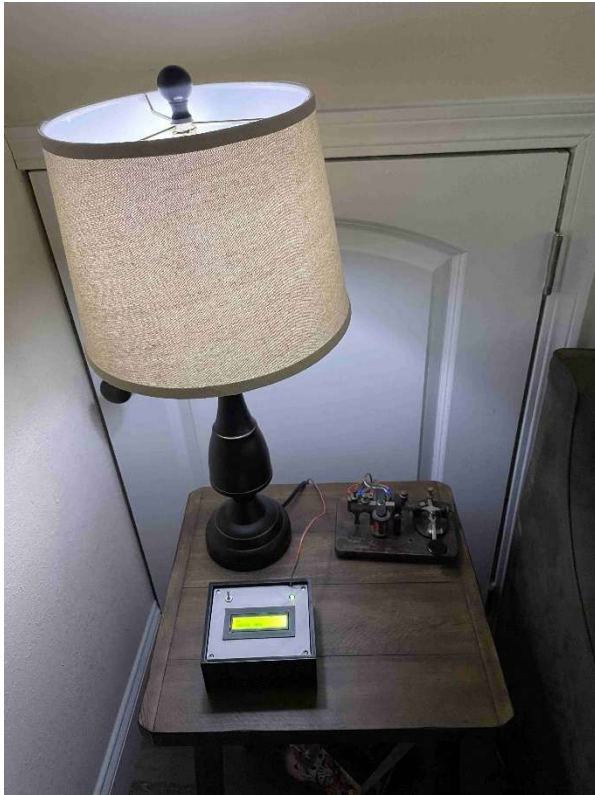


## Morse Code Decoder

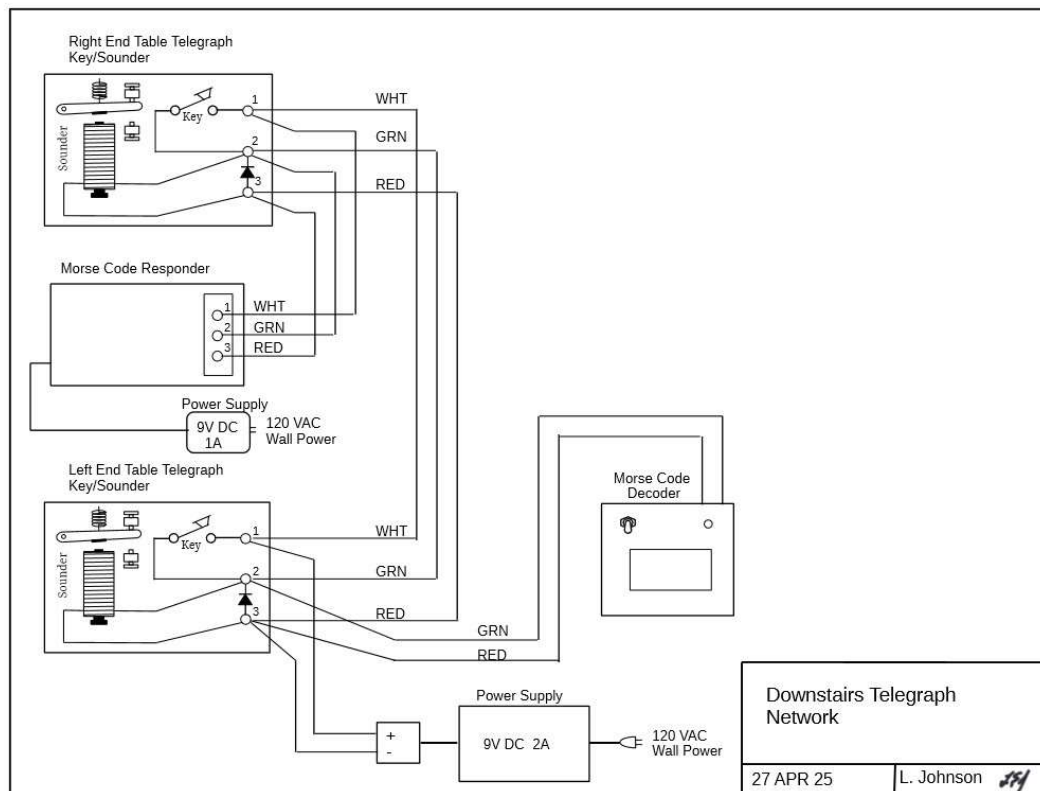
By  
Lloyd Johnson

This document is about an Arduino based project that incorporates antique telegraph key/sounders. The Morse Code Decoder decodes the activity of the sounder and displays the message on a 16 character, 2 line LCD display. This is done by monitoring the voltage across the sounder's coil and measuring the time duration each time the coil is energized. These time durations are converted into the dits and dahs that make up a Morse Code character. The dits and dahs are decoded into alphanumeric characters and displayed on the LCD display.

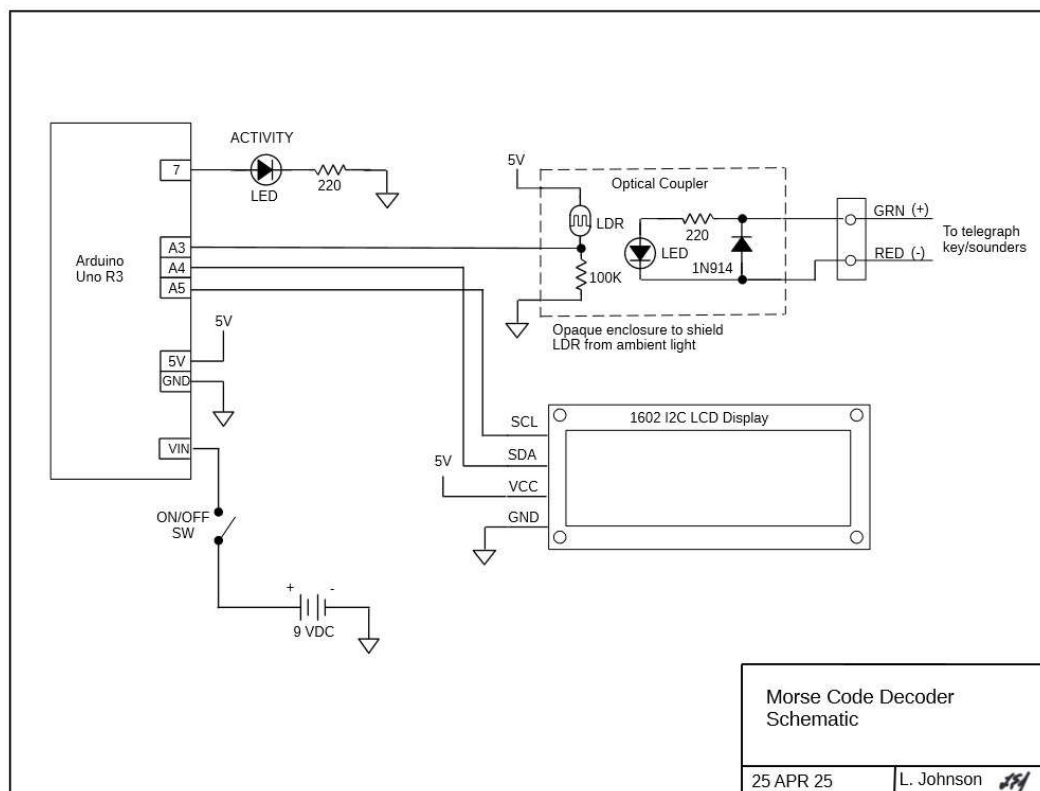
The Morse Code Decoder is shown being deployed in my downstairs telegraph network on the left end table as shown by the next two photographs.



My downstairs telegraph network consists of hardware located on two end tables on both sides of a sofa. I updated the drawing for this network to include the Morse Code Decoder. A diagram of the updated downstairs telegraph network follows. The Morse Code Responder shown in this diagram was my previous Arduino project. Although it was useful for testing the Morse Code Decoder, the Morse Code Responder is not essential for operation of the Morse Code Decoder. If interested, the Morse Code Responder documentation can be found at <https://github.com/LEJ-Projects/Morse-Code-Responder>.



The next image is a schematic diagram of the Morse Code Decoder.



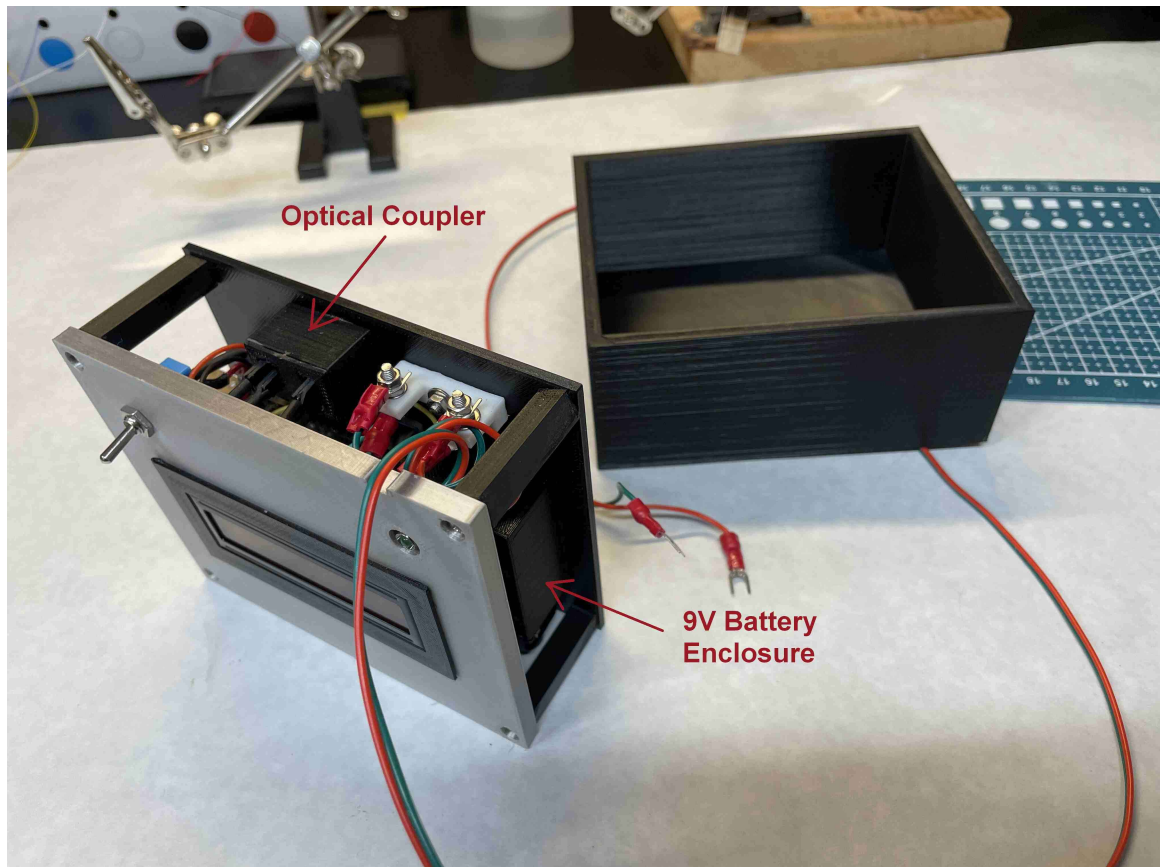
The main components are the optical coupler, the 16 character, 2-line LCD display and the Arduino Uno R3 microcontroller. I electrically isolated the telegraph key/sounder from the Arduino circuitry by using the optical coupler circuitry. This is shown in the Morse Code Decoder schematic in the region enclosed by a dotted line box.

When the current suddenly stops through the coil of a telegraph sounder (when a telegraph key is released), a large negative voltage spike appears across the sounder coil. This spike will destroy modern electronics unless some precautions are taken. The 1N914 diode clamps the negative spike. In addition, total electrical isolation is achieved by using an LED and a light dependent resistor (LDR) to couple the signal from the telegraph sounder to the Arduino. The optical coupler circuitry has responded well with input voltages ranging from 5 VDC to 9 VDC. The following is a picture of the optical coupler circuit before it is inserted into an opaque enclosure. The holes on the perforated circuit board are on 0.1-inch centers.



The optical coupler is enclosed in an opaque enclosure to prevent ambient light from interfering with its operation. This can be seen in the annotated photograph that follows:



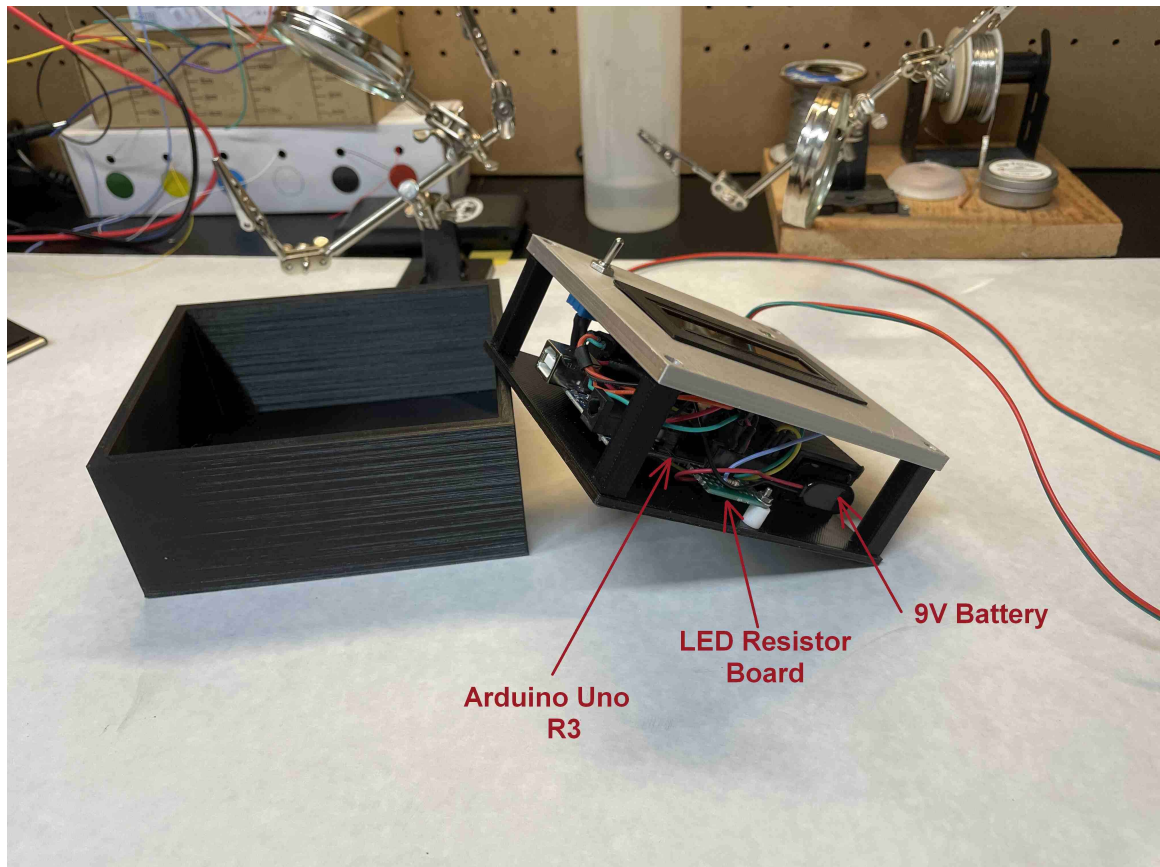


This photograph also illustrates how the Morse Code Decoder is assembled. The Morse Code Decoder sits in a 3D printed box. It can be removed from this box simply by turning it upside down and letting it drop out.

The optical coupler and the 9V battery enclosure were glued to the bottom panel. The white terminal strip was 3D printed for this project. It is attached to the bottom panel with a machine screw. The front panel can be separated from the Morse Code Decoder by removing the 4 machine screws in the corners of the front panel. I have not separated the front panel since the original installation because all electrical connections to and from the Arduino, the optical coupler, and the 1602-I2C display are made with pin and socket connectors that will disconnect easily if care is not taken.

The 1602-I2C LED display is attached to the back of custom 3D printed bezel with nylon screws. The bezel has a 2mm lip around the edge of the top surface enabling it to be dropped into a rectangular opening on the front panel and secured with glue. It is important to have the contrast on the 1602-I2C display properly adjusted before attaching the panel to the Morse Code Decoder since the contrast adjustment is no longer accessible when the panel is attached.

The next picture provides another angle of the Morse Code Decoder when removed from the outer enclosure.



The above picture shows how the Arduino, LED resistor board and 9V battery are installed on the bottom panel. The 9V battery slides out of the enclosure easily when it is necessary to replace it. So far it has only been necessary to replace the battery when the power switch was accidentally left in the “on” position overnight. In the two times this happened, the battery was completely drained. The LED on the front panel illuminates whenever the signal from the optical coupler indicates a sounder coil has been energized. The current limiting resistor for the LED is mounted on the LED resistor board near the bottom of the bottom panel.

The software running the Morse Code Decoder was written in C and was developed on the Arduino IDE (Integrated Development Environment). Arduino calls this program a sketch and uses the extension, “.ino” for the source file. The .ino file is a plain text file that can be viewed with Notepad or any other plain text editor. An Arduino sketch has two major sections: setup and loop. Prior to the setup section, there is code to identify include files, and to define variables and constants. The include files, Wire.h, and LiquidCrystal\_I2C\_Hangul.h are necessary to communicate with the 1602-I2C LCD display. A call to the function, LiquidCrystal\_I2C\_Hangul lcd with parameters 0x27, 16, and 2, identify the display has an address of 0x27 and is 16 characters by 2 lines. The address for the 1602-I2C LCD display may vary. There are data sheets available that identify which address to use. (The newer address is 0x3F.)

Besides defining the constants, TRUE and FALSE (which are defined as 1 and 0 respectively), I also defined several constants that affect how the Morse Code Decoder decodes and displays decoded Morse Code. These are shown in the following table:

| Constant | Value (in mSecs) | Description  |
|----------|------------------|--|
| DORMNT   | 6000             | Dormant Threshold – This constant represents the amount of time that must pass before a flag is set such that the next character will be displayed on a new line. The current line will be scrolled (redisplayed) on line 1 and the new character will be displayed on line 2. |
| ENDCHR   | 500              | End Character – This is the amount of idle time that must pass before a Morse Code character has been declared received. It represents the amount of time between the characters of a word and is sometimes referred to as the Inter-character space.                          |
| NOISTM   | 25               | Noise time – Any pulses shorter than 25 mSecs are discarded. This is done to avoid erroneously decoding key bounces as dits.   |
| MINDAH   | 250              | Minimum dah duration. Any pulses shorter than MINDAH are considered dits. Any pulses equal to or longer than MINDAH are considered dahs.   |
| SPCTHR   | 1000             | Space Threshold – Once the time between the prior key tap and the current key tap exceeds SPCTHR, a space will be inserted before displaying the next character (provided that this time does not also exceed DORMNT).   |

These values should work for speeds from 4.8 wpm through 14.4 wpm. A MINDAH value of 250 mSec corresponds to 4.8 wpm based on the formula  $wpm = 1200/T$  where T is the dit time in mSec. If the duration of a dah is less than 250 mSec, the dah will be erroneously decoded as a dit. Since a dah is defined as 3 dit times, the dit time becomes 83 mSec which corresponds to 14.4 wpm. Going faster than 14.4 wpm will result in dahs being interpreted as dits. This is not a problem for me. I can't go anywhere near that fast.

However, most active cw operators' speed exceeds 15 wpm. For them, the value of MINDAH should be lowered. MINDAH should be set to 86 mSec, to allow speeds from 14 wpm to 41 wpm to work correctly. In addition, ENDCHR and SPCTHR should also be reduced to a third of their current values. To test these changes, I temporarily updated the Morse Code Responder software to transmit messages at 20 wpm and verified the Morse Code Decoder can decode them properly when MINDAH, ENDCHR and SPCTHR were modified accordingly.

By defining variables at the beginning of the program before the setup and loop section, the variables are now global. With global variables, there is no need to return them as values or pass them as arguments when exiting or calling functions. This greatly simplified the programming. I've attempted to include a meaningful comment for each variable after its definition.

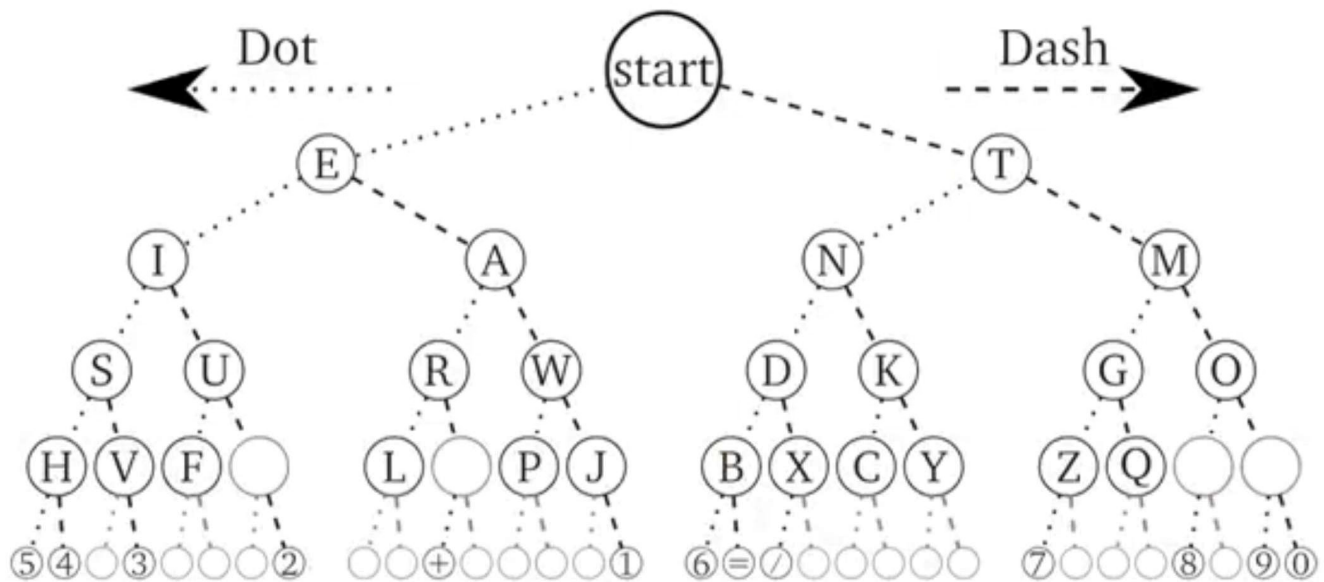
Code that is in the setup section is only executed one time when the Arduino is powered on or reset. The setup section contains function calls for establishing serial communication to a terminal and the 1602-I2C LCD display. This is followed by several more function calls for configuring the LCD display and the LED output pin. There are two variables that are initialized in this section.

The loop section (as its name implies) is the section of code that when execution is complete, it loops back to the beginning of the section and runs the section again. The loop section has three tasks to perform each loop. These are:

1. Get a Morse Code character.
2. Decode the Morse Code character
3. Display the Morse Code character

Getting a Morse Code character involves making a function call to the function, `getddchar`. Program execution will remain in this function until a Morse Code character is obtained. When the function does exit, the array, `mc` will contain a Morse Code character consisting of 0s and 1s corresponding to dits and dahs respectively. Within this function, flags are set for identifying if a leading space is needed, or if scrolling should occur due to inactivity. The variables for these two flags are `spfl` (space flag) and `scrlnw` (scroll now). These flags will be used later when displaying the Morse Code character. The value of the debounced symbol count variable, `dbsymnt` determines how many elements of the array to decode.

The next task is to decode the Morse Code array, `mc` and store the decoded character in the first element of the string array, `mcd`. The 2<sup>nd</sup> element of this array is preloaded with the string terminator, 0. The following diagram represents the approach this program uses for decoding dits and dahs (dots and dashes).



The first level of comparison is performed within the loop section. The comparison consists of checking if the first Morse Code symbol is dit or a dah by checking if `mc[0]` is 0 or 1. If `mc[0]` is 0 and the `dbsymnt` is 1, then the decoded Morse Code character is the letter, “E” and decoding is

complete. If `mc[0]` is 0 and the `dbsymcnt` is greater than 1 then the function `dd0()` is called which will determine if the Morse Code character array decodes to the letter, “I” or “A” or if additional functions must be called due the value of the debounced symbol count. The same logic is used if the first symbol is a dah which will result in either the letter “T” or additional function calls. The decoded Morse Code character is set to “ ” (a space), if there is no definition for the received dit/dah pattern.

The third task in the loop section is to display the Morse Code character. There are several steps that are first taken before writing the character to the display. The first step is to perform a “scroll now” check. The scroll now flag, `scrlnw` is checked and if true, a scrolling event occurs. This results in clearing the LCD display; printing the display line 2 string, `dl2` on the first line of the LCD; and clearing the space needed flag, `spfl`.

The next step of displaying the Morse Code character is to determine if a leading space is required and then display the space before displaying the Morse Code character. If the leading space needed flag, `spfl` is true and the number of characters already displayed is not 16, then the space is displayed and copied to current position of display line 2 string array. In addition, the location of the last space variable, `lstsp` is set equal to the character count variable, `ccnt`; and `ccnt` is incremented. However, if the leading space needed flag, `spfl` is true and the number of characters already displayed is 16, a scrolling event is needed, and the space will not be added. Instead, the display is cleared, the contents of the display line 2 string array, `dl2` is sent to the LCD display where it will appear on line 1. The LCD cursor position will be set to the beginning of the second line and the last space variable, `lstsp` will be set to -1 indicating there are no spaces in the second line.

The final step of displaying the Morse Code character starts with yet another scroll test. This is performed by calling the function, `scrolltest`. This function determines if a scroll event is necessary and if so, it will perform all the necessary actions to perform the scrolling as well as updating any necessary variables. When execution returns from the function, the Morse Code character is sent to the LCD display. The Morse Code character is also stored in the display line 2 string array, `dl2` at the array position pointed to by the character count variable, `ccnt`. The string array is indexed at 0. The character count variable, `ccnt` incremented and the string array is terminated by setting the position pointed to by the character count variable, `ccnt` to 0.

This completes execution of the section, loop. Program execution loops back to the beginning of the loop section and executes this section of code again. Execution of the loop section will continue until power is removed. Following the loop section are 26 function definitions.

The first four functions are called from the loop section. These functions are `scrolltest`, `getddchar`, `dd0`, and `dd1`.

The first function is `scrolltest`. The `scrolltest` function determines if a scroll event is necessary and if so, it will perform all the necessary actions to perform the scrolling as well as to update any necessary variables. A scroll event is required if the character count variable, `ccnt` is equal to 16. If not equal to 16, the function exits and no scrolling occurs. Scrolling begins by copying the display line 2 string array, `dl2` to the display line 1 string array, `dl1`. If the last space variable, `lstsp` is greater than 0, then smart scrolling occurs. Smart scrolling consists of terminating the display line 1 string array, `dl1` at the position pointed to by the last space variable, `lstsp`. The display line 1 string array, `dl1` is then displayed on line 1 of the LCD display. The characters stored in `dl1` after the last space are copied to display line 2 string array, `dl2` and displayed on the second line of the LCD display. The character count variable, `ccnt` is set equal to the length of `dl2` and the last space



variable, lstsp is set to -1 indicating there are no spaces in the new line displayed on the second line of the LCD display. To illustrate this, if the letters, “MORSE CODE DECOD” are received and displayed on the second line, with random words on the first line, before the last two letters are received, the LCD display might look like this:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | A | N | D | O | M |   | W | O | R | D | S |   |   |   |   |
| M | O | R | S | E |   | C | O | D | E |   | D | E | C | O | D |

When the next two letters (E and R) are received, the letter E results in a smart scrolling event. The random words on the first line are scrolled off. The characters before the last space on the second line are displayed on the LCD first line and characters after the last space are moved to the beginning of the second line. The new letters (E and R) are added to the end of the second line resulting in a display that looks like:

|   |   |   |   |   |   |   |   |   |   |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|--|--|--|--|--|--|
| M | O | R | S | E |   | C | O | D | E |  |  |  |  |  |  |
| D | E | C | O | D | E | R |   |   |   |  |  |  |  |  |  |

Instead of this:

|   |   |   |   |   |  |   |   |   |   |  |   |   |   |   |   |
|---|---|---|---|---|--|---|---|---|---|--|---|---|---|---|---|
| M | O | R | S | E |  | C | O | D | E |  | D | E | C | O | D |
| E | R |   |   |   |  |   |   |   |   |  |   |   |   |   |   |

The function to get dit-dah characters is getddchar. This function consists of three sections. They are:

1. Initialize variables
2. Gather duration of key taps
3. Debounce key tap duration and convert to Morse Code

Each time the function, getddchar is called, variables must be initialized. These variables consist of counters and flags. All the counter variables are set to 0 and all flags are set to false.

Next the function gathers the duration values for whenever a key is pressed. This is done by executing a while loop where execution will continue looping until the done flag is set to true. The while loop begins by the voltage from the optical coupler and evaluating it. A voltage above 2.5 volts indicates the key is pressed. There is a different set of actions taken if the key is pressed or not pressed. This is followed by some common actions (cleanup) that are taken after completing the key pressed (or not pressed) actions.

When it is determined that the key is pressed, the LED is turned on and the previous state variable, prev\_st is evaluated. If the previous state variable, prev\_st is 0 indicating that previously the key was not pressed, previous state variable, prev\_st is set to 1 and the duration for the current symbol count is set to 0, (dur[symcnt]=0). If the previous state variable, prev\_st was 1, the duration for the current symbol count is incremented.

When it is determined that the key is not pressed, the LED is turned off and the previous state variable, prev\_st is once again evaluated. If the previous state variable, prev\_st is 1, the previous state variable, prev\_st and the zero duration variable, zdur are both set to 0. The symbol count variable, symcnt is incremented. The transition of the previous state variable, prev\_st from 1 to 0 is indicative that a Morse Code symbol (dit or dah) duration has been measured and stored in the symbol duration array, dur. Note, the index identifying the duration of the last symbol is always 1

less than the symbol count. This is because the C language indexes arrays with the index 0 as the first element.

Additional evaluations are performed when the key is not pressed. These are independent of the value of the previous state variable, `prev_st`.

The first evaluation determines if the millisecond timer variable, `mtime` is to be incremented. The variable `mtime` is incremented if both the scroll now flag, `scrlnw` is false and the symbol count variable, `symcnt` is 0. Once the scroll now flag, `scrlnw` is true, there is no need to increment the millisecond timer variable, `mtime`. Allowing this variable to increment unchecked could result in it overflowing with unpredictable results. However, since it increments 1000 times per second, it would require about one month and 19 days to exceed 4,294,866,296 (maximum integer) and the battery would be dead well before then. Still, I do not let it increment unchecked.

The next evaluation performed when the key is not pressed determines if the leading space needed flag, `spfl` should be set true. This flag is set to true when the millisecond timer variable, `mtime` exceeds the space threshold constant, `SPCTHR` and the character count variable, `ccnt` is greater than 0. Including the character count variable, `ccnt` in this evaluation prevents erroneous leading spaces being displayed after powering on or a scrolling event.

The third evaluation performed when the key is not pressed determines if the scroll now flag, `scrlnw` should be set to true. This flag is set to true when the millisecond timer variable, `mtime` exceeds the dormant threshold constant, `DORMNT` and the scroll now check flag variable, `snckf` is true. The scroll now check flag, `snckf` is initialized as false when it is defined prior to the setup section. It is set to true at the end of this function just prior to the function exiting. By keeping this flag false until the first character is obtained, the new characters are displayed on the first line of the LCD instead of being displayed on the second line due to a premature scrolling event.

The fourth evaluation performed when the key is not pressed determines if the zero duration variable, `zdur` should be incremented. This counter increments when the symbol count variable, `symcnt` is greater than 1. The symbol count variable is incremented only when the `prev_st` transitions from 1 to 0 indicating that the key has been pressed and released.

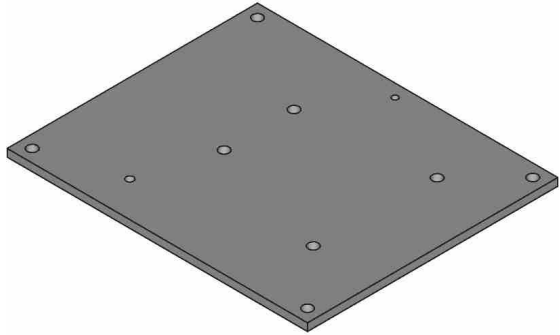
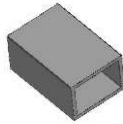
When the evaluations have been completed, there is some cleanup that is performed at the end of the while loop. The cleanup details consist of performing a 1 millisecond delay followed by an evaluation of the zero duration variable, `zdur`. The done flag is set to true if the value of `zdur` exceeds the value of the end character threshold constant, `ENDCHR`. When the done flag is true, program execution will drop out of the while structure (instead of looping back) and execution will continue with the next section of the function as described in the following paragraph.

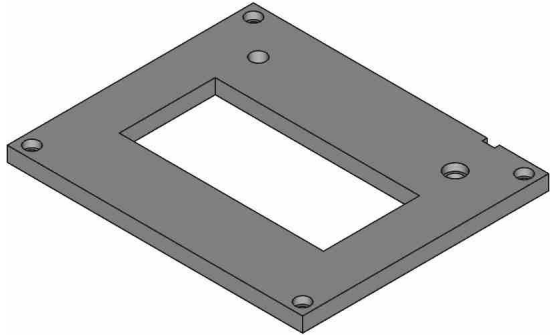
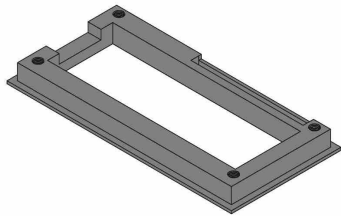
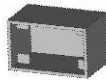
The third and last section of the `getddchar` function performs the task of debouncing key taps and converting the data in the duration array to Morse Code. A for loop is used to sequence through each element in the duration array, `dur`. Durations shorter than `NOISTIM` are considered switch bounces and are discarded. Only duration values greater than the value of the `NOISTIM` are used and stored in the debounced duration array, `dbdur`. Each time a debounced duration value is stored, it is also evaluated with the evaluation results stored in the Morse Code array, `mc`. A debounced duration with a value greater than the minimum dah constant, `MINDAH` is assigned the value 1 (for dah) and debounced duration with a value less than the minimum dah constant, `MINDAH` is assigned the value 0 (for dit). The debounced symbol count variable, `dbsymcnt` is incremented and the for loop will evaluate another symbol until the number of symbols evaluated is equal to the value of the symbol count variable, `symcnt`. Once all evaluations are completed, the enable new scroll check flag is set to true and the `getddchar` function exits by returning to code following the function call.

The next 24 functions are used for decoding the dit and dah information in the Morse Code array, mc into the single character string variable, Morse Code decoded, mcd. The string is defined as having 2 elements. The first element is the Morse Code character, and the second element is the string terminator. A string variable is used instead of character variable because the lcd.print function is expecting a string. It should be noted that each of the 24 functions have a name starting with the letters, dd and followed by 1 to 4 digits where the digits are either 1 or 0. The number of the digits indicates how many symbols of the Morse Code character have already been evaluated. The value of digits indicates the results of the evaluation. When the function dd0 is called, we know that the Morse Code character has 2 or more symbols and that the first symbol evaluated to 0 (a dit). If there are only 2 symbols, meaning the debounced symbol count variable, dbsymcnt is 2, then the Morse Code character is either the letter I, or the letter A depending on whether the second symbol is a 0 (dit) or a 1 (dah), respectively. If there are more than 2 symbols in the Morse Code character, then the function dd00 or dd01 is called depending on whether the second symbol is a 0 (dit) or a 1 (dah). The functions with 4 digits following the letters, dd will set the Morse Code character in the string array, mcd to either a number, punctuation or a space. The 4 digit functions will not call other functions. That means that at most, only the first 5 symbols are evaluated.


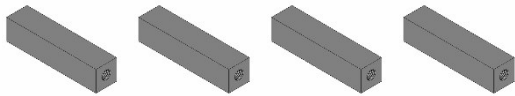
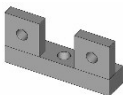
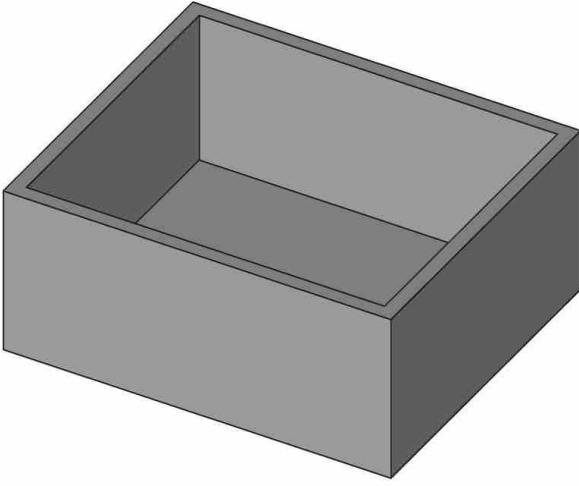
This concludes the description of the Morse Code Decoder Software.

The 3D printed parts used for assembly of the Morse Code Decoder were designed with FreeCAD 0.21.2 and printed on a Longer LK5 Pro 3D printer. The following table provides a description of each of the .stl files that were used for this project.

| Filename           | Description   | Image   |
|--------------------|---|---|
| MCD_Base.stl       | Morse Code Decoder – Base. This is the baseplate where the Arduino Uno, the optical coupler, the battery box, the terminal strip are mounted. The four screw holes in the center are spaced for the Arduino. The holes in the corner are for attaching the supports. The two holes on the edge are for the LED resistor board and the terminal strip. These holes may need to be resized to accommodate the screws that are used. |   |
| MCD_BatteryBox.stl | Morse Code Decoder – Battery Box. This box houses the 9V battery used for powering the device. The box is glued to the base plate next to the Arduino.  |  |
| MCD_FrontPanel.stl | Morse Code Decoder – Front Panel. The front   |   |

|                    |   |   |
|--------------------|---|---|
|                    | <p>panel has four counter sunk holes in the corners to accommodate the M3 machine screws used to fasten it to the supports. There are two other circular hole for the on/off switch and the LED. The hole for the LED is countersunk to accommodate the LED sockets. The center rectangular opening is for the 1602 LCD display bezel. The small rectangular opening on the top edge is for the wires going to the telegraph sounder.</p> |     |
| MCD_LCDBezel.stl   | <p>Morse Code Decoder – LCD Bezel. The 1602 LCD display typically presents mounting challenges. This bezel answers this challenge with cutaways to accommodate the circuit board pins and other protrusions on the display. The display is fastened to the bezel using the four threaded holes and M3 machine screws. The bezel is dropped into the front panel opening and secured with glue along the lip.</p>                          |   |
| MCD_OptiCupBod.stl | <p>Morse Code Decoder – Optical Coupler Body. The optical coupler circuit board is inserted into this box with the circuit board pins protruding from the box. This box is glued to the base plate to the side of the Arduino with the reset button. It should be positioned such that the on/off switch mounted on the front panel does not interfere with its position.</p>   |  |



|                    |   |  |
|--------------------|---|--|
| MCD_OptiCupLid.stl | <p>Morse Code Decoder – Optical Coupler Lid. This is the cover for the Morse Code optical coupler after the circuit board is installed. The posts on the corner hold the board securely against the protrusions in the body. The rectangular openings accommodate the pins protruding from the circuit board.</p>                                 |   |
| MCD_Support.stl    | <p>Morse Code Decoder – Support. These are the four supports that are installed between the front panel and the base. These supports are threaded on each end for a M3 machine screw.</p>   |    |
| MCD_Terminals.stl  | <p>Morse Code Decoder – Terminals. This item is fastened to the base with a machine screw and nut and serves as a strain relief between the optical coupler and the telegraph sounder. Machine screws and nuts are used to fasten wires from the optical coupler to one side of the terminal and wires to the telegraph sounder to the other.</p> |  |
| MCD_Shell.stl      | <p>Morse Code Decoder – Shell. This box serves as an enclosure for the Morse Code Decoder. The completed assembly is simply dropped inside.</p>   |  |