This write up will build a program that translates ASCII characters from the serial port into Morse code dots and dashes. In the process of developing this program, three basic programming constructs will be demonstrated, the look-up table (for data translation), the rolling buffer (for holding data until processed) and the Finite State Machine.

Morse code is a series of short and long sounds or lights, that have been defined to mean certain letters. A chart showing the basic Morse code sequences is in Figure 1. Note a dot (short) is approximately 50 milliseconds long, while a dash is three times the length of a dot, and the space between dots and dashes is the same as a dot. Also a space (no light) of three times the length of a dot is the space between characters. One of our challenges is going to be getting the computer to translate the ASCII characters coming in into a set of dots and dashes.

A •–	J •–––	S •••	1 •———–
B –•••	K –•–	T –	2 ••–––
C –•–•	L •–••	U ••–	3 •••––
D –••	M ––	V •••–	4 ••••–
E •	N –•	W •––	5 •••••
F ••–•	O –––	X –••–	6 –••••
G ––•	P •––•	Y –•––	7 ––•••
H ••••	Q ––•–	Z ––••	8 –––••
I ••	R •–•	0 –––––	9 ––––•

Figure 1. Morse Code Encoding

So as an ASCII character comes in, we will need to translate it into a pattern of 1 and 0's that matches the on-off pattern for its Morse code equivalent. The most effective way to perform this kind of a translation, where no real equations exists, is to use a look-up table, but what do we put into the table.

Well in this case we will want to encode the on and off patterns of Morse code into ones (on) and zeros (off), so consider the letter A, which is a dot dash. If we think of each bit as representing a 50 millisecond interval, the dot, space, dash would be 1 0111. But it might work better if we reverse the order of the bits in our encoding and shift the number to the right each step. Thus I encoded the A as the bit pattern 0001 1101, or 0x1D. Now a table of these translations can be seen in the code in Appendix A, and they are placed in sequential locations in the array. So if we have an A to translate we will need to access element 0, and use it to transmit. Similarly, B's translation is in element 1, thus to access the correct element in this table we can use the character in ASCII minus the ASCII character A or

```
TxData = MorseCodeAlpha[Incoming - 'A'];
```

Now it will take about 250 milliseconds to send an A in morse code, but serial-ASCII data comes in at a rate of about 1 millisecond per character. A rolling buffer is an effective way to hold data that may be coming in bursts, but needs to be pulled out and processed in a sequential fashion. A rolling buffer consists of an array of size N, and two integer pointers that indicate the location in the array into which

is to be inserted and extracted. This arrangement is shown graphically in Figure 2. with an array of N elements (0 to N-1) and then to variables that point to the in and out locations. Now when the two pointers are the same (A), that means there is no data in the buffer. Once data is available, it can be written to the location indicated by InPtr, InPtr is incremented and this will indicate that the buffer now contains data to be processed (B). The part of the code that handles the processing of data can see that the pointer are not equal, pull out the data out of the location indicated by OutPtr, and OutPtr is incremented (C). One final note, when InPtr or OutPtr are incremented and reach the value of N, they should be set back to zero.

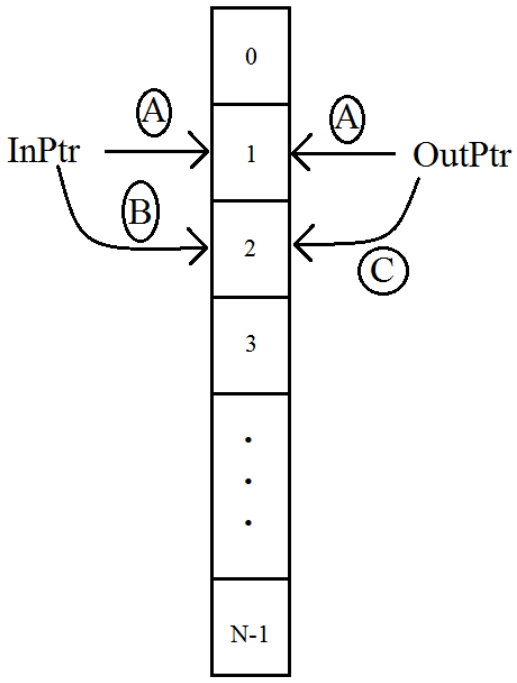

Figure 2. Rolling Buffer Graphic.

Finally a Finite State Machine (FSM) that will manage the shifting out of the bits from our encoding of the Morse code, and the turning on and off of a light or buzzer. The state machine is described by the STD in Figure 3. We have discussed the implementation of such machines many times.

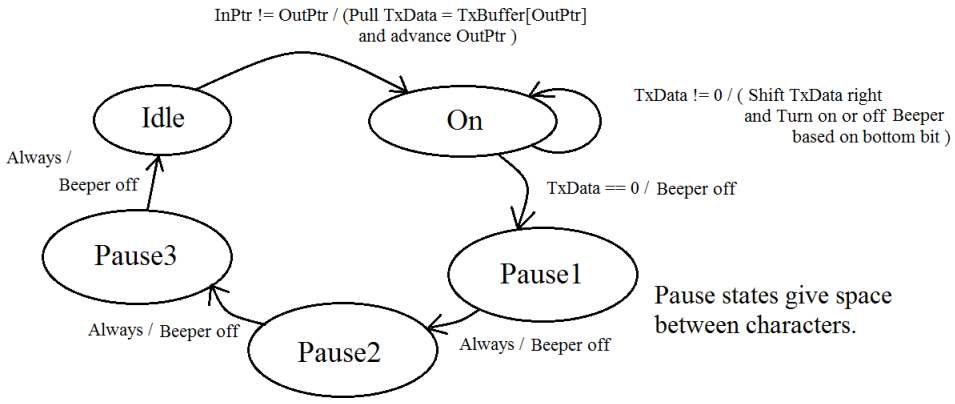

Figure 3. State Transition Diagram for the Sending of Dot and Dashes.

Appendix A: Code for character to Morse Code Translation and Transmitting

```
#ifndef AlphaToMorseCode_h
#define AlphaToMorseCode_h

// Set up look table to translate Alphas to
// Morse code dots and dashes
const long MorseCodeAlpha[26] = {
0x1D,  // A
0x157, // B
0x25,  // C
0x57,  // D
0x01,  // E
0x5D,  // F
0x177, // G
0x55,  // H
0x05,  // I
0x5D7, // J
0x1D7, // K
0x1F,  // L
0x77,  // M
0x17,  // N
0x11,  // O
0x155, // P
0x175, // Q
0x29,  // R
0x15,  // S
0x07,  // T
0x75,  // U
0x1D5, // V
0x1DD, // W
0x15D, // X
0xA5,  // Y
0x95 }; // Z

// Set up look table to translate numbers to
// Morse code dots and dashes
const long MorseCodeNumeric[10] = {
0x77777,  // 0
0x1DDDD,  // 1
0x07775,  // 2
0x1DD5,   // 3
0x0E55,   // 4
0x0155,   // 5
0x0557,   // 6
0x1577,   // 7
0x5777,   // 8
0x17777 }; // 9
```

```
// Function to translate via look-up tables,
// from ascii to encoded Morse code sequence
long Ascii2MorseCode(char Incoming)
{
	long ReturnValue = 0;

	// Convert lower case to upper case.
	if (Incoming >= 'a' && Incoming <= 'z')
		Incoming &= ~0x20;

	if (Incoming == ' ') // if we have a blank
		ReturnValue = 0; // Simply sent space time
	else if (Incoming >= 'A' && Incoming <= 'Z')
	{
		// If alpha, Load with encoded bit pattern.
		ReturnValue = MorseCodeAlpha[Incoming - 'A'];
	}
	else if (Incoming >= '0' && Incoming <= '9')
	{
		// If numeric, Load with encoded bit pattern.
		ReturnValue = MorseCodeNumeric[Incoming - '0'];
	}

	return ReturnValue;

}

#define MorseCodePin A0

// Define rolling buffer for incoming data.
#define BUFFERSIZE 256
char TxBuffer[BUFFERSIZE];
int TxInPtr = 0;
int TxOutPtr = 0;

// Support code for placing data in the buffer
int TxInsertInBuffer(char Input)
{
	TxBuffer[TxInPtr++] = Input; // Place input into buffer
	// and also advance pointer.
	if (TxInPtr == BUFFERSIZE)    // Check for wrap of pointer
		TxInPtr = 0;
	return (TxInPtr == TxOutPtr); // return true if buffer is full.
} // End of TxInsertInBuffer

// Pull data from buffer.
char TxPullFromBuffer()
{
	char Temp = TxBuffer[TxOutPtr++]; // Pull data from buffer, save for later
	// and also advance pointer.
	if (TxOutPtr == BUFFERSIZE)      // Check for wrap of pointer
		TxOutPtr = 0;
	return Temp;                      // Return data pulled from buffer.
} // End of TxPullFromBuffer
```

```
// Set up transmitter state machine
enum TxStates {
	TxIdle, // State waiting for character
	TxOn, TxOff, // toggle light on and off
	TxPause1, TxPause2, TxPause3
};// Pause between
// characters.
int TxState = TxIdle;
long TxData = 0; // Holds data being transmitted.
// State Transition Code
// Returns a flag of whether the light is on or off.
int TxNextState()
{
	int ReturnValue = 0;  // Default is light off.
	switch (TxState)
	{
	case TxIdle:  // idling waiting on data in buffer
		if (TxOutPtr != TxInPtr)
		{
			// Data in buffer, so
			Serial.print(TxBuffer[TxOutPtr]); // Debug code
			// Pull out data, and translate to MorseCodeBits.
			TxData = Ascii2MorseCode(TxPullFromBuffer());

			TxState = TxOn;              // Move to ON state.
			ReturnValue = TxData & 0x01; // bottom bit controls beeper.
		}
		break;
	case TxOn:
		TxData = TxData >> 1; // shift next bit to bottom
		ReturnValue = TxData & 0x01; // bottom bit controls beeper.
		if (TxData == 0)  // if all bits are done
		{
			TxState = TxPause1; // Move to pause states
		}
		break;
	case TxPause1: // Delay states waiting between characters.
		TxState = TxPause2;
		break;
	case TxPause2:
		TxState = TxPause3;
		break;
	default:
	case TxPause3:
		TxState = TxIdle;
		break;
	}
	return ReturnValue;

}// End of TxNextState


#endif
```

Appendix B: Button Support Code.

```
#ifndef ButtonDebounce_H
#define ButtonDebounce_H
// Set up pin and button state.
int ButtonPin = 4;
enum buttonStates { BS_Idle, BS_Wait, BS_Low };
buttonStates buttonState = BS_Idle;
unsigned long buttonTimer;

// Initialization code, setting up pin.
void ButtonInitialize(int pin)
{
	ButtonPin = pin;
	pinMode(ButtonPin, INPUT);
} // End of ButtonInitialize

// Function called in loop to check for button release.
// Returns a 1 on the buttons release.
int ButtonTest()
{
	// Read in the buttons current value.
	int Press = digitalRead(ButtonPin);
	int ReturnValue = 0;
	switch (buttonState)
	{
	case BS_Idle: // if we are waiting for a press,
		if (Press == LOW)
		{	// Once press occurs
			buttonTimer = millis();  // record time
			buttonState = BS_Wait;		// and move to next state
		}
		break;
	case BS_Wait: // button just went low
		if (Press == HIGH) // and now goes high
		{
			buttonState = BS_Idle;  // return to 0 state.
		}
		else // if still low
		{	// and sufficient time has passed.
			if (millis() - buttonTimer >= 10)
			{
				ReturnValue = 1; // Return 1 indicating pressed.
				buttonState = BS_Low; // move on to state two
			}
		}
		break;
	case BS_Low:
		if (Press == HIGH)
		{
			ReturnValue = 2; // Return 2 indicating release.
			buttonState = BS_Idle;
		} // End of high test.
		break;
	} // End of switch on buttonState

	return ReturnValue;

} // End of ButtonRead
#endif
```

Appendix C: Encoder Monitoring Code.

```
#ifndef EncoderMonitor_h
#define EncoderMonitor_h

// Variable for keeping track of encoder change.
volatile int EncoderValue;

// Service Routine for Encoder line A
void EncoderMonitorPinA(void)
{
	// compare pin 3 and pin 2
	if (bitRead(PIND, 3) == bitRead(PIND, 2))
	{
		EncoderValue--;
	}
	else
	{
		EncoderValue++;
	}
}// End of EncoderMonitorPinA

// Service Routine for Encoder line B
void EncoderMonitorPinB(void)
{
	// compare pin 3 and pin 2
	if (bitRead(PIND, 3) == bitRead(PIND, 2))
	{
		EncoderValue++; // Note this is the opposite
		// of the action in PinA.
	}
	else
	{
		EncoderValue--;
	}
} // End of EncoderMonitorPinB

// Initializes the Encoder Monitoring Code.
void EncoderInitialize()
{
	// Set encoder value to zero
	EncoderValue = 0;

	// Set up interrupts to monitor inputs from encoder.
	attachInterrupt(0, EncoderMonitorPinA, CHANGE);
	attachInterrupt(1, EncoderMonitorPinB, CHANGE);

}

#endif
```

Appendix D: Clock Basics, with Support for Button and Encoder setting of Clock.

```
#ifndef ClockBasics_H
#define ClockBasics_H

// Variable used as clock settings.
int Hours, Minutes, Seconds;

// This function is to be called every second
// to update the clock represented by the
// global variables Hours, Minutes, Seconds
void UpdateClock()
{
	// Check if Seconds not at wrap point.
	if (Seconds < 59)
		Seconds++; // Move seconds ahead.
	else
	{	Seconds = 0; // Reset Seconds
		// and check Minutes for wrap.
		if (Minutes < 59)
			Minutes++; // Move seconds ahead.
		else
		{	Minutes = 0; // Reset Minutes
			// check Hours for wrap
			if (Hours < 23)
				Hours++;// Move Hours ahead.
			else
			{
				Hours = 0;// Reset Hours
			}// End of Hours test.
		} // End of Minutes test
	} // End of Seconds test
} // end of UpdateClock()

// Send Hours, Minutes and Seconds to a display.
void SendClock()
{
	// Check if leading zero needs to be sent
	if (Hours < 10)
	{
		LcdDriver.print("0");
	}
	LcdDriver.print(Hours); // Then send hours
	LcdDriver.print(":"); // And separator
	// Check for leading zero on Minutes.
	if (Minutes < 10)
	{
		LcdDriver.print("0");
	}
	LcdDriver.print(Minutes); // Then send Minutes
	LcdDriver.print(":"); // And separator
	// Check for leading zero needed for Seconds.
	if (Seconds < 10)
	{
		LcdDriver.print("0");
	}
	LcdDriver.print(Seconds); // Then send Seconds
} // End of SendClock()
```

```
// States for setting clock.
enum ClockStates {
	CLOCK_RUNNING, CLOCK_SET_HOURS,
	CLOCK_SET_MINUTES, CLOCK_SET_SECONDS
};
ClockStates clockState = CLOCK_RUNNING;

// Function that processes incoming characters to set the clock.
void SettingClock(int Button, int EncUpDown)
{
	// interpret input based on state
	switch (clockState)
	{
	case CLOCK_RUNNING:
		if (Button)
		{
			clockState = CLOCK_SET_HOURS;
			Hours = 0;   // Reset clock values.
			Minutes = 0;
			Seconds = 0;
		}
		break;
	case CLOCK_SET_HOURS: //
		if (EncUpDown > 0)
		{
			Hours++;
			if (Hours > 23)
				Hours = 0;
		}
		else if (EncUpDown < 0)
		{
			Hours--;
			if (Hours < 0)
				Hours = 23;
		}
		else if (Button)
			clockState = CLOCK_SET_MINUTES;
		break;
	case CLOCK_SET_MINUTES: //
		if (EncUpDown > 0)
		{
			Minutes++;
			if (Minutes > 59)
				Minutes = 0;
		}
		else if (EncUpDown < 0)
		{
			Minutes--;
			if (Minutes < 0)
				Minutes = 59;
		}
		else if (Button)
			clockState = CLOCK_SET_SECONDS;
		break;
	case CLOCK_SET_SECONDS: //
		if (EncUpDown > 0)
		{
			Seconds++;
			if (Seconds > 59)
				Seconds = 0;
```

```
                }
                else if (EncUpDown < 0)
                {
                        Seconds--;
                        if (Seconds < 0)
                                Seconds = 59;
                }
                else if (Button)
                        clockState = CLOCK_RUNNING;
                break;

        }// End of clock mode switch.

} // End of SettingClock

#endif
```

Appendix E: Main Arduino Code for Morse Code Transmit

```
#include <LiquidCrystal.h>
// Defines for LCD
LiquidCrystal LcdDriver(11, 9, 5, 6, 7, 8);

#include "ASCIIToMorseCode.h"
#include "ClockBasics.h"
#include "ButtonDebounce.h"
#include "EncoderMonitor.h"
int EncoderTracking = 0;


// Timer to control Clock, TxMorseCode, Led and Encoder.
#define CLOCK_TIME 1000
int DOT_TIME = 50;
#define LED_FLASH_TIME 1000
#define ENCODER_INTERVAL 250
unsigned long ClkTimer, TxTimer, LedTimer, EncoderTimer;

// Tri Colored LED's
#define RedLedPin 13
#define GreenLedPin 12
#define BlueLedPin 10
unsigned long INTERVAL = 1000;
int LedState = BlueLedPin;

// setup code, ran once:
void setup()
{
	pinMode(RedLedPin, OUTPUT); // Set to control LED
	pinMode(GreenLedPin, OUTPUT); // Set to control LED
	pinMode(BlueLedPin, OUTPUT); // Set to control LED

	pinMode(MorseCodePin, OUTPUT); // Set to control beeper

	// Set up clock
	// first the inputs for setting the clock.
	ButtonInitialize(4); // connect to push button
	EncoderInitialize();   // Monitor encoder.
	EncoderTracking = EncoderValue;
	// Set initial time
	Hours = 9;
	Minutes = 58;
	Seconds = 58;

	// Lcd and Serial
	LcdDriver.begin(16, 2);
	Serial.begin(9600);    // Serial Port on

	TxTimer = millis();    // Timer for Morse Code transmitting.
	LedTimer = millis();   // Timer for led flash
}

int Lcd_Update = 0;
// main code, ran repeatedly:
void loop()
{
	// Clock Control
	if (millis() - ClkTimer >= CLOCK_TIME)
```

```
{
	if (clockState == CLOCK_RUNNING)
		UpdateClock();
	Lcd_Update = 1; // Set flag to update the display
	ClkTimer += CLOCK_TIME;
}

// If update of lcd is requested do it.
if (Lcd_Update)
{
	LcdDriver.clear();
	SendClock();  // Place time
	switch (clockState)
	{
	case CLOCK_SET_HOURS:
		LcdDriver.setCursor(0, 0);
		break;
	case CLOCK_SET_MINUTES:
		LcdDriver.setCursor(3, 0);
		break;
	case CLOCK_SET_SECONDS:
		LcdDriver.setCursor(6, 0);
		break;
	case CLOCK_RUNNING:
		LcdDriver.setCursor(0, 1);
	}
	LcdDriver.blink();
	Lcd_Update = 0; // Clear lcd update flag.
}

// Flash LED
if (millis() - LedTimer >= INTERVAL)
{
	// Update timer
	LedTimer += INTERVAL;

	// based on state toggle extra led
	switch (LedState)
	{
	case RedLedPin:
		PORTB = (PORTB & 0xDF) | 0x10; // Pin 13 off, Pin 12 on
		LedState = GreenLedPin;
		break;
	case GreenLedPin:
		PORTB = (PORTB & 0xEF) | 0x04; // Pin 12 off, Pin 10 on
		LedState = BlueLedPin;
		break;
	default:
	case BlueLedPin:
		PORTB = (PORTB & 0xFB) | 0x20; // Pin 10 off, Pin 13 on
		LedState = RedLedPin;
		break;
	} // end of led toggle.

	// Change interval after each cycle.
	if (LedState == RedLedPin)
	{
		INTERVAL -= 50;
		if (INTERVAL < 100)
			INTERVAL = 500;
```

```
            } // end of update of interval change
        } // end of led timer.

        // Timer to control time of beeper on and off.
        if (millis() - TxTimer >= DOT_TIME)
        {
            // Based on state,
            if (TxNextState())
            {
                digitalWrite(MorseCodePin, HIGH); // Beeper on.
            }
            else
            {
                digitalWrite(MorseCodePin, LOW); // beeper off.
            }
            TxTimer += DOT_TIME; // Update timer
        }

        // Watch button and encoder to set clock
        if (ButtonTest() == 1)  // Button pressed,
            SettingClock(1, 0);  // So update state.
        else if ((EncoderValue - EncoderTracking) >= 4)
        {   // The encoder has moved by one detent
            SettingClock(0, -1);  // Indicate positive travel
            EncoderTracking += 4;   // and move tracking variable.
        }
        else if ((EncoderValue - EncoderTracking) <= -4)
        {   // The encoder has moved by one detent in the opposite direction
            SettingClock(0, +1);  // Indicate negative travel
            EncoderTracking -= 4;   // and Update Tracking accordingly
        }

        // Check for incoming serial data and place it in the buffer.
        if (Serial.available())
        {
            TxInsertInBuffer(Serial.read()); // Read data into buffer
        } // End of Serial input

}// End of loop.
```