6.3 Processing and Control - Microcontroller

6.3.1 Overview

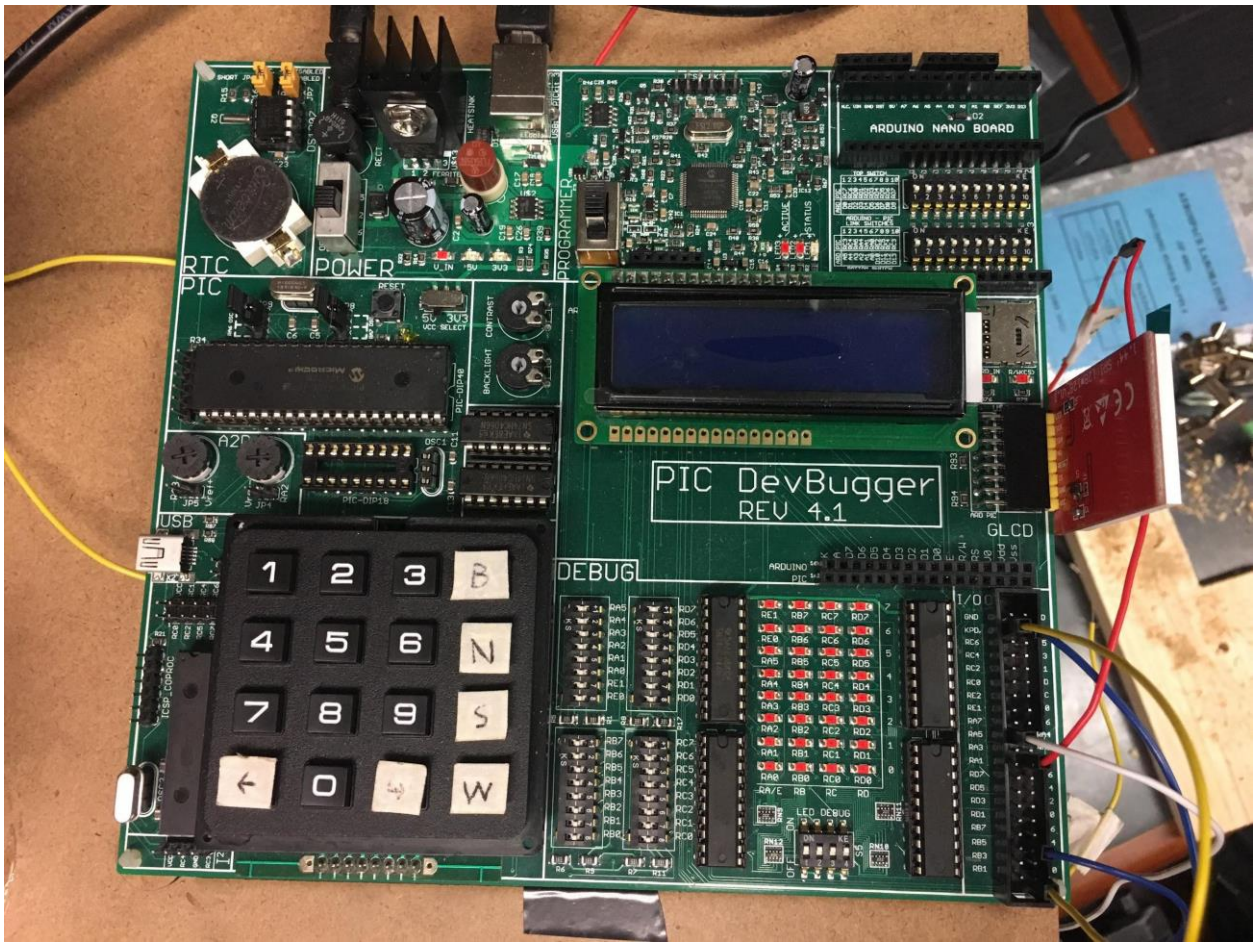


Figure 6.3.1.1: PIC18F4620 Microcontroller.

The microcontroller member is tasked with developing the sequential and combinational logic required for the system including the logic for the algorithms which control the sequence and timing of actuators based on voltage inputs from sensors and other features, see sections 6.3.2 for more details on the other features. To do so, they are provided with a microcontroller which is a computer on a chip that lacks the many of the standard peripherals that many computers have. These peripherals, which refer to external devices which provide inputs and outputs to a computer, that the microcontroller lacks include the keyboard, monitor, and power supply.

The specific microcontroller used for this system is the PIC18F4620 from Microchip. However, the microcontroller does include some peripherals. The first of these available peripherals are A/D converters, which are devices that convert input analog signals into a digital format. These signals are time-varying values that are used communicate information between devices and are often received in terms of voltage. Analog signals can adopt an infinite amount of values between a range of two values. Digital signals on the other are limited to specific quantities. To receive and transmit information between the microcontroller and other devices, the

microcontroller uses pins which are metal protrusions or insertions that can be attached to wires to transmit or receive electric signals.

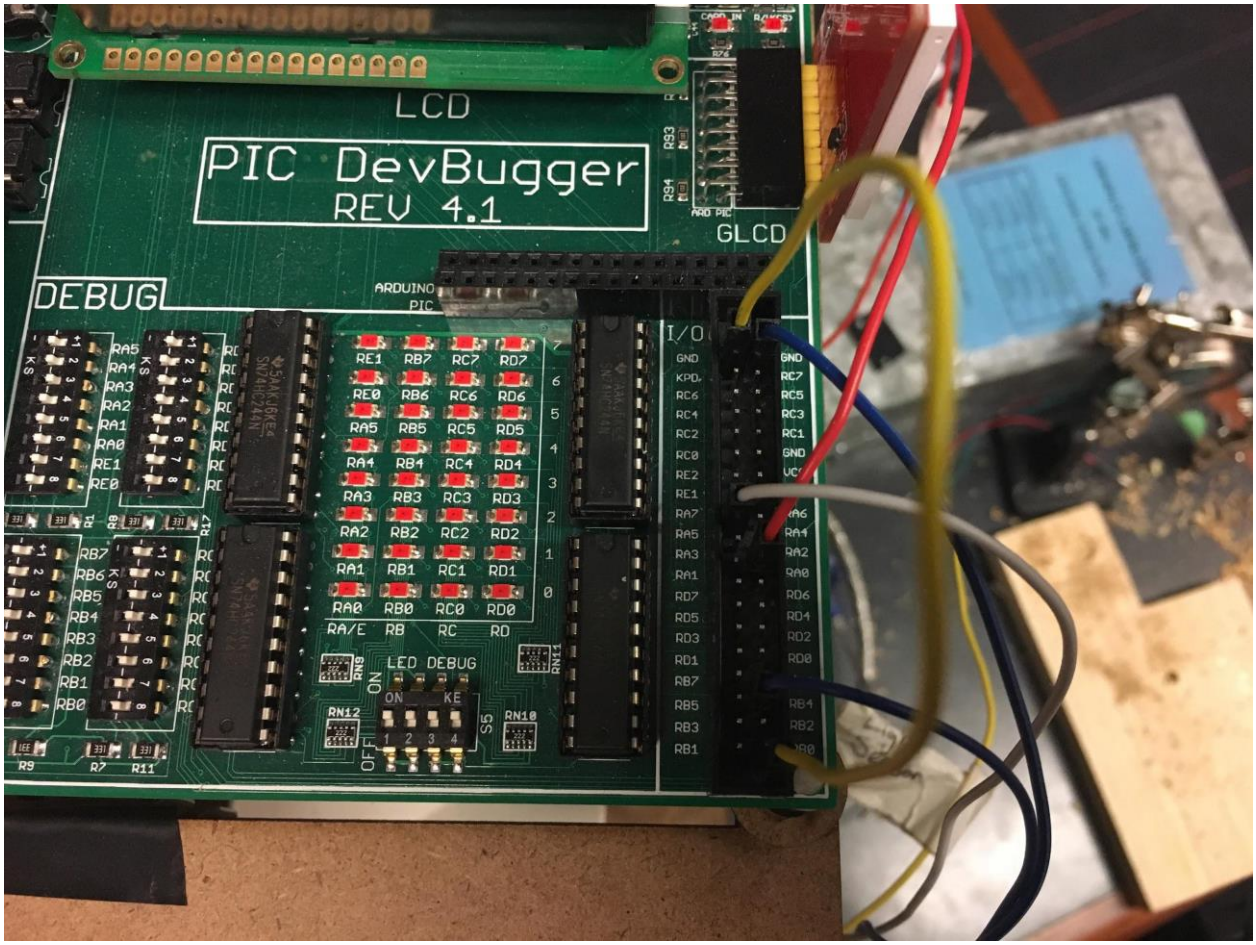


Figure 6.3.1.2: Pins and Debugging LEDs for PIC18F4620

The PIC18F4620 also includes timers which are based on a built-in crystal oscillator and count up to certain values and then activates a flag when the count is complete. There are multiple timer peripherals and they can be used to enable other features on the microcontroller such as Pulse Width Modulation (PWM), which can send high frequency output signals, the capture module which can read high frequency input signals, and the compare module which can compare a high frequency input signal to timers.

The microcontroller also features Interrupt Service Routines (ISR) which can perform activate certain commands when other actions are being performed by the microcontroller so long as certain conditions occur. Finally, the microcontroller includes EPROM, which refers to memory that does not reset upon power reset.

The microcontroller receives instructions through the C programming language and the code required will be communicated through the MPLAB X Integrated Development Environment (IDE) using the XC8 compiler. The IDE is software that facilitates software development and often includes a source code editor, build automation tools, and a debugger. The compiler converts the source code from the IDE into a format the microcontroller can interpret and execute upon. [3]

6.3.2 Assessment of the Problem

The microcontroller is required to develop the combinational and sequential logic required for operation. This includes the logic for the algorithms which control the sequence and timing of actuators based on voltage inputs from sensors, interfacing between the keypad and user interface displayed through the liquid-crystal display (LCD) and the GLCD. The microcontroller also develops and implements Permanent Logs that are stored in permanent EEPROM, Real-Time Date/Time Display which is a displayed clock that updates in real-time, and PC Interfacing.

The keypad is used to enable an operator to traverse through the UI and input instruction parameters. The instruction parameters refer to information input by the operator prior to operation to determine how many fasteners will be dispensed to each compartment. These instruction parameters including number of assembly steps, fastener sets, and number of fastener sets per step. The number of assembly steps refers to the number of compartments out of eight total compartments that will be filled with fasteners. This instruction parameter only accepts the following values: 4, 5, 6, 7, 8. A 4 step assembly requires every other compartment starting from C1 to be filled. The position of C1 is indicated by white electrical tape on the side of the compartment box which will be located adjacent and counterclockwise to C1. A 5-step assembly requires compartments 1, 2, 4, 5, and 7 to be filled. A 6-step assembly requires compartments 1, 2, 3, 5, 6, and 7 to be filled in. A 7-step assembly requires compartments 1, 2, 3, 4, 5, 6, and 7 to be filled. Finally, a 8-step assembly requires all available compartments to be filled [1].

Figure 6.3.2.1: Standard Keypad for Microcontrollers.

The fastener set refer to the combination of fasteners that will be dispensed into each compartment. This instruction parameter accepts the following combinations of bolts (B), nuts (N), spacers (S), and washers (W): B, N, S, W, BN, BS, BW, BBN, BBS, BBW, BNW, BSW, BWW, BNWW, BSWW, BBSW, BBNW, BNNW, BNNN, BWWW. These combinations are only accepted when input in the exact order as specified.

The fastener sets per step refers to the number of fasteners that will be dispensed into each compartment. This instruction parameter only accepts the following values: 1, 2, 3, 4. Both the fastener set and fastener sets per step parameters are requested multiple times for each compartment to be filled because this enables the operator to request unique fastener combinations to be dispensed into different compartments.

The UI will be primarily displayed through the LCD with supplementary information provided by the GLCD. The UI must intuitively communicate to the operator how to navigate the menus such that they can perform operation and access additional information such as the RTC and EEPROM logs as desired. The LCD can only display information through 2 lines of text with each line able to contain 16 characters including spaces, see figure 6.3.2.2 for the first interactable menu encountered by the operator in the UI.

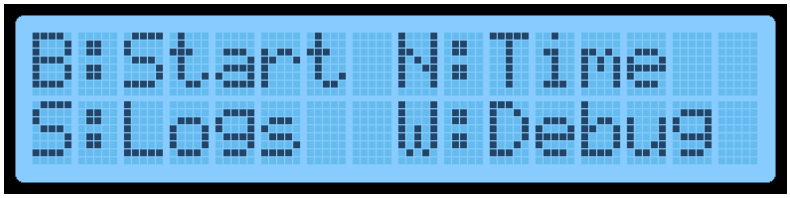


Figure 6.3.2.2: Initial Interactable Menu in UI.

The GLCD can only display images through pixels, which are the smallest elements of a picture represented on a screen. The GLCD has 128 pixels vertically and 128 pixels vertically, totaling 16384 total pixels. See figure 6.3.2.3 for a figure of the GLCD.

Figure 6.3.2.3: GLCD.

Once the operator traverses the UI and initiate operation, the operation itself must be automatic with no input from the operator that can manipulate the combinational and sequential logic which controls the operation. The operation must activate and deactivate the physical actuators and mechanisms in an order such that the input fasteners are correctly dispensed into the compartment box and the remaining fasteners are dispensed into a pickup reservoir.

After each operation, the microcontroller is to provide the display operation information at the operator's request. This operation information includes the operation time which is how long an operation takes to complete, number of remaining fasteners in each reservoir, and a summary of the instruction parameters.

Regardless of whether the operator requests the operation information, the operation information is to be stored in permanent EEPROM. The microcontroller is to store at least 4 of the most recent runs into EEPROM and this information should be able to be accessed at the operator's request through the UI.

The microcontroller is also to provide RTC Date/Time display at the operator's request. The RTC Date/Time display must update every second at intervals of 1 second, and must be accurate to the real world time within an uncertainty of $1\ second$.

Finally, after each successful operation, the operation information is to also be transmitted to a PC and presented on a PC interface. This information should then be able to be transferred to a text file on the PC.

6.3.3 Solution

Note for this section, many of the solutions require great detail to fully explain. It is implied that the solution section and the computer design subsections will be interpreted individually. It is highly recommended to consider the computer program as part of the solution because a majority of the solution from the microcontroller subsystem is the computer program.

6.3.3.1 Operation Logic

6.3.3.1.1 Design

The operation logic begins with rotating the stepper motor $360°$. This serves to open the compartment box regardless of the position at which it is inserted into the machine. The stepper motor can be instructed to perform a weaker or a strong rotation. The difference between the stronger and weaker rotations is that the stronger rotation can overcome the resistance provided between each compartment and rotate the lid between compartments but the weaker rotation cannot.

After the stepper motor has rotated $360°$ using the stronger rotation so that the lid is opened, the stepper motor begins to rotate forward using the stronger rotation again except this time, the microcontroller searches for the white electrical tape which indicates the position of C1, see figure 6.3.3.1.1. The microcontroller recognizes the tape using a black-white color sensor.

Figure 6.3.3.1.1: White Tape Indicating C1 on Compartment Box.

Once the tape has been detected by the color sensor, the stepper motor will have to rotate forward using the strong rotation for a short angle to fully open C1.

Afterwards, for 8 loops, with each loop corresponding to a unique compartment, the four vibration motors corresponding to each fastener type will be activated, the DC motors corresponding to the dispensing of the bolts and the washers will be activated, and the four solenoids corresponding to each fastener type that dispense the fasteners will extend and retract at timed intervals.

As the solenoids are constantly extending and retracting a micro switch will indicate whether or not the solenoid dispensed a fastener. The micro switch will transmit a signal to the microcontroller when the micro switch is not pressed and will stop transmitting a signal when the micro switch is not pressed. Thus when the microcontroller stops receiving signal from the micro switch, the microcontroller interprets it as a fastener being dispensed. The microcontroller then deducts the number of fasteners required to be dispensed into the compartment based on which micro switch was pressed. When the number of fasteners required to be dispensed into the compartment equals 0 for all fastener types, the loop ends and the stepper motor performs a strong forward rotation until the next compartment is open. The next loop then begins.

Once all the loops have been completed, the DC motor that controls the reservoir opening will rotate forward until the it is fully open. Then the solenoids will begin dispensing the remaining fasteners into the pickup reservoir. The dispensed microswitches will count the number of fasteners being dispensed into the pickup reservoir because this information is a component of the operator information that the operator can request. Once the maximum possible number of remaining fasteners have been dispensed or a set amount of time passes without fasteners being sensed by the micro switches, the DC motor controlling the reservoir opening will rotate in reverse, closing the reservoir opening. Then the stepper motor will perform a weak reverse rotation for 405° to completely snap and close the compartment box.

See Appendix G for a flowchart visualization of the operation logic design. The solution description of the Operation Logic continues and is explained in great detail in section 6.3.3.1.2.

6.3.3.1.2 Computer Program

The operation logic mainly serves to read inputs from the circuitry and send outputs to the circuitry. The inputs can be interpreted as values to be stored or as indicators to perform certain actions. The outputs control mechanisms set by the electromechanical member. Reading inputs and sending outputs involve using special function registers. Registers refer to a small space set in memory and registers can hold instructions for the microcontroller to perform, or values for the microcontroller to interpret. Special function registers control and monitor certain functionality in the microcontroller based on the value stored. The specific special function registers involved are PORTx, LATx, and TRISx, where x is an element of the set A, B, C, D, E. These letters refer to the pins on the microcontroller which are used to transfer voltage into the microcontroller. Each letter corresponds to 7 pins. Thus there are 35 available pins, RA0, RA1, RA2, ..., RA7, RB0, RB1, RB2, ..., RE0, RE1, RE2..., RE7. Each of these special function registers which hold 8 digit long values, the digits are referred to as bits and can only be 0 or 1. Each bit corresponds to a pin. The rightmost digit is referred to as bit 0 and the bit number increases from right to left. Bit 0 corresponds to pin 0, bit 1 corresponds to pin 1, and so on. Thus a value of 01010101 would have 1 at Rx0, Rx2, Rx4, and Rx6, and a 0 at Rx1, Rx3, Rx5, and Rx7.

Returning to the special function registers, PORTx stores whether or not an electrical signal is being input into a pin. If an input is detected, the microcontroller stores 1, and a 0 if there is no input detected. For example, if the microcontroller was receiving an electrical input only at pin RA0, the special function register PORTA would store 00000001 and PORTB, PORTC, PORTD, and PORTE would store all 0s. These values can be read using PORTxbits.Rx# where # refers to the pin number. For example, reading the value of pin RA0 would involve using “PORTAbits.RA0”.

LATx performs an opposite function to PORTx in that LATx outputs a voltage of 5V at a specified pin. For example, if the microcontroller were to output at pins RA0, RA1, and RA7, LATA would have the value 11000001 written into it by the microcontroller. The LATx registers can be written into using LATxbits.LATx#. For example, sending 5V out of pin RA0 would involve using “LATAbits.LATA0”.

TRISx indicates the “direction” the pins adopt. The direction simply refers to whether or not the pin could read an input using PORTx or send an output using LATx. A value of 1 indicates a pin can read an input and a 0 indicates a pin can send an output. The TRISx register can be written into using TRISxbits.TRISx#. Thus to set pin RA0 to read inputs using PORTA, TRISAbits.TRISA0 must be set to 1. To set pin RA0 to send outputs using TRISA, TRISAbits.TRISA0 must be set to 0 [3].

The first step of the operation logic is to rotate the stepper motor 360°. To rotate the stepper motor, 4 signals must be transmitted from the microcontroller to the stepper motor. These signals are sent with delays of 4 milliseconds (ms) and the order at which the 4 signals are sent determines both the direction of the stepper motor and the strength of the stepper motor. Labelling the four signals as signals 1, 2, 3, and 4, forward reaction requires signal 1, then signal 3, then signal 2, then signal 4 to be outputted to the stepper motor. A reverse rotation simply reverses the order at which the signals are outputted. To create a stronger rotation, intermediary signals must be provided. For example, the weaker rotation required each signal to be sent independently but the stronger forward rotation requires the following order of signals; signals 1 and 3, then signal 3, then signals 2 and 3, then signal 2, then signals 2 and 4, then signal 4, then signals 1 and 4, then signal 1. These sequence of signals must be repeated in loops, which are blocks of code that are repeated for a certain number of times or until a certain condition is met, until the stepper motor rotates a sufficient amount. A tabular simplification of

the signal requirements for the stepper motor can be seen in table 6.3.1. The pins for the stepper motor are RB0, RB1, RB2, and RB3 for signals 1, 2, 3, and 4 respectively. The RB# pins are normally reserved for the keypad but by disabling the keypad, the RB# pins become available for other functions. To disable the keypad, a signal must be sent to the KPD pin. Pin RE1 is used to send the signal to the KPD pin. The code can be found in Appendix E, lines 412-451. This code is split into two functions, one for the forward rotation and the other for the reverse rotation.

Table 6.3.3.1.1 Signal Sequence Requirements for Stepper Motor

	Forward Rotation	Reverse Rotation
Weaker Rotation	1→3→2→4	4→2→3→1
Stronger Rotation	1→13→3→23→2→24→4→14	14→4→24→2→23→3→13→1

The color sensor described in section 6.3.3.1.1 measures the color of the object in front of it by outputting a digital square wave signal to the microcontroller. The sensor will output higher frequencies the brighter the light being reflected off the sensor is. The frequency of the signal that the color sensor transmits to the microcontroller is higher than can normally be interpreted by the microcontroller. Thus, the use of the capture module is required to interpret the signal of the color sensor. As mentioned in section 6.3.1, the capture module is used to count the active highs during a period of time, an active high being when a signal reaches a peak. The peaks in this case are well defined because the signal acts as a digital square wave. Due to the possible variation in the total counts, this process is repeated 7 times and the counts are summed up. When the total count of active highs over the given period surpasses a threshold value found through experimentation, specifically 1200, the color sensor can set a flag that indicates the white tape was found. Figure 6.3.3.1.2 provides a visual example of a digital square wave signal. The digital square wave is sent to pin RC2. To use the capture module, various bits in the microcontroller must be set to various values. These exact configurations can be seen in Appendix E, lines 490-509. The entire code for the color sensor can be found in lines 484-535.

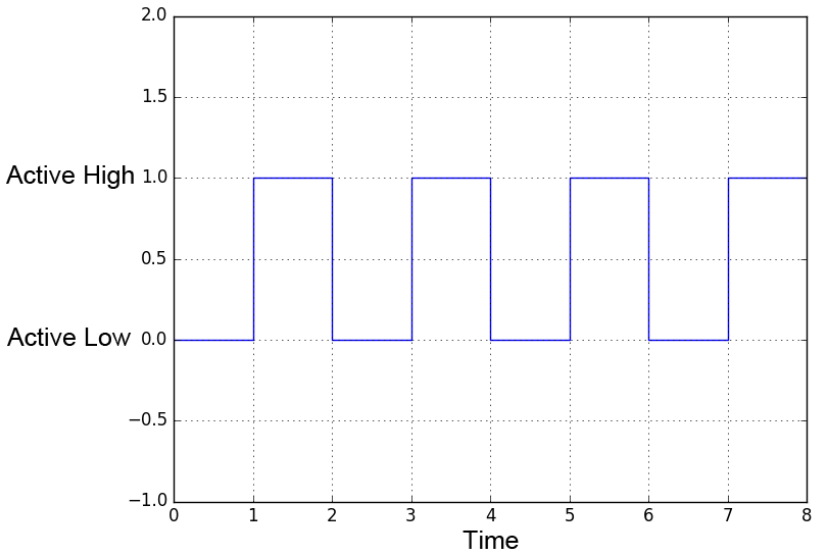


Figure 6.3.3.1.2: Example digital square wave signal where values of 1.0 are considered active high and values of 0.0 are considered active low. This figure was created using Python 3.

Next, the operation queues and dispenses the fasteners using a combination of vibration motors, DC motors, and solenoids. These mechanisms are operated by sending a voltage signal from the microcontroller using LATx. Pin RA4 sends a signal to the vibration motors for the bolts and nuts. Pin RA5 sends a signal to the vibration motors for the spacers and washers. Stopping the signal to the vibration motor will deactivate the vibration motors. The code for the vibration motors can be found in Appendix E, lines 594-611. The function controlling the vibration motors will activate the vibration motors if a 1 is input into the function and disable the vibration motors if a 0 is input. Each DC motor requires two input signals from the microcontroller. If the first input receives a signal and the second does not receive a signal, the DC motor will rotate forward. Switching the inputs such that the first input does not receive a signal and the second input does will cause the DC motor to rotate backwards. Ending transmission from the microcontroller to either inputs will cause the DC motor to stop rotating. Because the DC motors for bolts and washers only requires one direction of rotation, they only require 1 pin. The pin for the DC motor corresponding to bolts is RC7. The code can be found in lines 386-410. The pin for the DC motor corresponding to nuts is RE0 and the corresponding code can be found in pages 536-555. The function for this DC motor accepts an integer input. If the integer is 1, the DC motor moves forward. If the integer is 2, the DC motor moves in reverse. If the integer is 0, the DC motor stops moving. The solenoids act similarly to the vibration motors in that sending a signal to a solenoid will cause the solenoid to retract and stopping the signal to the solenoid will cause the solenoid to extend back to its neutral protruded position. Pins RB4, RB5, RB6, and RB7 correspond to the solenoids for bolts, nuts, spacers, and washers respectively. The code for the solenoids can be found in lines 452-483. The functions used for the solenoids extend the solenoid for 300 milliseconds (ms) and the detract the solenoids.

To recognize whether a fastener was dispensed, micro switches are placed to sense the dispensed fastener. The micro switches send signals to pins RA0, RA1, RA2, and RA3 respectively. The micro switches return a 1 if the micro switch detects an input, 0 otherwise. For every activation of a micro switch, a variable holding the number of bolts, nuts, spacers, and washers will have 1 added to the variable corresponding to the activated micro switch. The code for the micro switches can be found in Appendix E, lines 556-593. A loop indicating the compartment to be filled will only complete when each variable is equal to the number of each fastener as specified by the user. How the microcontroller acquires the user specifications are explained in section 6.3.3.2.2.

For the code that performs the overall operation, see Appendix E, lines 2198-2782.

6.3.3.2 Keypad

6.3.3.2.1 Design

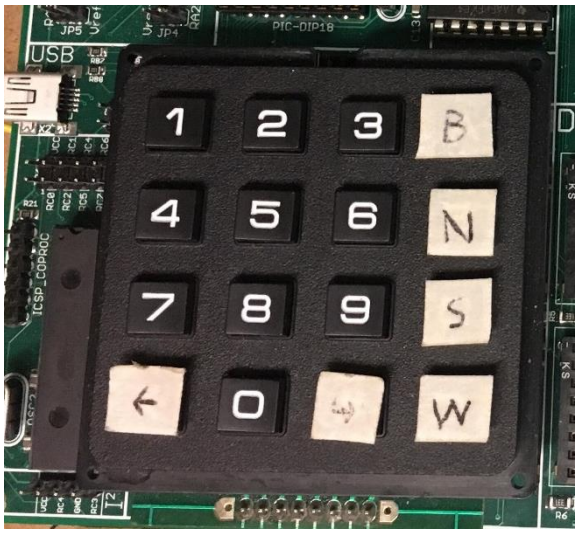


Figure 6.3.3.2.1: System Keypad.

The keypad is used to receive inputs from the operator at various points in standby mode, which refers to the state of the machine where operation is not occurring. The inputs can be interpreted by the microcontroller to perform certain actions or to store various values indicated by the operator. As seen in figure 6.3.3.2.1, the physical keypad has been modified to improve the intuitiveness of the menus.

When setting the operation instruction parameters, the operator inputs the parameters when indicated using the keypad. These values will then be stored and used during operation.

The keypad is also used to navigate through the UI. The UI will communicate valid button selections that when pressed, will move the operator to the selected menu. To do so, the key pressed in the keypad is interpreted as a character and if the character input matches specific values, the menu will change based on the value that the input character matches.

The keypad can also be temporarily disabled to enable the pins often used by the keypad to be used for other purposes. The keypad is disabled during the operation mode of the solution as operation mode is automatic and requires no operator input during the operation. To disable the keypad, 5 V must be input into the KPD pin found on the microcontroller.

The solution description of the Keypad continues and is explained in great detail in section 6.3.3.2.2.

6.3.3.2.2 Computer Program

The keypad is connected to the RB pins and an encoder is used to interpret the signals sent by the buttons of the keypad to the pins as values from the keypad. The circuit diagram for how the keypad data is transmitted can be seen in figure 6.3.3.2.2. The signals are sent through the bus to the pins which can then be converted into a value using the code “unsigned char keypress = (PORTB & 0xF0) >> 4;” This code is used

throughout the main code found in Appendix E. However, this function interprets the signal as a numerical value. This numerical value is then used as an index to an array referred to as keys. An array is a collection of values that are stored at various locations within in it. An index denotes the location of an element in an array. Arrays are indicated by curly brackets {} that contain the elements stored within the array. For example, pressing the button for # would return the value 14, which is the index at which # is stored in the keys array.

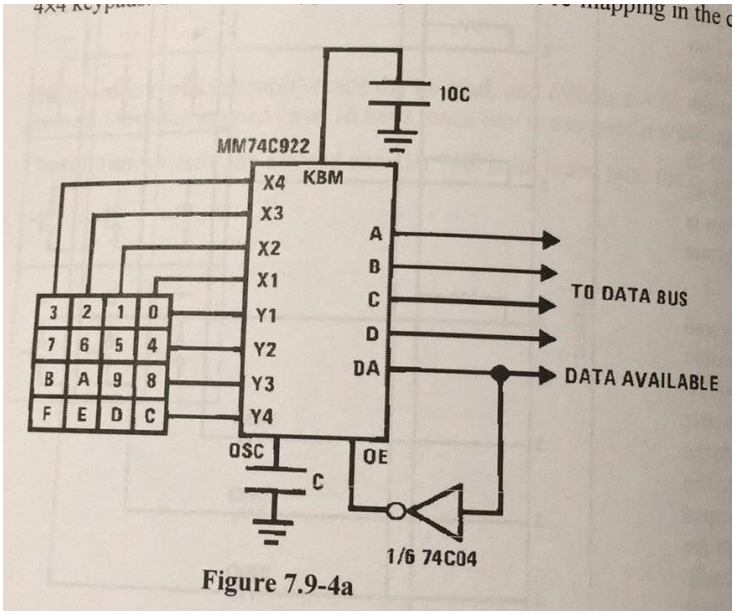
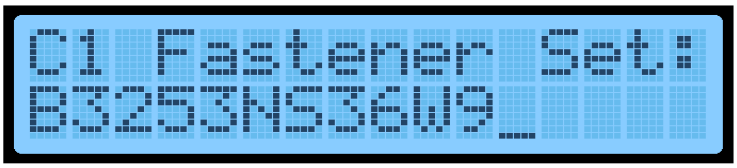


Figure 6.3.3.2.2: Encoder for Keypad. [3]

The keypad receives information by interpreting a keypad input as a character data type and storing the character into an array that emulates the displayed LCD. For example, when the user inputs B1234, the LCD will display B3253NS36W9 and the array will store the values ‘B’, ‘3’, ‘2’, ‘5’, ‘3’, etc. Most operator inputs are displayed on one line of the LCD which has space for 16 character inputs. The apostrophes indicate that the stored inputs are characters. The character data type can store all symbols but cannot be used to perform conventional arithmetic. See figure 6.3.3.2.3 for a visualization of this logic.

LCD Display:



Array: {'B', '3', '2', '5', '3', 'N', 'S', '3', '6', 'W', '9'}

Figure 6.3.3.2.3: Array Emulating LCD to Track Operator Inputs.

When the operator chooses to erase a character from the input using the clear button, the microcontroller will remove the most recently added character from the array and remove the most recently added character from the LCD, moving the cursor back by one as well. When all 16 spaces are filled with characters, no more characters can be added unless characters are erased prior. When the operator presses the enter key, the

microcontroller compares the input array with many pre-existing arrays that have valid inputs stored. The pre-existing arrays for the number of assembly steps instruction parameter are {‘4’}, {‘5’}, {‘6’}, {‘7’}, and {‘8’}. The pre-existing arrays for the fastener sets are {'B'}, {'N'}, {'S'}, {'W'}, {'B', 'N'}, {'B', 'S'}, {'B', 'W'}, {'B', 'B', 'N'}, {'B', 'B', 'S'}, {'B', 'B', 'W'}, {'B', 'N', 'W'}, {'B', 'S', 'W'}, {'B', 'W', 'W'}, {'B', 'N', 'W', 'W'}, {'B', 'S', 'W', 'W'}, {'B', 'B', 'S', 'W'}, {'B', 'B', 'N', 'W'}, {'B', 'N', 'N', 'W'}, {'B', 'N', 'N', 'N'}, and {'B', 'W', 'W', 'W'}. The pre-existing arrays for the fastener sets per step are {‘1’}, {‘2’}, {‘3’}, {‘4’}. Both the fastener set parameters and the fastener set per step parameters are requested a number of times equal to the number of assembly steps instruction parameter.

If the input array does not match any of the pre-existing arrays, the input is considered invalid and the operator is asked to repeat the input. If the input array does match one of the pre-existing arrays, the value stored within the input-array is interpreted into values that can be used for the operation. For the number of assembly steps input parameter, the quantity stored in the array is directly converted into another stored value. For the fastener set input parameter, which determines which fasteners will be dispensed into each compartment, the number of each fastener type, represented by the letters B, N, S, and W for bolts, nuts, screws, and washers respectively, the quantity of each letter within the array is counted and stored as unique values. The separate values for each compartment are then stored into a 8-element large array with an element corresponding to a compartment. For the fastener set per step input parameter, all the values for each compartment are stored in another 8-element large array with an element corresponding to a compartment. From these arrays, the fastener set is stored as a single 4 digit value. The thousands digit stores the number of bolts, the hundreds digit store the number of nuts, the tens digit stores the number of spacers, and the ones digit stores the number of washers. Thus, fastener set of BNWW would have a value of 1102 stored representing it. For the code that stores the operator instruction parameters, see Appendix E, lines 1439-2197.

6.3.3.3 Liquid-Crystal Display

6.3.3.3.1 Design

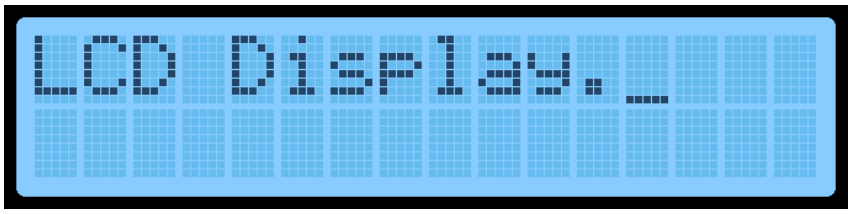


Figure 6.3.3.3.1: LCD.

The LCD used in this system includes two lines with 16 available spaces within each character. The LCD is used to present the UI and convey how the operator can navigate through the menus in tandem with the keypad to access desired information or initiate operation. Menus are a set of options presented to the operator through the UI but menus can also refer to screens that do not provide and instead communicate information.

In the solution, the menu seen in figure 6.3.3.3.2 is displayed after a brief introductory message upon powering the solution. The introductory message features an animation where the message slides from the left to the center of the LCD. This animation can be skipped by pressing any button on the keypad. The skipping is performed using the ISR which activates certain lines of code when certain conditions are met regardless of

where in the code the program is when the conditions are met. See Appendix E, lines 3753-3783 for the code of the ISR.

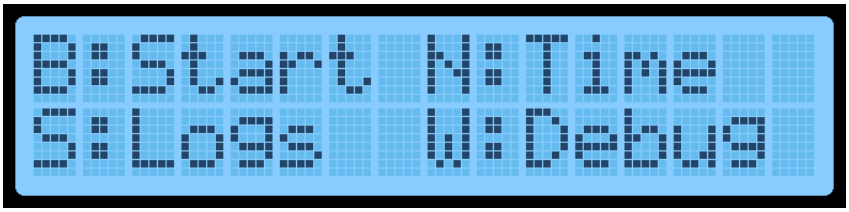


Figure 6.3.3.3.2: Initial LCD Menu.

Afterwards, the operator can choose to select B, N, S or W from the keypad as seen in figure 6.3.3.3.2 to navigate to another menu. The debug menu UI accessible by pressing W will not be detailed in this section because the debug menu is not used in the solution but rather served to facilitate the development of the solution. If the operator were to press B, the UI would begin requesting information relevant to the operation from the operator by changing to the menu seen in figure 6.3.3.3.3.

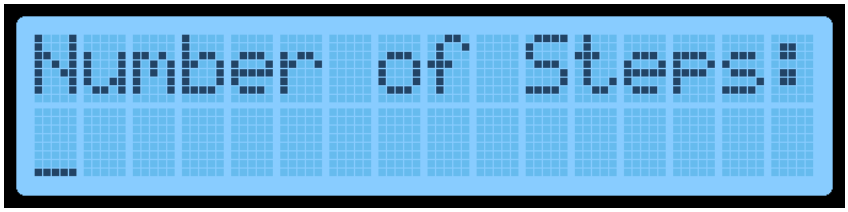


Figure 6.3.3.3.3: Compartment Faster Set Request LCD Menu.

At the menu presented in figure 6.3.3.3.3, the operator must input the desired parameters into the second line of the LCD. After a valid instruction parameter, is input, the LCD presents the menu shown in figure 6.3.3.3.4.

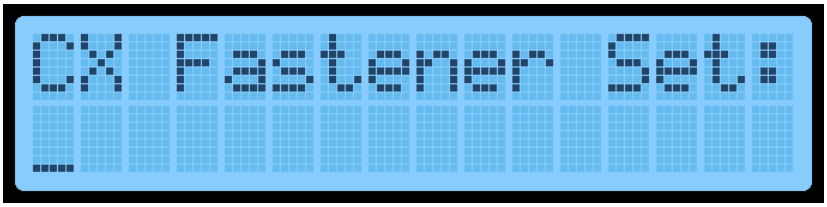


Figure 6.3.3.3.4: Faster Set Request LCD Menu.

The menu seen in figure 6.3.3.3.4 is repeated for the number of times specified in the number of assembly steps. The X seen in the figure is replaced by the number of the compartment relevant to the information being requested. The input from this menu determines the fastener set for each step. The menu then switches to the menu seen in figure 6.3.3.3.5. This menu similarly repeats for the number of times specified in the number of assembly steps with the X acting as an identical placeholder value in the figure. The menu presented in figure 6.3.3.3.4 and figure 6.3.3.3.5 differ in that the latter menu requests the fastener sets per step instruction parameter.

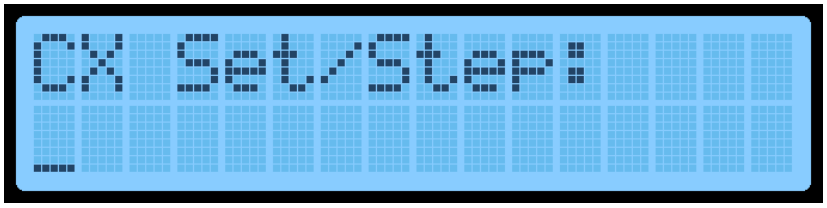


Figure 6.3.3.3.5: Fastener Set Per Step Request LCD Menu.

After the instruction parameter menus, the machine begins operation. The operation is automatic, thus the LCD simply states that the machine is in Operation Mode and prevents the operator from inputting any commands to the microcontroller during operation. The LCD menu displayed during Operation Mode can be seen in figure 6.3.3.3.5.

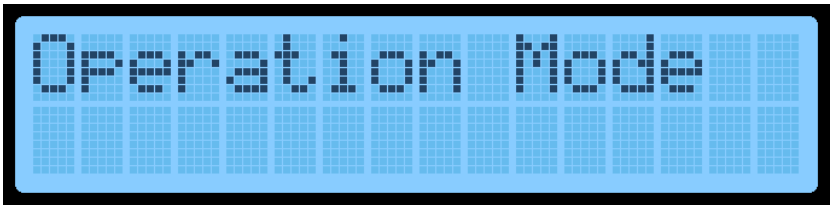


Figure 6.3.3.3.5: Operation Mode LCD Screen.

Once Operation Mode is complete, the LCD displays the option for the operator to see information relevant to the recently completed operation. This menu can be seen in figure 6.3.3.3.6.

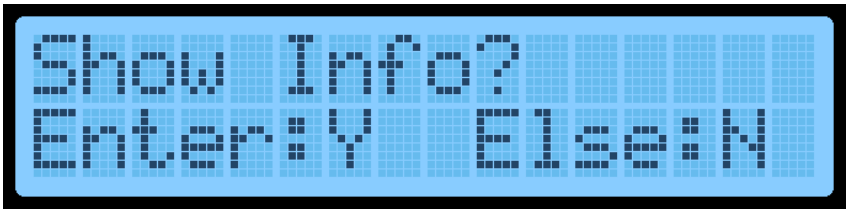


Figure 6.3.3.3.6: Additional Information Request LCD Menu.

If the operator presses any key other than enter, the UI will return to the initial menu shown in figure 6.3.3.3.2. Otherwise, the information is displayed through a series of menus that are progressed through by pressing the keypad. The first menu displays the date and time at which the operation began. This information is presented by YY/MM/DD in the first line and HH:MM:SS in the second line. This menu can be seen in figure 6.3.3.3.7. The YY/MM/DD and HH:MM:SS in figure 6.3.3.3.7 act as placeholder values in the figure to demonstrate where the actual values would be located.

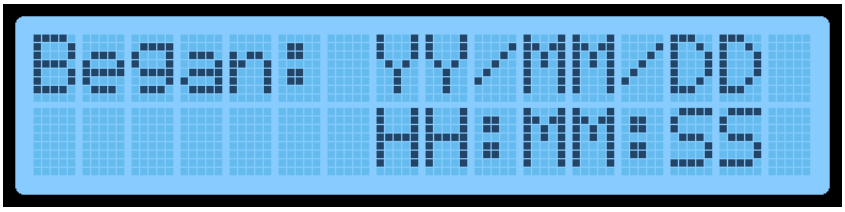


Figure 6.3.3.3.7: Starting Date and Time LCD Display Menu.

Next, the Operation Time is presented. The Operation Time is defined as the duration of the operation and is expressed in MM:SS. This menu can be seen in figure 6.3.3.3.8 where the MM:SS act as placeholder values to indicate where the actual values would be located.

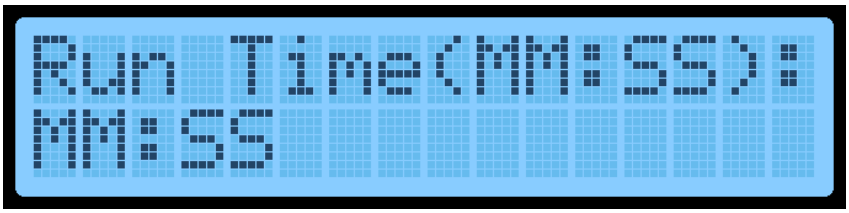


Figure 6.3.3.3.8: Operation Time LCD Display Menu.

Afterwards, the number of remaining fasteners for each fastener type. B, N, S, and W stand for bolts, nuts, spacers, and washers respectively. The menu can be seen in figure 6.3.3.3.9. The X in the figure is a placeholder indicating the location the actual values would be placed.

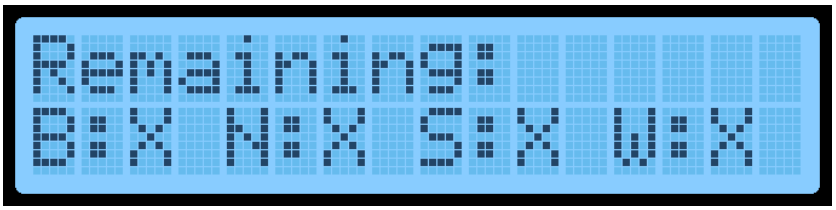


Figure 6.3.3.3.9: Remaining Fasteners LCD Display Menu.

Then the operator instruction parameters are presented. First, the number of assembly steps instructed is presented. This menu can be seen in figure 6.3.3.3.10.

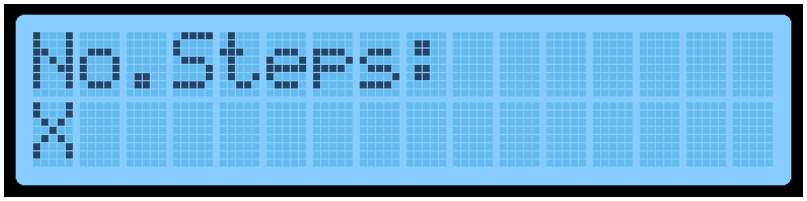


Figure 6.3.3.3.10: Number of Assembly Steps LCD Display Menu.

The fastener set instruction parameter is then presented for each compartment. For compartments that were not filled due to the selected number of assembly steps, the second line of the menu is empty. The menu can be seen in figure 6.3.3.3.11. The X in the figure acts as a placeholder to indicate where the compartment number is located. The figure displays BNNW as a one example of the faster sets that could be displayed in the LCD menu.

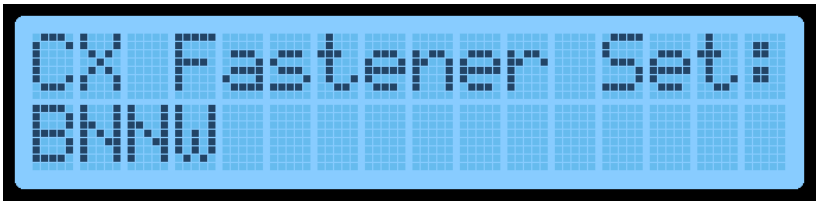


Figure 6.3.3.3.11: Fastener Set LCD Display Menu.

The fastener set per step instruction parameter is next presented for each compartment. For compartments that were not filled due to the selected number of assembly steps, the second line of the menu is empty. The menu can be seen in figure 6.3.3.3.12. The X's in the figure act as placeholders to indicate where the compartment number and fastener sets per step value are located.

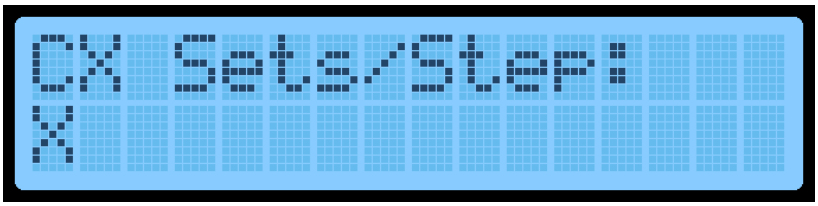
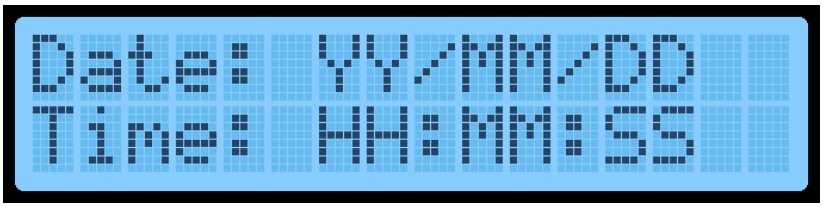


Figure 6.3.3.3.12: Fastener Sets Per Step LCD Display Menu.

Returning to the initial menu from figure 6.3.3.3.2, pressing B on the keypad will lead to a menu which displays date/time that updates in real-time. This menu can be seen in figure 6.3.3.3.13.



6.3.3.3.13: RTC-Based Date/Time LCD Display Menu.

Selecting C from the initial menu from figure 6.3.3.3.2 will lead to the logs at which point the menus will provide the choice of displaying EEPROM logs through the PIC or the PC. This menu can be seen in figure 6.3.3.3.14.

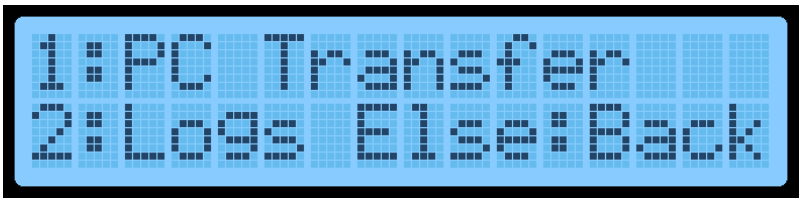


Figure 6.3.3.3.14: EEPROM PC or PIC LCD Display Menu.

When viewing the EEPROM logs through the PIC, see section 6.3.3.3.7 for more details , the same menus as 6.3.3.3.7 to 6.3.3.3.12 will be presented in the same order. A slight discrepancy is that each menu will display the log number chosen as well. Figure 6.3.3.3.15 displays the menu requesting the log input from the operator. Figure 6.3.3.3.16 displays an example of one such slightly modified menu, specifically modifying the menu found in figure 6.3.3.3.7.

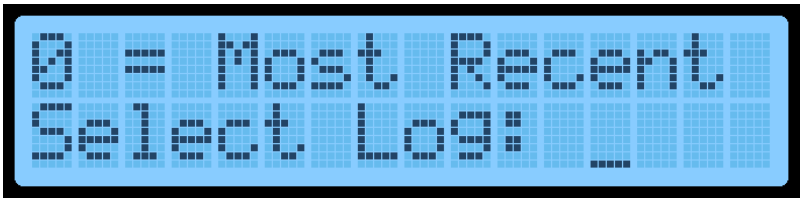


Figure 6.3.3.3.15: Log Request LCD Display Menu.

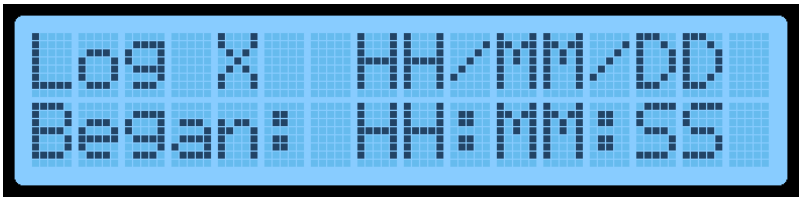


Figure 6.3.3.3.16: Starting Date and Time LCD Display Menu Modified for EEPROM Logs.

Viewing the EEPROM logs leads to the same log request as displayed on figure 6.3.3.3.16. There are no other menus because the information is then transmitted and communicated to the operator through the PC.

For more information on how the information input is performed, see sections 6.3.3.2.

The solution description of the LCD continues and is explained in great detail in section 6.3.3.3.2.

6.3.3.3.2 Computer Program

To display information on the LCD, the “print” function, which outputs a string onto the LCD display, must be used. Strings refer to arrays of values of the character data type. The character data type are simply symbols and the microcontroller cannot interpret them as having any meaning beyond visual representation. For more information on arrays, see section 6.3.3.3. The “__lcd_newline()” function, which moves the cursor to the first space in line 2 if the cursor is currently in line 1, was also used when displaying messages on the LCD. When displaying values from the RTC, the format specified %02x must be used to correctly interpret the values. For more details on the RTC, see section 6.3.3.4. For more information on the %02x format modifier, see section 6.3.3.5.2. The code used for the LCD is used consistently throughout the code found in Appendix E.

6.3.3.4 Real-Time Clock

6.3.3.4.1 Design

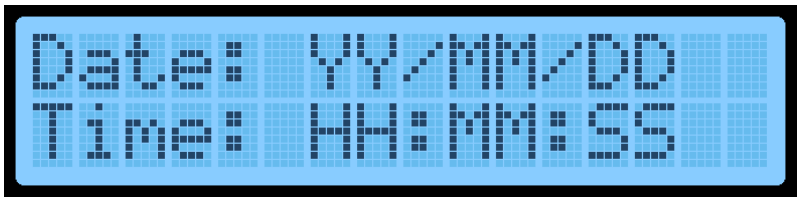


Figure 6.3.3.4.1: RTC-Based Date/Time LCD Display Menu.

The RTC is a real-time clock driven by a crystal oscillator internal of the microcontroller. The RTC will update every second accurate to the real world assuming it has been configured correctly. More information on the configuration can be found in section 6.3.3.4.2.

The RTC is used to display the real-time as seen in figure 6.3.3.3.13 in a menu accessed from the main menu, see section 6.3.3.3 for more details on the menus.

The RTC is also used to store the date/time at which the operation begins and ends, which are then used to calculate the operation time, see section 6.3.3.5 for more details on the operation time. The starting date/time is stored in EEPROM logs once an operation is complete, allowing the operator to access when the operation occurred after the operation is complete. See section 6.3.3.7 for more information on the EEPROM.

The solution description of the RTC continues and is explained in great detail in section 6.3.3.4.2.

6.3.3.4.2 Computer Program

The RTC uses the I^2C serial computer bus, which means that the RTC cannot function in tandem with the GLCD which requires the SPI serial computer bus. To alleviate this, the solution initiates the I2C using the function "I2C_Master_Init()" which disables other buses and enables the I2C bus.

The RTC works with the I2C bus by communicating to the I2C where the current date and time is stored in the microcontroller memory. This stored date and time updates every second using the internal oscillator, which sends a signal that can be interpreted as a digital square wave. After counting the exact number of active highs that pass within the span of a second, the microcontroller updates the current date and time value. See section 6.3.1 for more detail on what a signal is and section 6.3.3.1 for an explanation of what the terms square wave signals and active high refer to, and how digital square wave signals can be interpreted. The memory location must be communicated to the I2C bus because the microcontroller cannot interpret the data stored in the memory location without the assistance of the I2C bus. The function "I2C_Master_Read" causes the I2C to inform the microcontroller of the values that were interpreted from the memory location. The microcontroller then stores these values into an array where different elements of the array hold different date and time values, Before these values can be presented on an LCD, the values must be converted to strings using the sprintf function, see section 6.3.3.5.2 detail on the sprintf function. Strings refer to arrays of character data type values. The character data type are simply symbols and the microcontroller cannot interpret them as having any meaning beyond visual representation. See section 6.3.3.3 and for more details on arrays. The values must be converted to strings because the LCDs can only display character values. The converted values can then be presented on an LCD as seen in figure 6.3.3.4.1. For details on what an array is, see section 6.3.3.3. For the code that displays the RTC in the corresponding menu, see Appendix E, lines 1403-1436.

By repeatedly sending the I2C the memory location and then requesting the resultant data, the I2C will send the values of a clock that are updated in real time.

The RTC must be configured to a value to be accurate to the real-world. To do this, the function "RTC_setTime()" provided by the sample code is used to write the values from an array called date_time which contains the date/time information that the RTC will be reset to upon programming the microcontroller. The microcontroller subsystem member can store the values of the date/time corresponding to the date/time of the real world when the microcontroller completes programming. The programming process requires 61 seconds to complete. Thus, the values stored into the array date/time should be the time at which the microcontroller begins programming added by 61 seconds. The values stored into the array are identical to the hexadecimal values

returned by the RTC described earlier in this section. The code for the “RTC_setTime()” function can be found in Appendix E, lines 3787-3806.

6.3.3.5 Operation Time

6.3.3.5.1 Design

The operation time refers to the span of time elapsed between the beginning of automated operation and the end of operation. The operation time can be viewed after an operation is completed, see section 6.3.3.3 for more details.

The operation time of each run stored in EEPROM can also be accessed from the EEPROM logs, see section 6.3.3.7 for more details.

The solution description of the operation time continues and is explained in great detail in section 6.3.3.5.2.

6.3.3.5.2 Computer Program

To calculate the operation time, the RTC is used, see section 6.3.3.4.2 for more details. The relevant detail is that when the microcontroller interprets the current time, it stores the values of the current time in an array, see section 6.3.3.3.2 for more details. When displaying the RTC, these values would be overwritten to present the updated time. However, before the operation begins, the microcontroller reads the current time then stores the values in another array that will not be overwritten, this array is referred to as “start” because it holds the time at which the operation began. See Appendix E, lines 2204-2223 for the code that stores the start time. The same actions occur at the end of operation, with the current time being stored in an array referred to as “end” because it holds the time at which the operation ends. See Appendix E, lines 2787-2810 for the code that stores the start time. The values within the end array can then be subtracted by the values within the start array to acquire the operation time. The operation time displays in the solution will only present the minutes and second of the operation time because the operation time of the solution must not exceed three minutes due to a design constraint, see section 2.3 for more details on the design constraints. Thus, it can be assumed that the years, months, days, and hours elapsed during operation will always be 0. However, the solution must first overcome three obstacles.

First, the time required to read and store the time values skews the calculated operation time. However, the resultant discrepancy is in fact nominal, thus it has no discernable effect on the operation time.

Second, when the microcontroller stores the current time values, these values are hexadecimal. Hexadecimal is a numerical system with base 16 that differs from the commonly used decimal system. Normally, hexadecimal values can be easily converted to the decimal equivalent. For example 0d30 converts to 0x1E. The 0d is a prefix indicating a decimal number and 0x is a prefix indicating a hexadecimal number. Thus, 0d30 is 30 in decimal and 0x1E is 1E in hexadecimal. The obstacle with the discrepancy in number systems is that the I2C bus where the microcontroller receives the current time values, see section 6.3.3.4 on what the I2C is and how it is relevant to the RTC, sends hexadecimal values that do not directly convert. For example, when the I2C bus attempts to communicate 30 seconds, it will communicate 0x30, which converts to 0d48. To correctly convert the received value 0x30 to 0d30, the format specifier “%02x” must be used. The %02x format

specifier converts a value into their literal character data type equivalent and then removes all letters except the first two. The character data type are simply symbols and the microcontroller cannot interpret them as having any meaning beyond visual representation. For example, 0xA15F will be converted to character values "5F" which are then stored as an array. To use the %02x format specifier, the "sprintf" function can be used. The sprintf function converts all specified non-character values in a desired string and converts them into an indicated format using format specifiers. The resultant change is then outputted into another string. The converted value is stored in an array because a character data value can only store one letter, thus to store words with multiple letters, the character values must be stored letter by letter from left to right into an array one letter at a time. Arrays refer to as storages for multiple elements that are stored in a specific order. For more information on arrays, see section 6.3.3.3. An array of characters are referred to as a "string." Characters can be easily converted to their integer equivalents in the C programming language by subtracting the character "0" from the character to be converted. Therefore, each value in the array can be converted to an integer and then the integers can be manipulated to acquire the correct integer value. A visual representation of this process can be seen in figure 6.3.3.5.1. See Appendix E, lines 717-722 for the code that performs this conversion.

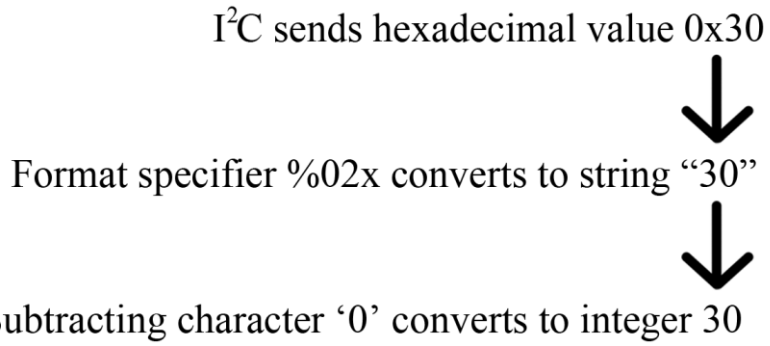


Figure 6.3.3.5.1: Visual Representation of Hexadecimal to Decimal Conversion Required for Operation Time Calculations.

The third obstacle is that the arithmetic when subtracting the values of the start array from the values of the end array must be done correctly. The values cannot be directly subtracted once they have converted to integers because certain values will reset. For example, every hour the minutes and seconds values will reset. Thus, attempting to directly subtract a start time of 14 hours, 59 minutes and 59 seconds from an end time of 15 hours, 1 minute, and 0 seconds will result with negative runtimes. This obstacle is resolved by converting the times to their total seconds then subtracting the two resultant values instead. After the subtraction the value must be converted into a minutes and seconds format which can be done using the following equation:

$$min = \left(\frac{t_{total}}{60}\right) - (t_{total}\ mod\ 60)$$
$$\text{sec} = \left(t_{total} - (min \times 60)\right)$$

Where min indicates the minutes value of the operation time in minutes, sec indicates in the seconds value of the operation time in seconds, and t_{total} indicates the total operation time in seconds. If either value is a single digit number, a 0 is outputted to the left of the min and sec values are outputted in their respective locations onto the LCD. The locations can be seen in figure 6.3.3.5.1. The MM indicates the location of the

minutes value and SS indicates the location of the seconds value in the figure. See Appendix E, lines 2813-2844 for the code that performs this calculation.

6.3.3.6 Graphical Liquid-Crystal Display

6.3.3.6.1 Design

Figure 6.3.3.6.1: GLCD.

Figure 6.3.3.6.2: GLCD Text Output with all characters and numbers displayed.

Figure 6.3.3.6.3: GLCD Progress Bar.

Figure 6.3.3.6.4: GLCD Menu Indicating Remaining Fasteners in a Step During Operation, with Progress Bar Displayed. Initial Fasteners to be Dispensed, 3 Washers, Updates to 2 Washes after One is Dispensed.

The GLCD is used to display additional information alongside the LCD to the operator, see section 6.3.3.3 for more information on the LCD. Unlike the LCD, which displays character symbols, the GLCD can only change the color of pixels displayed. Pixels refer to the smallest elements of a picture represented on a screen. The GLCD displays 128 pixels horizontally and 128 pixels vertically, totally 16384 available pixels. The colors the GLCD can display are black, grey, white, red, orange, yellow, green, blue, indigo, and violet.

The microcontroller developed a function for displaying text on a GLCD even though the function is not available on the GLC. See figure 6.3.3.6.2 for this functionality. The available rectangle output and the developed text output functions were then used to improve the UI by displaying background colors and text indicative of the step or menu the operator was in when preparing an operation, implementing a progress bar which indicated the progress of operation, see figure 6.3.3.6.3 for an image of the progress bar, and displaying the number of fasteners to be dispensed in each assembly step during assembly. The latter was updated in real time as the operation progressed. For example, an assembly step which required two bolts and a washer to be dispensed would have a GLCD display that indicated two bolts and one washer remaining for the operation step. After a bolt is dispensed, the GLCD would then indicate that there are only one bolt and one washer remaining for the operation step. See figure 6.3.3.6.4 to see this GLCD menu. See section 6.3.3.6.2 for more details on all the aforementioned GLCD functionality

Each letter and number shown in figure 6.3.3.6.2 were designed to be fit into a 4 square wide and 6 square tall design. Each square was made using 9 pixels with each side being 3 pixels long. Thus each character required 12 pixels horizontally and 18 pixels vertically, totalling 216 pixel. This size was used for spaces and the distance between two characters was 6 pixels.

The solution description of the GLCD continues and is explained in great detail in section 6.3.3.6.2.

6.3.3.6.2 Computer Program

Sample code provided by Tyler Gamvrelis provided a function called “glcdDrawRectangle()” that drew a rectangle of any shape, color, and location on the GLCD. The microcontroller member used this function to create another function called “GLCDtext()” which automatically converted and displayed text on the GLCD as desired by the microcontroller member even though the GLCD lacked the functionality to directly display text like the LCD can.

To create the characters displayed using the GLCDtext function, each character was first design in Photoshop using a 4 by 6 grid. Once the character was designed, the design was converted into an digit long sequence of 0’s and 1’s. Along each row and then along each column, a non-filled square was indicated by a 0 and a filled-square was indicated by a 1. See figure 6.3.3.6.5 for a visual representation of the conversion. These numerical conversions will be referred to as design values.

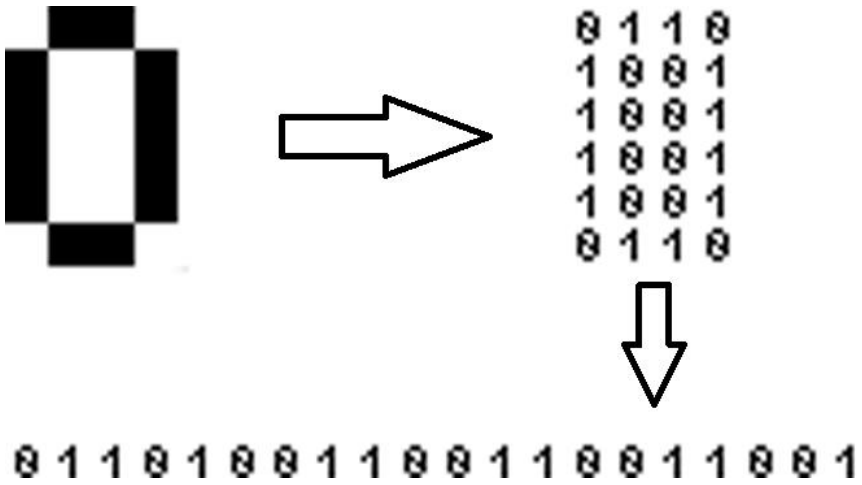


Figure 6.3.3.6.5: Diagram indicating conversion of character design into numerical value.

The design values were then stored into an array. See section 6.3.3.3.2 to see a more detailed description of an array respectively. The first 10 characters in the array are the numbers from 0 to 9 and the rest are the letters in alphabetical order. When the GLCDtext was called, a string, which is a sequence of characters, had to be input into the function. The function then interpreted each character in the string as a number indicating the position of the letter in the aforementioned array. If the character is a number, the character can be directly converted into the index, which refers to the position in the array, because arrays start counting from 0. Therefore, the design for 0 was stored in array index 0, the design for 1 was stored in array index 1, and so on. The letters were easily convertible to the correct index because each letter have numerical equivalencies that helps computers interpret characters. The system to convert between characters and the numerical equivalent as understood by computers is referred to as the American Standard Code for Information Interchange (ASCII). For example, the ASCII equivalent of the letter “A” is 65 in decimal, the ASCII equivalent for the letter “B” is 66, and so on. Since the numbers took indexes 0 to 8 in the array, subtracting 57 from each character yielded the correct index. See figure 6.3.3.6.2 for all the possible characters displayed on the GLCD.

Then the function interpreted each digit in the corresponding design value. If the function encountered a 0, the function simply moved along 12 pixels. If the function encountered a 1, the function displays a 9 pixels at a square area 3 pixels down and 3 pixels right of the current GLCD location stored by the microcontroller. This location value was then updated by 3. After 4 integers from the design value was read, the microcontroller would then temporarily go down 16 pixels to enter the next line of pixels. Once a character value was completed, the GLCD location value would be returned to what it was prior to outputting the character and then this value would be updated by 12 to the right. This function does not support text wrapping, thus each line had to be specified individually. The function accepted a number as an argument as well to indicate the line number to display the text. A 0 indicated the top line and as the number increased, the line number increased. See Appendix E, lines 649-695.

The GLCD progress bar was displayed by first displaying an empty progress bar outline onto the bottom of the GLCD using a function developed by the microcontroller member referred to as “GLCDprogressSetup()”. A variable called step initially at value 0. At every instance the function is called, the count had 1 added to it and a 1 pixel wide rectangle with the height of the progress bar was outputted at the location indicated by step. For example, at count 0, the first pixel of the progress bar would be filled. The function was then repeated

throughout operation to continuously update the progress bar. The progress bar accepted 122 calls before the operation was declared complete because the width of each line of the progress bar outline was 4 pixels. Considering 2 lines in the GLCD progress bar and a total of 128 available pixels horizontally, 122 pixels were required to be filled. See figure 6.3.3.6.3 for a demonstration of the progress bar. The code relating to the GLCD progress bar can be found in Appendix E, lines 696-713.

6.3.3.7 EEPROM

6.3.3.7.1 Design

The EEPROM is used to store the relevant operation information permanently. Permanently in this case refers to after power reset. The microcontroller often refreshes its memory upon power reset excluding the code directly programmed directly into it and the RTC values, see section 6.3.3.4 for more information on the RTC. The EEPROM memory is not erased upon power reset, thus information of previous operations can be viewed as the operator desires. The EEPROM memory is erased when programming the microcontroller but the completed solution does not require the microcontroller to be programmed again. The EEPROM stores the following information after a run:

- Number of bolts, nuts, screws, and washers remaining.
- The date/time at which the operation began, see section 6.3.3.4 for more details.
- The operation time of the operation, see section 6.3.3.5 for more details.
- The operation instruction parameters for each compartment including:
    - Number of Assembly Steps.
    - Fastener Sets.
    - Fastener Set Per Step.

The EEPROM can be displayed through both the LCD and the PC interface. The LCD interface can be found in section 6.3.3.3. The PC interface can be found in section 6.3.3.8.

The solution description of the EEPROM continues and is explained in great detail in section 6.3.3.7.2.

6.3.3.7.2 Computer Program

The EEPROM stores data by first enabling the EECON1bits.WREN bit and disabling the EECON1bits.EEPGD and EECON1bits.CFGS bits in the microcontroller. Enabling is done by setting a bit to 1 and disabling is done by setting a bit to 0. Enabling writing also requires a sequence of registers to be set to various values, similar to a password. See section 6.3.3.1.2 for more details on bits and registers. This sequence is:

1. EECON2 = 0x55
2. EECON2 = 0x0AA
3. EECON1bits.WR = 1

The 0x refers to hexadecimal values. For more detail on hexadecimal values, see section 6.3.3.5.2. Afterwards, the desired value is stored as a character to the EEDATA register where the EEDATA register transfers the data to the EEPROM by storing in the EEADRH and EEADR registers.

Reading EEPROM values simply requires reading the values at the address where the log desired is stored in the EEADRH and EEADR registers. The address refers to the location in memory at which information

is stored. Each byte of memory, which is equivalent to one piece of information to be stored, will required 1 address. The addresses go from 0x00 to 0xFF, thus there are 255 addresses to store at. Each log containing the information for 1 run requires 38 logs, allowing for 6 runs to be stored at any one time. If logs beyond a sixth are attempted to be stored, the oldest log will be deleted to create memory space. The address corresponding to the log desired can be found using the following method. First, the last address at which a log stores data contains the following value:

$$value = (log \times 38) + 1$$

Where value is the value is the value stored in the SPBRG special function register and log is the log number as specified by the operator. For example, a value of 77 would indicate that the next 38 logs contain the information for log 2. Table 6.3.3.7.2.1 provides a tabular representation of what data is stored in what order. The order at which information is stored into EEPROM is relevant because as information is stored, the address at which it is stored increases. Thus, information stored later will be stored at higher addresses.

Table 6.3.3.7.2.1: Order at Which the Value is Stored and Value to be stored for EEPROM. The initial address can be calculated using the equation from earlier in this section. This initial address will then correspond to the lowest order value address.

Order at Which the Value is Stored	Value
0	Log Completion Flag
1	Number of Bolts Remaining
2	Number of Nuts Remaining
3	Number of Spacers Remaining
4	Number of Washers Remaining
5-10	Start Date/Time
11-12	Operation Time
13-28	Instruction Parameter: Fastener Sets for All Compartments
29-36	Instruction Parameter: Fastener Sets per Step for All Compartments
37	Instruction Parameter: Number of Assembly Steps
38	Emergency Stop
39	Number of stored logs - 1.

The Emergency Stop value found from 6.3.3.7.2.1 at order 38 is left over from old legacy code that was not updated. The emergency stop was once implemented from the microcontroller side and could be activated by the operator by pressing the enter button during operation. Once the code was change to disable the keypad, see section 6.3.3.2.2 for more details on disabling the keypad, the operator input for the emergency stop was impossible to read, thus it was removed.

When requested by the operator, the information mentioned in section 6.3.3.3.4.1 were retrieved from the EEPROM using the methods explained earlier in this section and then displayed.

The code for the EEPROM can be found in Appendix E, lines 225-381.

6.3.3.8 PC Interface

6.3.3.8.1 Design

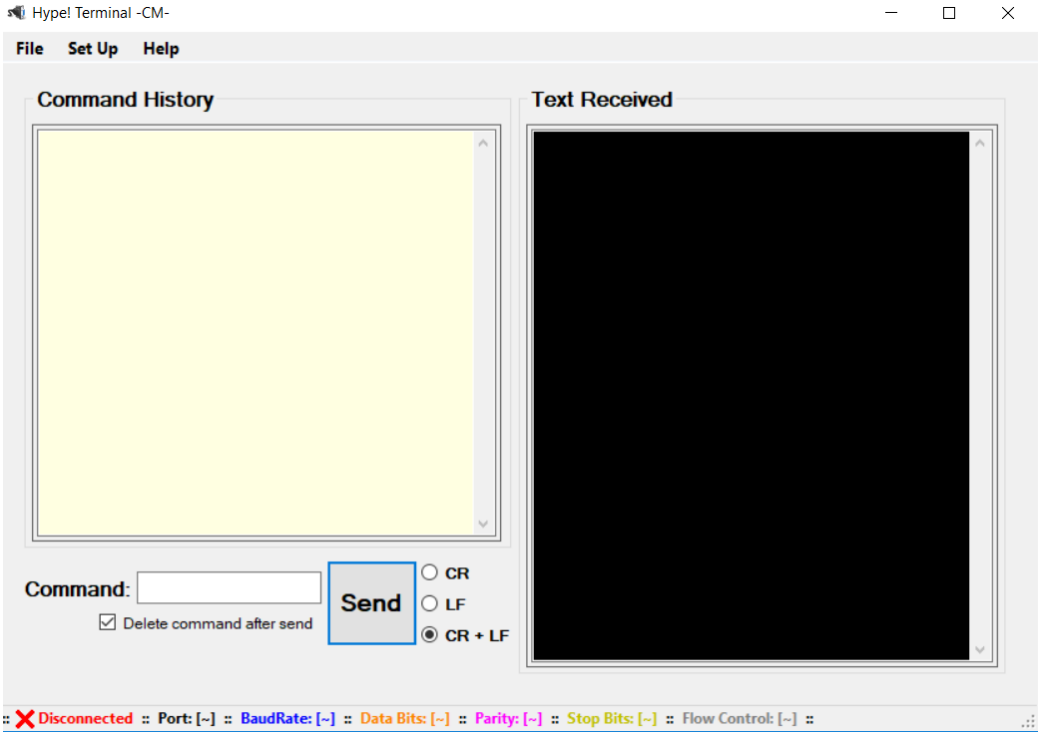


Figure 6.3.3.8.1: PC Interface Presented through Hype! Terminal.

The PC Interface refers to information from the microcontroller being presented on an external PC. The PC interface is used to display the operation information automatically after an operation is completed and to display EEPROM information when requested by the operator. The following information will be displayed on the PC interface in both instances:

- Number of bolts, nuts, screws, and washers remaining.
- The date/time at which the operation began, see section 6.3.3.4 for more details.
- The operation time of the operation, see section 6.3.3.5 for more details.
- The operation instruction parameters for each compartment including:

- Number of Assembly Steps.
- Fastener Sets.
- Fastener Set Per Step.

The operator can request the EEPROM to be displayed through the LCD UI. See section 6.3.3.3.1 for more information on the PC EEPROM display through the LCD UI.

When displaying the information on the PC Interface, the information is specifically displayed on a hyperterminal program called Hype! Terminal. A hyperterminal is a program that communicates with COM ports on the computer. A hyperterminal can both send and receive information through these ports. For this solution, the hyperterminal is only required to receive information. Hype! Terminal is a simplified hyperterminal that enables for simple presentation of the information from the microcontroller. See figure 6.3.3.8.1 for an image of Hype! Terminal without any inputs and without the cable connecting the microcontroller and the PC connected.

See figure 6.3.3.8.2 for an example of the PC interface immediately after an operation. See Appendix E, lines 2861-3116 for the code that displays the PC Interface immediately after Operation.

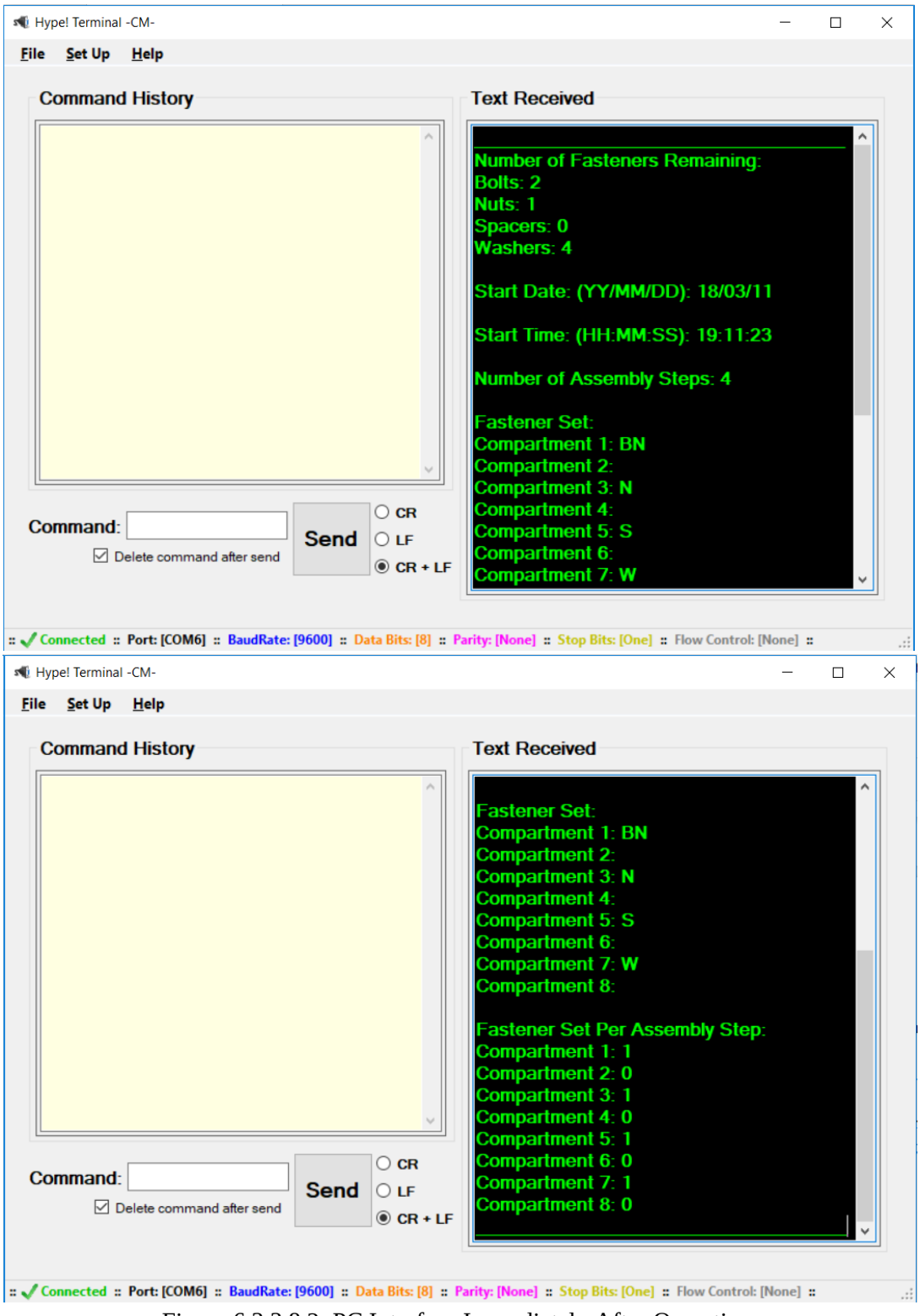


Figure 6.3.3.8.2: PC Interface Immediately After Operation.

See figure 6.3.3.8.3 for an example of the PC interface for EEPROM. See Appendix E, lines 3314-3570 for the code that displays the PC Interface for EEPROM.

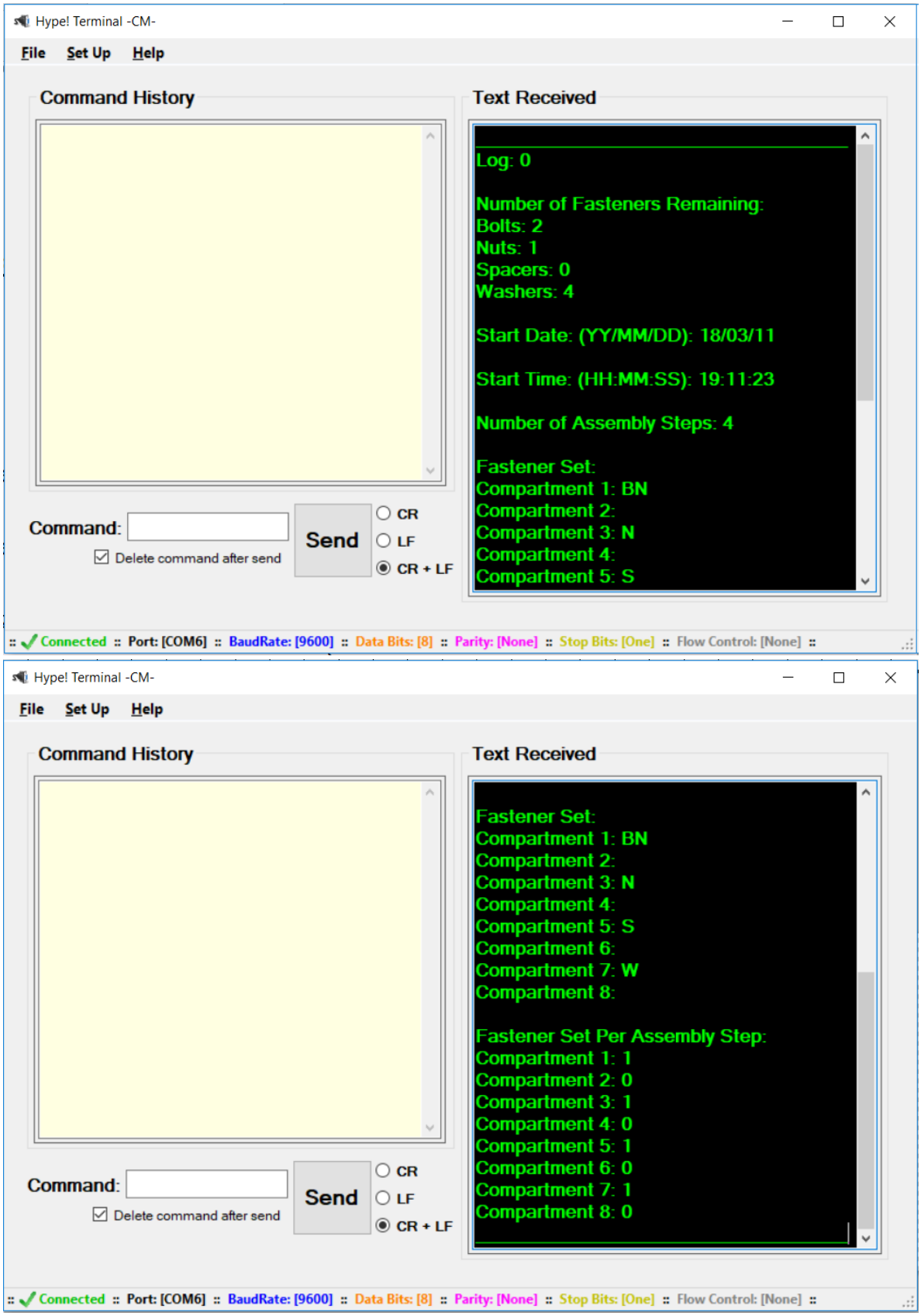


Figure 6.3.3.8.3: PC Interface For EEPROM

Hype! Terminal allows for the operator to directly copy the text displayed on the hyperterminal and paste it into a separate file such as a text file. This text file can then be stored and saved to be viewed on the PC at a later time.

The solution description of the operation time continues and is explained in great detail in section 6.3.3.5.2.

6.3.3.8.2 Computer Program

The information to be displayed on the PC interface is transmitted by the microcontroller using the Universal Asynchronous Receiver-Transmitter (UART). UART enables a sequence of 0's and 1's to be sent through the UART transmission pin, defaulted to RC6, see section 6.3.3.1.2 for more information on pins. The sequence sent through the RC6 pin must be transmitted through a RS232 adapter into a Universal Serial Bus (USB) cable. The USB is then connected to a COM port in the PC which is where the PC receives the sequence. The sequence is then converted by a hyperterminal to display values [8].

To begin using UART, certain bits need to be enabled and disabled. See Appendix E, lines 621-636 to see these bit configurations.

While setting the bits, the special function register SPBRG must also be set. The SPBRG determines the rate at which 0's and 1's are transmitted from the RC6 pin. Having an incorrect rate of transmission can cause the values interpreted by the hyperterminal to be incorrect. Calculating the SPBRG value to be stored is done using the following equation:

$$SPBRG = (f/(64 * baud)) - 1$$

Where SPBRG refers to the value to be written into the SPBRG special register, f refers to the frequency of the oscillator driving the microcontroller, which is 40000000, and baud refers to the baud rate. The baud rate is the number of signal changes per second. The baud rate is often 9600. Thus, a baud rate of 9600 would yield a SPBRG value of approximately 64. This value did not work with the PIC. Thus, an oscilloscope was used to determine the baud rate by counting the number of signal changes per second. The oscilloscope output for the RC6 pin can be seen in figure 6.3.3.8.4. The pin is attempting to use UART to output "E" through RC6 because the sequence of 0's for "E" is 01010101010101, which happens to simplify the process for determining the baud rate [8].

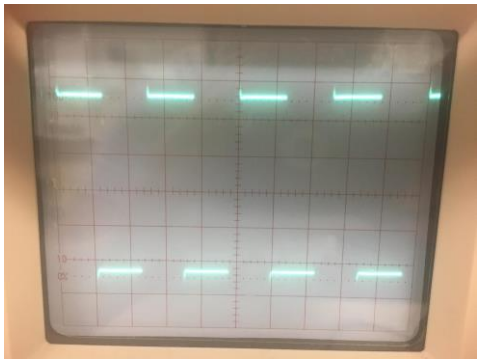


Figure 6.3.3.8.4: Oscilloscope Output for Pin RC6.

Using the oscilloscope, the baud rate was found to be approximately 2480, resulting with a SPBRG value of 251.

6.3.4 Suggestions for Improvement

The microcontroller subsystem could be improved by first cleaning and optimizing the code. The code for the microcontroller is very inefficient and often takes less optimized, simpler method that increase the processing time of the code. One notable example of the inefficient code is the calculation for the number of each fastener required from operation based on the fastener set. The calculation is based on a 4 digit value where the thousands digit stores the number of bolts, the hundreds digit store the number of nuts, the tens digit stores the number of spacers, and the ones digit stores the number of washers. See section 6.3.3.2.2 for more details on how the 4 digit value was acquired. The calculation used in the code repeatedly takes the 4 digit value to calculate the separate fastener amounts but this calculation could be optimized by using previously calculated values. For example, once the number of bolts is calculated, the resultant number of bolts can be subtracted from the 4 digit value after dividing the 4 digit value by 100 to find the number of nuts.

Another area for improvement is the EEPROM. The implementation of EEPROM is inefficient because one log requires 38 bytes to store, see section 6.3.3.7 for more information on EEPROM. Due to the maximum EEPROM available memory of 256 bytes, the current solution can only store 6 logs. Although this number of logs is above the minimum required amount of logs, the number of logs capable of being stored by the PIC can be improved through the following methods. First, values that are being stored in separate bytes can be combined when being stored. For example, the number of remaining bolts, nuts, spacers, and washers are stored as 4 individual bytes but could be stored as 2 bytes by combining them into a pair of 2 digit long numbers. The reason why a single 4 digit number is not suggested is because a 4 digit number requires more memory than a byte, thus it cannot be stored into a byte.

Another area for improvement is to remove legacy code. Legacy code refers to code that was used in older iterations of the code and eventually became obsolete but were not removed. The most notable example of remaining legacy code is that an earlier iteration of the code had emergency stop from the software side implemented. During operation, the operator could a button on the keypad and cancel the operation immediately. However, this functionality was removed but the operation still checks for the emergency stop at certain parts of the code and the EEPROM still uses a byte per log to store whether or not the run had been stopped by the software-side emergency stop or not.

7.1.7 Pin Assignments

The pin assignments are represented in table 7.1.7.1. Pin assignments indicate how the pins, which are components used for the microcontroller to send and receive electrical signals, receive electrical signals which are meant to correspond to various mechanisms.

For more information on how the mechanisms are connected to the pins, see section 6.2. For more information on how the pins interact with the microcontroller, and how the microcontroller accepts inputs and sends outputs to the pins, see section 6.3. Section 6.3.3.1.1 contains great detail on the pins.

The red cells in the table indicate pins that are unavailable for interacting with mechanisms due to the fact that writing to the pins would affect other aspects of the PIC. B, N, S, and W represent bolts, nuts, screws, and washers respectively.

Table 7.1.7.1: Pin Assignments

Pin	A	B	C	D	E
0	Micro Switch B	Stepper Motor	DC W		DC Dispenser
1	Micro Switch N	Stepper Motor	DC Dispenser		Keypad Disable
2	Micro Switch S	Stepper Motor	Color Sensor		
3	Micro Switch W	Stepper Motor			
4	Vibration BN	Solenoids			
5	Vibration SW	Solenoids			
6		Solenoids	UART		
7		Solenoids	DC B		