	1	0	(Reserved)
1	1	1	9-bit

• **Bit 0 – UCPOLn: Clock Polarity**
This bit is used for synchronous mode only, and should be 0 when in async mode. The UCPOLn bit sets the relationship between data output change, the data input sample, and the synchronous clock (XCKn pin).

Table 9 UCPOLn Bits Settings

UCPOLn	Output of TxDn Pin	Input on RXDn Pin
0	Rising XCKn Edge	Falling XCKn Edge
1	Falling XCKn Edge	Rising XCKn Edge

-- BAUD RATE REGISTER --

Table 10 UBRRnL and UBRRnH - USART Baud Rate Registers

Bit	15	14	13	12	11	10	9	8
UBRRHn	-	-	-	-	UBRR[11:8]			
UBRRLn	UBRR[7:0]							
Bit	7	6	5	4	3	2	1	0
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Values	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

• Bit 15:12 – Reserved Bits

These bits are reserved for future use. For compatibility with future devices, these bits should be written to zero when UBRRH is written to.

• Bit 11:0 – UBRR11:0: USART Baud Rate Register

UBRRH contains the four most significant bits, and UBRRL contains the eight least significant bits of the desired baud rate, as determined by **Equation 1**. Writing UBRRL will trigger an immediate update of the baud rate prescaler.
Ongoing USART operations will be corrupted when the baud rate is changed.
The following equation defines the value of UBRR, where ***F_CPU*** is the CPU clock speed, and ***BR*** is the baud rate in bauds (bits/sec). ***S*** is dependent on a few UCS bits, as outlined in the table.s

Equation 1 BRR Equation

$$\boxed{BBR = F_{CPU}/(S * BR) - 1}$$

Mode	Sync (**UCSnC[7:6]**)	Double Speed (**UCSnA[1]**)	Async (**UCSnC[7:6]**)
S	2	8	16

This page left intentionally blank.

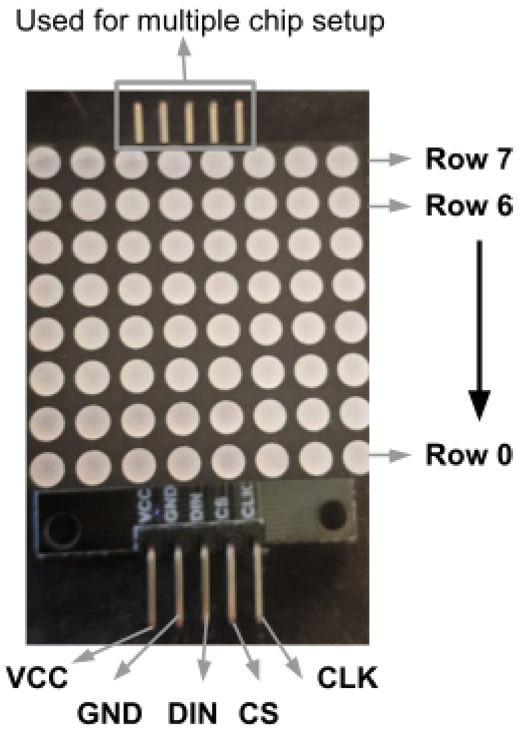
--- MAX7219 8x8 LED Matrix ---

The MAX7219 chip is prominently used to easily control upto 8 different seven-segment displays. Here, it has been repurposed to be used with an 8x8 LED matrix, with each different display being replaced with a row of 8 LEDs. Note that this is possible due to seven-segment displays actually having 8 LEDs including the decimal point (DP).

While the MAX7219 is designed in such a way that multiple of them can be used if connected sequentially, that use case will not be covered here.

All information in this section has been sourced from the official MAX7219/MAX7221 datasheet, which can be found at https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf

Figure 1 MAX7219 8x8 LED Matrix Diagram



-- PINS --

The MAX7219 mounted on the LED matrix has 5 input pins: **VCC**, **GND**, **DIN**, **CS**, AND **CLK**. **VCC** should be connected to +5V, and **GND** to 0V. The other three pins are data input pins.

-- SENDING DATA --

The MAX7219 takes in data through the using the **DIN**, **CS**, and **CLK** pins, in 16-bit frame sizes.

Table 11 shows the MAX7219 chip's expected 16-bit frame format: D15 through D12 are unused; D11 through D8 specifies the address for the data in D7 through D0 to be written to.

Table 11 MAX7219 Data Frame Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	ADDRESS				MSB			DATA				LSB

Frames are expected to be sent MSB first (D15 first, D0 last).

When sending a frame, the **CS** pin must be LOW (0V) for the duration of the write, and set back to HIGH (+5V) when finished.

For each bit sent, **CLK** must be set LOW (0V), **DIN** set to the bit's value (0 = LOW, 1 = HIGH), and then **CLK** set back to HIGH (+5V).

In summary, the pseudo code for sending a frame would be the following:

```
#define LOW 0
#define HIGH 1
byte address = 0bXXXX;
byte data = 0bXXXXXXXX;
short frame = (address << 8) & data;

//Start
CS.SetState(LOW);

//Send bit 15
CLK.SetState(LOW);
DIN.SetState((frame >> 15) & 1);
CLK.SetState(HIGH);
//Send bit 14
CLK.SetState(LOW);
DIN.SetState((frame >> 14) & 1);
CLK.SetState(HIGH);
...
...
//Send bit 0
CLK.SetState(LOW);
DIN.SetState((frame >> 0) & 1);
CLK.SetState(HIGH);

//Stop
CS.SetState(HIGH);
```

-- REGISTER DESCRIPTIONS --

Table 12 MAX7219 Register Descriptions

Name	Addr	(Hex)	Description	Valid Values
No-op	0000	(0x0)	No Operation	0x00 : 0xFF
Digit 0 thru 7	0001 thru 1000	(0x1) thru (0x8)	Controls Digit (Row) 0 thru 7	0x00 : 0xFF
Decode Mode	1001	(0x9)	Controls how digit registers are decoded (bit flags)	0x00 - No decoding 0x01 - BCD decode digit 0 0x02 - BCD decode digit 1 0x03 - BCD decode digit 1,0 … 0xFF - BCD decode all digits
Intensity	1010	(0xA)	Controls LED intensity (via PWM)	0x0 : 0xF (min : max)
Scan Limit	1011	(0xB)	Controls number of digits displayed	0x0 - Only digit 0 enabled 0x1 - Digits 0 & 1 enabled 0x2 - Digits 0,1,2 enabled 0x3 - Digits 0,1,2,3 enabled … 0x7 - All digits enabled
Shutdown	1100	(0xC)	Global LED switch	0x0 - OFF 0x1 - ON
Display Test	1111	(0xF)	Used for testing (turns on all LEDs) (does not overwrite)	0x0 - Normal Operation 0x1 - Display Test Mode

This page left intentionally blank

METHODS

--- CODE ---

To simplify and modularize our code, we created 5 different c files, each with their own responsibility, denoted by their file name. These modules were coded to be reusable, in case we should need their functionality for future projects.

Ideally this modularization would have been able to utilize object-oriented programming (OOP), but due to the limitations of C, this was emulated by creating functions where the first input is the utilized structure, passed by reference. It should be noted this is similar to how OOP works at an assembly level.

Every function is prefixed by the data type it expects as it’s first argument.

In these files you will commonly see `reg*` (reg star) data types. These are equivalent to `volatile unsigned char`, and represent a reference to a register.

Each module was manually tested independently prior to using them together.

Lab4.ino

See **Appendix A (source code)**.

Here exists the main code responsible for functionality, which utilizes all of the other abstracted code.

Owing to their legibility, `serialSetup` and `max7219Setup` will not be explained.

A character buffer (array) is created to temporarily hold a number of characters between the time when they are received and when they are displayed. The buffer is treated as a circular array: whenever it's start or end indices are modified, they are looped back to 0 if they exceed the buffer size.

The buffer is constantly attempted to be read from while it's not empty. When read from, the display is briefly cleared before the character is written to it. Following this operation, the character is "removed" from the buffer by moving the start index ahead of its position.

When Rx is received on USART 0, an interrupt service routine is run. If the character is alphanumeric, it is added to the end of the buffer, and the buffer's end index is incremented.

🔶It is possible that the buffer may overflow if too many characters are received before they can all be displayed. This limit is defined by `BUFFER_SIZE`, and exceeding it will result in undefined behaviour.

samsUSART.h

See **Appendix B (source code)** and Section ATMega2560 USART.

The `USART` structure represents a set of registers used to control a single USART port.

🔶This file includes definitions for USART 0 through USART 4, which may cause issues if programming a board with less USART ports.

Most functions serve to easily interface with the Control and Status registers. There are numerous macros defined in this file to make setting states easier. Please see the source code in **Appendix B**.

🔶The functions in this file were written in accordance the Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V documentation, and may not work for other boards.

`USART_SetBaudRate` sets the UBRRH and UBRRL registers in accordance with **Equation 1**. At run-time, it checks the proper bits to see if the USART port is in synchronous or double transmission speed mode, and calculates a different value accordingly.

🔶`USART_SetBaudRate` requires that `F_CPU` be defined.

`USART_Write` and `USART_Read` are basically aliases for the USART's Data Register.

samsGPIO.h

See Appendix C (source code).

This module is sourced from our code in previous labs, and is used to easily interface with GPIO ports.

The `GPIO` structure represents a set of registers used to control a single GPIO pin. As there are a large number of ports, none of them are predefined in this file, you will have to define them yourself.

`LOW`, `HIGH`, `INPUT`, and `OUTPUT` macros have been conditionally defined in this file, should you be using an IDE that does not define them.

`GPIO_SetMode` should be called before attempting to read/write using `GPIO_SetState`/`GPIO_GetState`.

samsMAX7219.h

See **Appendix D (source code) and** Section MAX7219**.**

This module depends on **samsGPIO.h**

The `MAX7219` structure represents three GPIO pins used to control a MAX7219 chip.

Many functions in this file depend on `MAX7219_SendData` function, and are essentially just aliases for it. For example, `MAX7219_StartTest` simply sends ‘1’ using the Display Test Register address.

`MAX7219_SendData` is an alias to setting every digit’s LEDs off, as well as clearing the Display Test register.

All of the different registers’ addresses have been defined in macros. See the source code.

samsMatrixDisplay.h

See **Appendix E (source code).**

This file depends on **samsMAX7219.h** and **MatrixDisplayCharLUT.h**.

This file acts to further abstract the MAX7219 chip, specifically for an 8x8 LED matrix. Although the functions are prefixed with `MATRIX`, please note they actually expect a `MAX721` struct.

`MATRIX_SetRow` is an alias to `MAX7219_SendData`. `row` is 0-indexed. Please note that the authors are aware that this technically sets the columns of the matrix.

`MATRIX_WriteChar` looks up the desired character in the lookup table found in **MatrixDisplayCharLUT.h**, and calls `MATRIX_SetRow` with the appropriate values. Should the character not be alphanumeric, instead the ':(' entry will be used.

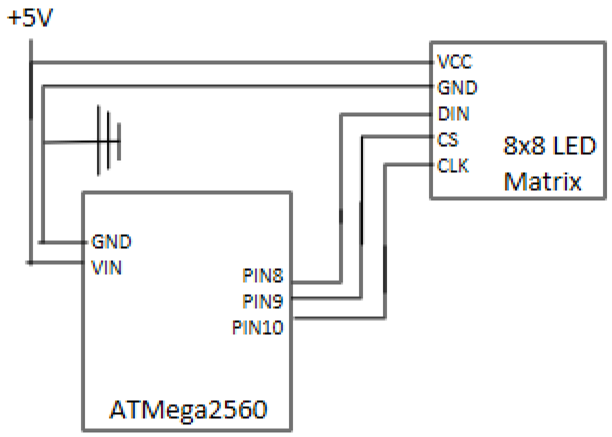
MatrixDisplayCharLUT.h

This file merely holds a large 2-D array of bytes, used by **samsMatrixDisplay.h** to translate characters into bit masks for each of the eight rows on the LED Matrix. It can be found in **Appendix F.**

--- CIRCUITRY ---

The circuitry for this lab is outlined in the circuit diagram below. DIN was connected to PINH5 (Pin 8), CS was connected to PORTH6 (Pin 9), and CLK was connected to PINB4 (Pin 10)

Figure 2 ATMega to 8x8 LED Matrix Circuit Diagram



ISSUES ENCOUNTERED

There were several minor issues encountered in this lab.

The first issue encountered was that only the first row of the matrix would light up when attempting to print a character. Initially, it was thought that the lights had burnt out, however `StartTest` allowed us to illuminate the entire matrix and disprove that. It was then thought that there was an issue with the `WriteChar` function, though none could be found. Through testing, the issue was found to be that `SetScanLimit` needed to be called and set to 8. This was then added to the `max7219Setup` function.

Another issue occurred when trying to set the frame size to 8. In conflict with what the ATMega datasheet denotes, `SetFrameSize` was attempting to simply write the binary value of the frame size to the UCSZ register. This was remedied by first subtracting 5 from the frame size before setting UCSZ to the value. A special case was also included for a frame size of 9 as was necessitated.

The final issues pertained to the character table. In order to display that a non-alphanumeric value was transmitted, a frowny face was added to the end of the character table. Since the table was initialized to have 38 rows, it was assumed that the size needed to be increased to 39, allowing the error character to be placed at index 38. However, trying to display this character caused the matrix to turn off. After counting the entries in the table, it was realized that there were only 36 characters, so the added error character was actually at index 36. The `WriteChar` function and the size of the table were then updated to fix this error.

Also, it was discovered that the ‘1’ character was backwards. This was fixed by updating the hexadecimal values in the character table.

RESULTS

After fixing the issues that occured, the device worked exactly as expected. For Part 1, we were able to display 0 to 9 and A to Z. Then, in Part 2 we were able to display strings from the terminal. Overall, this lab was a success.

APPENDIX A: Lab4.ino

```
#include "samsUSART.h"
#include "samsMAX7219.h"
#include "samsMatrixDisplay.h"
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>


MAX7219 led =
{
    .DIN = {.DDR=&DDRH, .PORT_OUT=&PORTH, .PORT_IN=&PINH,
.PIN=5}, //Pin 8
    .CS = {.DDR=&DDRH, .PORT_OUT=&PORTH, .PORT_IN=&PINH,
.PIN=6},  //Pin 9
    .CLK = {.DDR=&DDRB, .PORT_OUT=&PORTB, .PORT_IN=&PINB,
.PIN=4} //Pin 10
};

void max7219Setup(){
    MAX7219_SetupGPIO(led);
    MAX7219_TurnOn(led);
    MAX7219_Clear(led);
    MAX7219_SetScanLimit(led, 8);
}

void serialSetup()
{
    USART_SetBaudRate(USART0, 9600);
    USART_SetFrameSize(USART0, 8);
    USART_SetStopBits(USART0, USART_ONE_STOP_BIT);
    USART_SetParity(USART0, USART_PARITY_DISABLED);
    USART_EnableRxCompleteInterrupt(USART0);
    USART_EnableRx(USART0);
    USART_EnableTx(USART0);
}


#define BUFFER_SIZE 80
```

```
char buffer[BUFFER_SIZE];
volatile int bufferStart = 0;
volatile int bufferEnd = 0;

ISR(USART0_RX_vect)
{
    char c = USART_Read(USART0);
    USART_Write(USART0, c); //Echo
    if(isalpha(c) || isdigit(c))
    {
        buffer[bufferEnd] = c; //Store
        bufferEnd++; //Increment end point
        if(bufferEnd == BUFFER_SIZE) //Loop
            bufferEnd = 0;
    }
}

int main()
{
    max7219Setup();
    serialSetup();
    sei();

    while(1)
    {
        while(bufferStart != bufferEnd)
        {
            MAX7219_Clear(led);
            _delay_ms(200);
            MATRIX_WriteChar(led, buffer[bufferStart]); //Wite
to LED

            bufferStart++; //Increment start point
            if(bufferStart == BUFFER_SIZE) //Loop
                bufferStart = 0;

            _delay_ms(1000);
        }
    }
}
```

APPENDIX B: samsUSART.h

```
#ifndef SAMS_SERIAL
#define SAMS_SERIAL

#include <avr/io.h>
typedef volatile unsigned char reg;

struct USART
{
    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]]
};

//USART Definitions//
USART USART0 = {.BRRL=&UBRR0L, .BRRH=&UBRR0H, .CSRA=&UCSR0A,
.CSRB=&UCSR0B, .CSRC=&UCSR0C, .DR=&UDR0};
USART USART1 = {.BRRL=&UBRR1L, .BRRH=&UBRR1H, .CSRA=&UCSR1A,
.CSRB=&UCSR1B, .CSRC=&UCSR1C, .DR=&UDR1};
USART USART2 = {.BRRL=&UBRR2L, .BRRH=&UBRR2H, .CSRA=&UCSR2A,
.CSRB=&UCSR2B, .CSRC=&UCSR2C, .DR=&UDR2};
USART USART3 = {.BRRL=&UBRR3L, .BRRH=&UBRR3H, .CSRA=&UCSR3A,
.CSRB=&UCSR3B, .CSRC=&UCSR3C, .DR=&UDR3};

//Function Prototypes//
bool USART_RxComplete(USART &U); //RxC (R)
bool USART_TxComplete(USART &U);  //TxC (R)
void USART_SetTxComplete(USART &U); //TxC (W 1)
void USART_ClearTxComplete(USART &U); //TxC (W 0)
bool USART_DataRegisterEmpty(USART &U); //UDRE (R)
bool USART_FrameError(USART &U); //FE (R)
bool USART_DataOverrunError(USART &U); //DOR (R)
bool USART_ParityError(USART &U); //PE (R)
bool USART_DoubleTransmissionSpeedIsEnabled(USART &U); //U2X
(R)
void USART_EnableDoubleTransmissionSpeed(USART &U); //U2X (W 1)
void USART_DisableDoubleTransmissionSpeed(USART &U); //U2X (W
0)
bool USART_MultiProcessorCommunicationModeIsEnabled(USART &U);
//MPCM (R)
void USART_EnableMultiProcessorCommunicationMode(USART &U);
//MPCM (W 1)
void USART_DisableMultiProcessorCommunicationMode(USART &U);
//MPCM (W 0)

//UCSRB//
bool USART_RxCompleteInterruptIsEnabled(USART &U); //RXCIE (R)
void USART_EnableRxCompleteInterrupt(USART &U); //RXCIE (W 1)
void USART_DisableRxCompleteInterrupt(USART &U); //RXCIE (W 0)
bool USART_TxCompleteInterruptIsEnabled(USART &U); //TXCIE (R)
void USART_EnableTxCompleteInterrupt(USART &U); //TXCIE (W 1)
void USART_DisableTxCompleteInterrupt(USART &U); //TXCIE (W 0)
bool USART_DataRegisterEmptyInterruptIsEnabled(USART &U);
//UDRIE (R)
void USART_EnableDataRegisterEmptyInterrupt(USART &U); //UDRIE
(W 1)
void USART_DisableDataRegisterEmptyInterrupt(USART &U); //UDRIE
(W 0)
bool USART_RxIsEnabled(USART &U); //RXEN (R)
void USART_EnableRx(USART &U); //RXEN (W 1)
void USART_DisableRx(USART &U); //RXEN (W 0)
bool USART_TxIsEnabled(USART &U); //TXEN (R)
void USART_EnableTx(USART &U); //TXEN (W 1)
void USART_DisableTx(USART &U); //TXEN (W 0)
byte USART_RxBit9(USART &U); //RXB8 (R)
byte USART_TxBit9(USART &U); //TXB8 (R)
void USART_SetTxBit9(USART &U); //TXB8 (W 1)
void USART_ClearTxBit9(USART &U); //TXB8 (W 0)

//UCSRC//
#define USART_ASYNC 0b00
#define USART_SYNC  0b01
#define USART_MSPIM 0b11
byte USART_GetMode(USART &U); //UMSEL (R)
void USART_SetMode(USART &U, byte mode=USART_ASYNC); //UMSEL
```

```
(W)

#define USART_DISABLED 0b00
#define USART_RESERVED 0b01
#define USART_EVEN     0b10
#define USART_ODD      0b11
byte USART_GetParity(USART &U); //UPM (R)
void USART_SetParity(USART &U, byte mode=USART_DISABLED); //UPM
(W)

#define USART_ONE_STOP_BIT 0b0
#define USART_TWO_STOP_BIT 0b1
byte USART_GetStopBits(USART &U); //USBS (R)
void USART_SetStopBits(USART &U, byte size=USART_ONE_STOP_BIT);
//USBS (W)

#define USART_FIVE_BIT_FRAME 0b000
#define USART_SIX_BIT_FRAME 0b001
#define USART_SEVEN_BIT_FRAME 0b010
#define USART_EIGHT_BIT_FRAME 0b011
#define USART_NINE_BIT_FRAME 0b111
byte USART_GetFrameSize(USART &U); //UCSZ (R)
void USART_SetFrameSize(USART &U, byte
size=USART_EIGHT_BIT_FRAME); //UCSZ (W) size must be between 5
and 9

#define USART_RISING_EDGE 0
#define USART_FALLING_EDGE 1
byte USART_GetClockPolarity(USART &U);
void USART_SetClockPolarity(USART &U, byte
polarity=USART_RISING_EDGE);

//Core Funcs
void USART_SetBaudRate(USART &U, unsigned long BAUD_RATE);
void USART_Write(USART &U, byte data);
byte USART_Read(USART &U);




//Function Implementations//
//UCSRA//
```

```
//RxC (R)
bool USART_RxComplete(USART &U)
{
    return *U.CSRA & 0b10000000; // != 0
}

//TxC (W/R)
bool USART_TxComplete(USART &U)
{
    return *U.CSRA & 0b01000000; // != 0
}
void USART_SetTxComplete(USART &U)
{
    *U.CSRA |= 0b01000000;
}
void USART_ClearTxComplete(USART &U)
{
    *U.CSRA &= 0b10111111;
}

//UDRE (R)
bool USART_DataRegisterEmpty(USART &U)
{
    return *U.CSRA & 0b00100000; //!= 0
}

//FE (R)
bool USART_FrameError(USART &U)
{
    return *U.CSRA & 0b00010000; //!= 0
}

//DOR (R)
bool USART_DataOverrunError(USART &U)
{
    return *U.CSRA & 0b00001000; //!= 0
}

//PE (R)
bool USART_ParityError(USART &U)
{
    return *U.CSRA & 0b00000100; //!= 0
```

```
}

//U2X (R/W)
bool USART_DoubleTransmissionSpeedIsEnabled(USART &U)
{
    return *U.CSRA & 0b00000010; //!= 0
}
void USART_EnableDoubleTransmissionSpeed(USART &U)
{
    *U.CSRA |= 0b00000010;
}
void USART_DisableDoubleTransmissionSpeed(USART &U)
{
    *U.CSRA &= 0b11111101;
}

//MPCM (R/W)
bool USART_MultiProcessorCommunicationModeIsEnabled(USART &U)
{
    return *U.CSRA & 0b00000001;
}
void USART_EnableMultiProcessorCommunicationMode(USART &U)
{
    *U.CSRA |= 0b00000001;
}
void USART_DisableMultiProcessorCommunicationMode(USART &U)
{
    *U.CSRA &= 0b11111110;
}




//UCSRB//
//RXCIE (R/W)
bool USART_RxCompleteInterruptIsEnabled(USART &U)
{
    return *U.CSRB & 0b10000000; //!= 0
}
void USART_EnableRxCompleteInterrupt(USART &U)
{
   *U.CSRB |= 0b10000000;
```

```
}
void USART_DisableRxCompleteInterrupt(USART &U)
{
    *U.CSRB &= 0b01111111;
}

//TXCIE (R/W)
bool USART_TxCompleteInterruptIsEnabled(USART &U)
{
    return *U.CSRB & 0b01000000; //!= 0
}
void USART_EnableTxCompleteInterrupt(USART &U)
{
    *U.CSRB |= 0b01000000;
}
void USART_DisableTxCompleteInterrupt(USART &U)
{
    *U.CSRB &= 0b10111111;
}

//UDRIE (R/W)
bool USART_DataRegisterEmptyInterruptIsEnabled(USART &U)
{
    return *U.CSRB & 0b00100000; //!= 0
}
void USART_EnableDataRegisterEmptyInterrupt(USART &U)
{
    *U.CSRB |= 0b00100000;
}
void USART_DisableDataRegisterEmptyInterrupt(USART &U)
{
    *U.CSRB &= 0b11011111;
}

//RXEN (R/W)
bool USART_RxIsEnabled(USART &U)
{
    return *U.CSRB & 0b00010000; //!= 0
}
void USART_EnableRx(USART &U)
{
    *U.CSRB |= 0b00010000;
```

```
}
void USART_DisableRx(USART &U)
{
    *U.CSRB &= 0b11101111;
}

//TXEN (R/W)
bool USART_TxIsEnabled(USART &U)
{
    return *U.CSRB & 0b00001000; //!= 0
}
void USART_EnableTx(USART &U)
{
    *U.CSRB |= 0b00001000;
}
void USART_DisableTx(USART &U)
{
    *U.CSRB &= 0b11110111;
}

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

//RXB8 (R)
byte USART_RxBit9(USART &U)
{
    return (*U.CSRB & 0b00000010) >> 1;
}
byte USART_TxBit9(USART &U)
{
    return *U.CSRB & 0b00000001;
}
void USART_SetTxBit9(USART &U)
{
    *U.CSRB |= 0b00000001;
}
void USART_ClearTxBit9(USART &U)
{
    *U.CSRB &= 0b11111110;
}
```

```
//UCSRC//
//UMSEL (R/W)
byte USART_GetMode(USART &U)
{
    return (*U.CSRC & 0b11000000) >> 6;
}
void USART_SetMode(USART &U, byte mode)
{
    *U.CSRC &= 0b00111111; //Clear
    *U.CSRC |= (mode & 0b11) << 6; //Set
}

//UPM (R/W)
byte USART_GetParity(USART &U)
{
    return (*U.CSRC & 0b00110000) >> 4;
}
void USART_SetParity(USART &U, byte mode)
{
    *U.CSRC &= 0b11001111; //Clear
    *U.CSRC |= (mode & 0b11) << 4; //Set
}

//USBS (R/W)
byte USART_GetStopBits(USART &U)
{
    return (*U.CSRC & 0b00001000) >> 3;
}
void USART_SetStopBits(USART &U, byte size)
{
    *U.CSRC &= 0b11110111; //Clear
    *U.CSRC |= (size & 0b1) << 3; //Set
}

//UCSZ (R/W)
byte USART_GetFrameSize(USART &U)
{
    return (*U.CSRB & 0b00000100) + ((*U.CSRC & 0b00000110) >>
1);
}
void USART_SetFrameSize(USART &U, byte size) //size can be 5-9
bits
```

```
{
    *U.CSRC &= 0b11111001; //Clear UCSZ 0 & 1
    *U.CSRB &= 0b11111011; //Clear UCSZ 2

    if(size == 9)
        size = 0b111;
    else
        size = size - 5;

    *U.CSRC |= (size & 0b011) << 1; //Set UCSZ 0 & 1
    *U.CSRC |= (size & 0b100); //Set UCSZ 2
}

//UCPOL
byte USART_GetClockPolarity(USART &U)
{
    return *U.CSRC & 0b00000001;
}
void USART_SetClockPolarity(USART &U, byte polarity)
{
    *U.CSRC &= 0b11111110; //Clear
    *U.CSRC |= polarity & 0b1; //Set
}


//Core Funcs
void USART_SetBaudRate(USART &U, unsigned long baud_rate)
{
    unsigned short val = 0;

    if(USART_GetMode(U) == USART_SYNC) //SYNC
        val = F_CPU/(2UL*baud_rate)-1;
    else if(USART_DoubleTransmissionSpeedIsEnabled(U)) //Double
ASYNC
        val = F_CPU/(8UL*baud_rate)-1;
    else //ASYNC
        val = F_CPU/(16UL*baud_rate)-1;

    *U.BRRH = (val >> 8) & 0b00001111; //Set URBBH
    *U.BRRL = val; //Set UBRRL
}
```

```
void USART_Write(USART &U, byte data)
{
    *U.DR = data;
}
byte USART_Read(USART &U)
{
    return *U.DR;
}
#endif
```

APPENDIX C: samsGPIO.h

```
#ifndef SAMS_GPIO
#define SAMS_GPIO
typedef volatile unsigned char reg;

struct GPIO
{
  reg* DDR;
  reg* PORT_OUT;
  reg* PORT_IN;
  byte PIN;
};

#ifndef INPUT
#define INPUT 0
#endif
#ifndef OUTPUT
#define OUTPUT 1
#endif
//Functions
void GPIO_SetMode(GPIO &GPIO, byte MODE)
{
  if(MODE == INPUT)
    *GPIO.DDR &= ~(1 << GPIO.PIN);
  else
    *GPIO.DDR |= 1 << GPIO.PIN;
}
```

```
#ifndef LOW
#define LOW 0
#endif
#ifndef HIGH
#define HIGH 1
#endif

void GPIO_SetState(GPIO &GPIO, byte STATE)
{
  if(STATE == LOW)
    *GPIO.PORT_OUT &= ~(1 << GPIO.PIN);
  else
    *GPIO.PORT_OUT |= 1 << GPIO.PIN;
}

byte GPIO_GetState(GPIO &GPIO)
{
  if(*GPIO.PORT_IN >> GPIO.PIN)
    return HIGH;
  else
    return LOW;
}
#endif
```

APPENDIX D: samsMAX7219.h

```
#ifndef SAMS_MAX7219
#define SAMS_MAX7219
#include "samsGPIO.h"

struct MAX7219
{
    GPIO DIN;
    GPIO CS;
    GPIO CLK;
};

//Core Functions//
void MAX7219_SetupGPIO(MAX7219 &disp);
void MAX7219_SendMSB(MAX7219 &disp, byte b);
void MAX7219_SendByte(MAX7219 &disp, byte b);
void MAX7219_SendData(MAX7219 &disp, byte addr, byte data);
void MAX7219_Clear(MAX7219 &disp);
//General Functions//
void MAX7219_TurnOff(MAX7219 &disp);
void MAX7219_TurnOn(MAX7219 &disp);
#define MAX7219_BCD_DECODING 0b11111111
#define MAX7219_NO_DECODING  0b00000000
void MAX7219_SetDecoding(MAX7219 &disp, byte
mode=MAX7219_NO_DECODING);
void MAX7219_SetIntensity(MAX7219 &disp, byte intensity=0);
//intensity = [0, 15]
void MAX7219_SetScanLimit(MAX7219 &disp, byte nOutputs=8);
//nOutputs = [1, 8]
void MAX7219_StartTest(MAX7219 &disp);
void MAX7219_StopTest(MAX7219 &disp);


//Addresses
#define MAX7219_NO_OP_ADDR 0b0000
#define MAX7219_DIGIT_0_ADDR 0b0001
#define MAX7219_DIGIT_1_ADDR 0b0010
#define MAX7219_DIGIT_2_ADDR 0b0011
#define MAX7219_DIGIT_3_ADDR 0b0100
```

```
#define MAX7219_DIGIT_4_ADDR 0b0101
#define MAX7219_DIGIT_5_ADDR 0b0110
#define MAX7219_DIGIT_6_ADDR 0b0111
#define MAX7219_DIGIT_7_ADDR 0b1000
#define MAX7219_DECODE_MODE_ADDR 0b1001
#define MAX7219_INTENSITY_ADDR 0b1010
#define MAX7219_SCAN_LIMIT_ADDR 0b1011
#define MAX7219_SHUTDOWN_ADDR 0b1100
#define MAX7219_DISPLAY_TEST_ADDR 0b1111

//Core Functions//
void MAX7219_SetupGPIO(MAX7219 &disp)
{
     GPIO_SetMode(disp.DIN, OUTPUT);
     GPIO_SetMode(disp.CS, OUTPUT);
     GPIO_SetMode(disp.CLK, OUTPUT);
}

void MAX7219_SendMSB(MAX7219 &disp, byte b)
{
     GPIO_SetState(disp.CLK, 0);
     GPIO_SetState(disp.DIN, (b >> 7));
     GPIO_SetState(disp.CLK, 1);
}

void MAX7219_SendByte(MAX7219 &disp, byte b)
{
     MAX7219_SendMSB(disp, b << 0); //7th bit
    MAX7219_SendMSB(disp, b << 1); //6th bit
    MAX7219_SendMSB(disp, b << 2); //5th bit
    MAX7219_SendMSB(disp, b << 3); //4th bit
    MAX7219_SendMSB(disp, b << 4); //3th bit
    MAX7219_SendMSB(disp, b << 5); //2th bit
    MAX7219_SendMSB(disp, b << 6); //1th bit
    MAX7219_SendMSB(disp, b << 7); //0th bit
}

void MAX7219_SendData(MAX7219 &disp, byte addr, byte data)
{
    GPIO_SetState(disp.CS, 0);
     MAX7219_SendByte(disp, addr);
     MAX7219_SendByte(disp, data);
     GPIO_SetState(disp.CS, 1);
}

void MAX7219_Clear(MAX7219 &disp)
{
     MAX7219_SendData(disp, MAX7219_DIGIT_0_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_1_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_2_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_3_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_4_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_5_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_6_ADDR, 0);
     MAX7219_SendData(disp, MAX7219_DIGIT_7_ADDR, 0);
    MAX7219_StopTest(disp);
}

//General Functions//
//Shutdown Register//
void MAX7219_TurnOff(MAX7219 &disp)
{
     MAX7219_SendData(disp, MAX7219_SHUTDOWN_ADDR, 0b00000000);
}
void MAX7219_TurnOn(MAX7219 &disp)
{
     MAX7219_SendData(disp, MAX7219_SHUTDOWN_ADDR, 0b00000001);
}

//Decode Mode Register//
void MAX7219_SetDecoding(MAX7219 &disp, byte mode)
{
     MAX7219_SendData(disp, MAX7219_DECODE_MODE_ADDR, mode);
}

//Intensity Register//
void MAX7219_SetIntensity(MAX7219 &disp, byte intensity)
//intensity = [0, 15]
{
     MAX7219_SendData(disp, MAX7219_INTENSITY_ADDR, intensity);
}

//Scan Limit Register// (Sets how many outputs are
disped/written to)
```

```
void MAX7219_SetScanLimit(MAX7219 &disp, byte nOutputs)
//nOutputs = [1, 8]
{
     nOutputs--;
     MAX7219_SendData(disp, MAX7219_SCAN_LIMIT_ADDR, nOutputs);
}

void MAX7219_StartTest(MAX7219 &disp)
{
     MAX7219_SendData(disp, MAX7219_DISPLAY_TEST_ADDR, 1);
}

void MAX7219_StopTest(MAX7219 &disp)
{
     MAX7219_SendData(disp, MAX7219_DISPLAY_TEST_ADDR, 0);
}

#endif
```

APPENDIX E: samsMatrixDisplay.h

```
#ifndef SAMS_MATRIX_DISPLAY
#define SAMS_MATRIX_DISPLAY
#include "samsMAX7219.h"
#include "MatrixDisplayCharLUT.h"


void SetRow(MAX7219 &disp, byte row, byte data);
void WriteChar(MAX7219 &disp, unsigned char x);


void SetRow(MAX7219 &disp, byte row, byte data)
{
  MAX7219_SendData(disp, row+1, data);
}

void WriteChar(MAX7219 &disp, unsigned char x)
{
  int loc = 0;
  int i = 0;
  if(isalpha(x))
  {
    x = toupper(x);
    loc = x - 55;
  }
  else if(isdigit(x))
  {
    loc = x - 48;
  }
  else
  {
    loc = 36;
  }


  for(i = 0; i < 8; i++)
    SetRow(disp, i, CharTable[loc][i]);
}
```

```
#endif
```

APPENDIX F: MatrixDisplayCharLUT.h

```
#ifndef CHARLUT
#define CHARLUT

unsigned char CharTable[37][8]={
{0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x3C},//0
{0x10,0x30,0x50,0x10,0x10,0x10,0x10,0x7C},//1
{0x7E,0x2,0x2,0x7E,0x40,0x40,0x40,0x7E},//2
{0x3E,0x2,0x2,0x3E,0x2,0x2,0x3E,0x0},//3
{0x8,0x18,0x28,0x48,0xFE,0x8,0x8,0x8},//4
{0x3C,0x20,0x20,0x3C,0x4,0x4,0x3C,0x0},//5
{0x3C,0x20,0x20,0x3C,0x24,0x24,0x3C,0x0},//6
{0x3E,0x22,0x4,0x8,0x8,0x8,0x8,0x8},//7
{0x0,0x3E,0x22,0x22,0x3E,0x22,0x22,0x3E},//8
{0x3E,0x22,0x22,0x3E,0x2,0x2,0x2,0x3E},//9
{0x8,0x14,0x22,0x3E,0x22,0x22,0x22,0x22},//A
{0x3C,0x22,0x22,0x3E,0x22,0x22,0x3C,0x0},//B
{0x3C,0x40,0x40,0x40,0x40,0x40,0x3C,0x0},//C
{0x7C,0x42,0x42,0x42,0x42,0x42,0x7C,0x0},//D
{0x7C,0x40,0x40,0x7C,0x40,0x40,0x40,0x7C},//E
{0x7C,0x40,0x40,0x7C,0x40,0x40,0x40,0x40},//F
{0x3C,0x40,0x40,0x40,0x40,0x44,0x44,0x3C},//G
{0x44,0x44,0x44,0x7C,0x44,0x44,0x44,0x44},//H
{0x7C,0x10,0x10,0x10,0x10,0x10,0x10,0x7C},//I
{0x3C,0x8,0x8,0x8,0x8,0x8,0x48,0x30},//J
{0x0,0x24,0x28,0x30,0x20,0x30,0x28,0x24},//K
{0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x7C},//L
{0x81,0xC3,0xA5,0x99,0x81,0x81,0x81,0x81},//M
{0x0,0x42,0x62,0x52,0x4A,0x46,0x42,0x0},//N
{0x3C,0x42,0x42,0x42,0x42,0x42,0x42,0x3C},//O
{0x3C,0x22,0x22,0x22,0x3C,0x20,0x20,0x20},//P
{0x1C,0x22,0x22,0x22,0x22,0x26,0x22,0x1D},//Q
{0x3C,0x22,0x22,0x22,0x3C,0x24,0x22,0x21},//R
{0x0,0x1E,0x20,0x20,0x3E,0x2,0x2,0x3C},//S
{0x0,0x3E,0x8,0x8,0x8,0x8,0x8,0x8},//T
{0x42,0x42,0x42,0x42,0x42,0x42,0x22,0x1C},//U
{0x42,0x42,0x42,0x42,0x42,0x42,0x24,0x18},//V
```

```
{0x0,0x49,0x49,0x49,0x49,0x2A,0x1C,0x0},//W
{0x0,0x41,0x22,0x14,0x8,0x14,0x22,0x41},//X
{0x41,0x22,0x14,0x8,0x8,0x8,0x8,0x8},//Y
{0x0,0x7F,0x2,0x4,0x8,0x10,0x20,0x7F},//Z
{0x0,0x24,0x24,0x24,0x0,0x3C,0x42,0x42} //:(
};

#endif
```