Master Lock®
Combination Cracker

Matt Leotta
Patrick LaRocque
Jessica Tse

12/10/02
Microprocessor Systems
Section 03

Table of Contents

List of Figures

List of Tables

Abstract

The Master Lock® Combination Cracker has two goals: to open the lock given a three-number combination and to crack the combination given the Master Lock®. The Combination Cracker is a device consisting of several mechanical components which operate jointly and are controlled by the Motorola M68HC12 microprocessor through software written in C. A 12-volt DC motor is used to rotate the lock dial and is the chief component in determining which of the possible combinations will open the lock. A pull-type, 48W solenoid pulls up on the arm of the lock when the system is opening or cracking the combination. A sixteen-character, two-line LCD screen and a Greyhill series 96 4x4 keypad provide the interface for data input and output for the user of the device.

The Combination Cracker provides a creative and clean method for opening a Master Lock® when the user enters the combination and enables the user to retrieve a lost combination for a given lock quickly, consistently, and efficiently. With the short waiting time (maximum of five minutes) for the determination of the unknown combination, the Master Lock® Combination Cracker is clearly a viable alternative to the more destructive options available for opening locks with forgotten combinations. The following report describes in detail the design, construction, and methodology involved with the Combination Cracker. In addition, it also includes electrical schematics, software source code listings, and other diagrams to further describe the system and device.

1 Introduction

The state of today's society calls for secure storage of one's material property. From this, there is inherently a need for small personal locks; one of the most common of which is the combination Master Lock®. In certain cases, the lock owner may not be able to open the lock easily, due to arthritis or general lack of manual dexterity. In other cases, the combination to the lock may be forgotten and the owner would like to open the lock without damage and retrieve the combination so the lock may be reused. The typical lock owner would only need this service on rare occasions, so this device might be more fitting for a locksmith. Other users of this device could be owners of storage facilities or health fitness centers. In both businesses, customers often are allowed to use there own locks to temporarily store person belongings. When customers fail to remove their locks after the allotted time span, the owner is forced to cut them off. The Master Lock® Combination Cracker provides an alternative in which the owner may remove the lock and then reuse or resell it at a later date.

According to information on the Master Lock® website, retrieving a lost combination for a lock is much more difficult now than it was in the past. For security reasons, Master Lock® no longer provides combinations over the phone. Extensive paper work and notarization is required before this information is given out. In some cases this procedure may not be possible or may take too long. This is no longer a problem with our device.

The Master Lock® Combination Cracker automatically opens Master Lock® brand combination padlocks. The device uses a motor and solenoid to manipulate the lock, a keypad and LCD display for user I/O, and a Motorola 68HC12 microcontroller to interface these components and provide a software-based PID controller for the motor. The completed system opens any Master Lock® either by accepting a three-number combination from the user or by experimentally determining the combination through trial and error. We were also able to significantly reduce the time required to find the combination by using a known code-cracking methodology. This enhancement requires precise interaction between the solenoid and motor to "feel" the notches in the rotating cam within the lock. For the purposes of our prototype, the lock is placed inside the device, proving the design concepts for opening the lock. A final production model would be made more portable, and be able to be placed on the lock when it is still attached to something.

2 Materials and Methods

This section details the materials and methods used to implement the Master Lock® combination cracking device. The various mechanical, electrical/electronic, and software components of the system are discussed in the following subsections.

2.1 Mechanical Components

The mechanical components of the system are used to physically manipulate the lock. These include solenoid for opening the lock and a motor with integrated gearhead and encoder for entering the combinations. Additionally, a variety of parts are used to position and secure the motor, solenoid, and lock and to interconnect the motor and solenoid with the lock.

2.1.1 Solenoid

Mounted vertically above the lock is a pull-type, 48W solenoid (See Figure 10.). A solenoid was chosen for the task of pulling the arm of the Master Lock open for two reasons. The first is that it fires rapidly, and the second is that it has plenty of power. We completed an experiment on a lock to determine how much force is required to pull the arm and open the lock. The lock was attached upside-down to a test apparatus, and weights were added slowly to a hook attached to the arm of the lock with the correct combination already entered. The following data was collected on ten trials:

Run	Mass (grams)	Mass (ounces)
1	1300	45.856
2	1250	44.092
3	1250	44.092
4	1250	44.092
5	1220	43.034
6	1240	43.740
7	1260	44.445
8	1260	44.445
9	1150	40.565
10	1200	42.329
Average	**1238**	**43.669**

Table 1: Lock Opening Force Test

With this information, we knew that the solenoid needed to be strong enough to pull with at least 44 ounces of force intermittently. Intermittence was chosen due to the fact that the solenoid only needed to pull quickly at the lock to open it, and hold up briefly when trying to crack the lock. The Deltrol D-70 solenoid proved to be strong enough, and exactly what was required for the job. Figure 11 shows a plot of force versus stroke distance for this particular solenoid. Opening a lock requires about 0.5 inches of stroke. The solenoid was purchased from All Electronics Co. Since the solenoid requires 48W, and the lab power supplies can only supply about 6W, we needed to use an additional power supply. For this, we used standard 250W ATX computer power supply.

A special apparatus was constructed to connect the solenoid plunger to the lock arm. We needed a way for the solenoid to tightly grip the arm, yet also be able to attach and detach from the lock without needing to open the lock. The resulting grip connector is shown in Figure 8. Two metal clamps are positioned to hold the arm of the lock. These clamps are set to pivot about a pin so that they may separate and release the lock. However, when the other end of the pin is inserted into a second hole the clamps are locked together. This pin is also used as a contact point for success switch lever.

2.1.2 Motor

Mounted horizontally in front of the lock dial is a precision DC motor. This small motor contains an integrated gearhead and optical encoder (See Figure 9.). It was donated by MicroMo Electronics, Inc. Unfortunately, the unit is an outdated model and MicroMo was not able to provide the exact specifications. We do know that this is a 12VDC, 22mm diameter motor. The maximum speed after gear reduction is approximately 150 RPM.

The encoder provides a two channel quadrature encoded output. Through experimentation we determined that each channel provides 82900 pulses per revolution of the output shaft. However, we do not know how much of this is due to the resolution of the encoder and how much is due to the gear reduction. Through comparisons to similar devices currently manufactured by MicroMo, we estimate that the encoder provides about 1024 pulses per revolution and the gear reduction ratio is approximately 81:1.

The output shaft of the motor is connected to the dial of the Master Lock® using a specially designed socket. The socket was molded from 3/4 inch diameter electrical heat shrink tubing which shrinks to a minimum of a 1/2 inch diameter. A 5/8 inch diameter pinion was attached to the output shaft of the motor and aligned with the lock dial. A 3/4 inch long section of the heat shrink tubing was placed over pinion and dial. This tubing was heated with a butane torch to provide a form fitting rubber socket. To ensure a tight grip with minimal slippage, we wrapped wires tightly around the tubing over the dial, pinion, and shaft (See Figure 9.).

2.1.3 Frame

When planning the construction of the apparatus, it was important to consider where each of the mechanical components would be placed. The forces produced within the complete assembly are shown in Figure 13. The frame was built around the components so that each part would function as desired. Wood was chosen as the building material of choice since it is strong yet easy to shape and work with. As can be seen in Figure 5, most of the prototype body was built with wood. The first step in the construction involved attaching the solenoid and lock to a vertical piece of plywood measuring 4" x 11.5". The lock is held onto the prototype by a slot cut out of the wood with the exact dimensions of the lock, ensuring that there would be no slippage, and that the lock would be held securely in place. This piece with the slot was elevated from the piece of plywood mentioned earlier with another piece in between to align the solenoid plunger with the lock properly. The success switch was mounted next to the solenoid also with a spacer, so that the switch arm could be triggered by the grip locking pin on the solenoid plunger. This switch was carefully positioned so that it would only be triggered when solenoid plunger moved far enough to open the lock.

Two pieces of plywood also measuring 4" x 11.5" were attached to the sides of the prototype allowing it to stand freely, and to provide platforms to mount the track for the motor. The motor was affixed to a piece of sheet metal measuring 4" x 3.5". This piece of sheet metal glides freely between two tracks attached to the side support pieces of wood. Two springs were attached to the sheet metal on one end, and to the vertical portion of the apparatus on the other. The springs apply force holding the motor to the face of the lock; they also allow the motor to smoothly return to operating position. Sheet metal was chosen for implementing the base to attach the motor to for two reasons. The first is the strength; aluminum is an adequately strong material for mounting to. The second reason is the flexibility. A thin piece of sheet metal allows for a small amount of play in the movement of the motor, allowing it to twist and turn just slightly to accommodate for slight misalignment of the shaft and dial.

2.2 Electrical and Electronic Components

The electrical and electronic components of the system include the power switching circuitry for the motor and solenoid, the encoder decoding and counting circuitry, the keypad, and the LCD display. These components are the links between the mechanical components and the software components. Each of these components is described in detail below.

2.2.1 Power Switching

All of the power switching circuitry is found on a small circuit board mounted on the back of the combination cracking device. This small driver board is shown in Figure 6 and the corresponding schematic is shown in Figure 3. The two ICs and one DIP relay are connected to the board with sockets so that they can be easily replaced if they burn out. The driver board is powered separately by a computer power supply (via a standard 4-pin connector) to provide the additional power needed to drive the solenoid. Low power control signals are communicated to and from the protoboard via a 10-pin dual-row header connected to a 3 ft. ribbon cable. A special DIP-like adaptor was created to separate the pins enough to plug into the protoboard (See Figure 7.). An 8-pin single-row header is used to connect to the motor ribbon cable. One of these pins is not used and plugged on the socket, thus the remaining seven pins can not accidentally be connected backwards. Two additional 2-pin connectors are used to attach the solenoid and success test switch.

An active low signal from the protoboard is used to drive the solenoid. This signal is fed through an inverter and then to the base of an NPN transistor. The emitter of the transistor is used to drive the coil of a double-throw double-pole DIP relay. When activated, the relay completes the 12V solenoid circuit. Both switches in the relay are used in parallel because of the large amount of current. Each switch is rated for 2A and the solenoid draws 4A total. A transistor is needed to drive the 5V relay coil because the 7404 Hex Inverter chip does not provide enough power.

The motor is powered with an SN754410NE Dual H-Bridge IC. The H-Bridge maps input logic signals (0V or +5V) to the required output provided by V_{CC2} (+12V). A motor direction control signal is passed through inverters and then to the H-Bridge in both its inverted and original form. This ensures that outputs to the motor are 0V and +12V, but outputs are swapped depending on the state of the motor direction control bit. When this bit is high, Motor + gets 12V, Motor – gets 0V, and the motor spins clockwise. When it is low, Motor + gets 0V, Motor – gets 12V, and the motor spins counter-clockwise. However, both of these outputs to the motor are set low if the enable input is low. By sending a pulse-width modulated (PWM) signal to the enable bit, the motor is quick switched on and off. Thus, the duty cycle of the PWM signal controls the effective voltage to the motor which controls the motor's velocity.

The two feedback signals from the encoder are passed from the driver board directly back to the protoboard. The switch that monitors the lock's position also sends a signal back to the protoboard. This signal is normal pulled high with a 1KΩ resistor. The seventh and final signal communicated between the driver board and protoboard is a common ground. This is needed because driver board and protoboard are powered with different power supplies. The signals sent between the two boards are summarized in Table 2. Direction is with respect to the driver board.

Pin #	Name	Direction	Description
1	GND	Bidirectional	Common ground between boards
2	*unused*		
3	CH B	Output	Channel B encoder signal from motor
4	*unused*		
5	CH A	Output	Channel A encoder signal from motor
6	*unused*		
7	TEST	Output	Success test signal from switch (active low)
8	MOT PWM	Input	Motor PWM signal
9	SOL EN	Input	Solenoid enable signal (active low)
10	MOT DIR	Input	Motor Direction signal (1=CW, 0=CCW)

Table 2: Control Signals Sent to Driver Board

2.2.2 Encoder Decoding and Counting

The protoboard contains the circuitry to decode and count the encoded position signals (See Figure 16). The two encoder signal (CH A, and CH B) are fed into a HCTL-2016 Quadrature Decoder/Counter chip. This chip is designed to sample the channel inputs and increment or decrement a 16-bit counter accordingly. The counter is then read by the 68HC12 as if it were a memory chip. Eight data lines are provide by the HCTL-2016, and these are connected to Port D (J8 pins 29 – 36) of the EVB (See Figure 2.). The high byte and low byte of data must be read on alternate clock cycles through the same data pins. Thus, the HCTL-2016 would interface easily with the 68HC12 if it were in Normal Expanded-Narrow mode. However, the 68HC12 is locked into Normal Expanded-Wide mode by the EVB. This still works, but the HCTL-2016 must be mapped to alternating odd addresses. The bytes are accessed individually and then combined in the software.

To actually access the HCTL-2016 registers from within the 68HC12, address decoding is required. This is done with some external glue logic on the protoboard. The HCTL-2016 is mapped to the $2000 to $2FFF address space. Although only two addresses are actually needed, the logic is simpler when the entire range is used. The upper four addresses lines, Port A 4-7 (J9 pins 3-6), are passed through a 4-input NAND gate after inverting the appropriate signals. This produces a low asserted signal whenever the most significant nibble is a binary 0010 ($2). This signal is combined with the low asserted CSD signal with an OR gate. Thus the resulting signal goes low only when accessing memory in the $2000 to $2FFF range.

The HCTL-2016 also requires clock, select, and reset signals. The clock is used as the sampling period for the input channels A and B. For this we used the E-clock signal provided by PE4 (J8 pin 34). The frequency of the clock must be at least three times faster than the fastest frequency of the encoder channels for correct operation. At the maximum speed of150 RPM, the frequency of the encoder channel is a little over 20KHz. The E-clock runs at 8MHz so this is definitely sufficient.

The select signal is used to select which byte (high or low) is presented on the data bus. We used PB1 (J8 pin 60) for the select signal; this signal is address line 1. Since we can only access the HCTL-2016 with odd addresses, this causes the high and low bytes to map to alternating odd addresses.

The reset set signal is provided by G1 (J8 pin 16). This low asserted signal must be manual asserted in the software whenever the chip needs to be reset.

2.2.3 LCD and Keypad

The LCD and Keypad hardware was wired exactly as described in the ECSE-4790 Microprocessor Systems: Motorola 68HC12 User's Manual. Refer to the schematic in Figure 4.

A single unit containing both the LCD and keypad components was provided by the lab to be interfaced with the 68HC12. The keypad was connected to the microcontroller through Port J to allow control by the Key Wakeup Interrupt. In the keypad, as shown in Figure 4, eight pins connected to the EVB, where four were used for the row select and four were used for the column select. The column select pins were connected with pull-up resistors. The four row select pins were connected to the four higher bits of Port J (bits 4-7) and were configured as output bits that were normally held low. When a button was pressed, one of the column select pins was then pulled low, triggering the Key Wakeup Interrupt. The Key Wakeup Interrupt had to then determine which row was activated and return the correct character for that button. Refer to Section 2.3.3 for the software involved with the interrupt handling.

The sixteen-character by two-line LCD and corresponding controller were connected to Port H and three open pins of Port G for data lines and control bits. This enabled the microprocessor to feed in the appropriate characters to the LCD to be displayed. Bit PG5 was used as an enable bit, PG3 indicated read/write direction, and PG4 corresponded to RS on the LCD controller. The remaining eight lines (Port H) were used as the data lines. The LCD control is further described in Section 2.3.4.

2.3 Software Components

The software components consist of five main parts: combination cracking, motor control, keypad, LCD, and overall integration (main program). Each of these is discussed in detail in the following subsections.

2.3.1 Combination Cracking

Three main steps are involved with cracking the combination. First, the last number of the combination must be found. Every recent Master Lock® has twelve notches in one of its cams. The notches can be felt by pulling up on the latch of the lock while rotating the dial until it gets stuck. Eleven of these notches are decoys and one corresponds to the last number in the combination. Traditional methods of combination cracking rely on estimating which notches are centered over which numbers on the dial. Of the twelve notches, seven of them are between numbers in the dial, and four of the remaining notches have the same ones digit (e.g. 8, 18, 28, 38). The last number in the combination is the remaining number corresponding to the twelfth notch. This method is rather imprecise and difficult to implement in code. Fortunately, through experimentation we determined that the width of the real notch is slightly wider than that of the decoys. Although average notch size varies from lock to lock, the real notches tend to be around 1.5 times as wide as the decoys. Although this is difficult to detect by hand, it is rather easy for our device to take these measurements. The motor 'feels' where the 12 notches are around the dial of the lock and measures each. The largest notch corresponds to the last number in the combination.

Second, the first and second numbers in the combination are calculated. The first number modulus 4 (firstNum%4) will be the same as the last number modulus 4 (lastNum%4), and the second number plus 2, modulus 4 ((secondNum+2)%4) will be the same as the last number plus 2, modulus 4 ((lastNum+2)%4), so there are 10 possible numbers for each the first and second numbers of the combination. These numbers were stored in two arrays of 10 `char` values each.

Third, the possible combinations are tried in a particular order to minimize the number of rotations that need to be done to the dial. The combinations start with the smallest possible first number. The initial second number is the first possibility to the right of (or larger than) the first number. The combinations are tried in order from there. When done in this order, the first three full rotations are not needed between combinations that differ only in the second number. After the first number is entered, the dial is rotated to the second number and then the third number, and if this combination is not successful, the dial is moved back to the next second number and then the third number, and so on. After all of the second number possibilities are tried, the full three rotations are done to initiate the set of combinations with a new first number. For example, a possible sequence could be:

```
2-4-18
2-8-18
2-12-18
2-16-18
2-20-18
...
2-36-18
2-0-18
6-8-18
6-12-18
...
```

This methodology allows the system to move through the combinations much more quickly, and allows lock to be cracked within five minutes or less.

The function `findPossibleCombos()` finds the possible first and second numbers in the combination that correspond to the third number that is returned from the motor control code. It determines the order in which the possible combinations should be tried and makes the appropriate calls to the motor functions.

After trying the combinations, a message is displayed to the user indicating whether or not the attempt to crack the lock was successful. This is done in `crackCombination()`. If it was a successful crack, the message "`Cracked Combo:`" is displayed with the correct combination. Otherwise, if the system was unable to open the combination for any particular reason, the message "`Could not crack lock combo`" is displayed to the user. Theoretically, the system should always be able to find the combination of the lock unless a mechanical part of the system fails, so this message would likely not be displayed. All of the code performing this functionality can be found in Appendix E in the `main.c` source code.

2.3.2 Motor Control

Since the motor used is not a stepper motor, precise positional control must be obtained through a feedback loop. The motor feedback is provided by the integrated optical encoder. Pulse counting is done in hardware as described above. The motor control portion of the code relies heavily on the real time interrupt service routine to perform most of the work. At regular intervals, the position counter is read and the motor PWM is updated accordingly. All of the motor control is contained in `motor.h`, which is listed in Appendix E.

The only restriction on the RTI period is that it must occur often enough so that the positional register in the HCTL-2016 does not overflow. Since the 2-byte register is read as a signed integer, and the direction of the motor is variable, the RTI must occur before 32768 encoder counts occur. At the maximum of 150 RPM with 82900 counts per revolution, the RTI must occur at least once every 158 ms. We decided to run the RTI much more frequently (every 8.192 ms) for faster response and better control.

The motor speed itself is controlled by producing a pulse-width modulated signal. This is done automatically by the 68HC12 by configuring the timer output compares accordingly. TOC0 is configured to set Port T pin 0 on each successful compare of _H12TC0. Similarly, TOC7 is configured to reset Port T pin 0 on each successful compare of _H12TC7. The first of these registers is adjusted to adjust the duty cycle and the latter is permanently set to 0.

The RTI and PWM described above are initialized in a routine named MotorInit(), which must be called early in the main program. This routine also initializes some global motor control flags and sets the default state for the motor PWM and solenoid position. To make the code more readable, many of the low level hardware commands have been encapsulated in macros. Macros include setting motor direction, setting solenoid state, and reset the pulse width. Each macro involves accesses a specific bit on one of the ports using masking. The RESET_PULSE_CNT() macro, for example, sets bit 1 of Port G low and then immediately high again; this creates the short low asserted pulse needed to reset the HCTL-2016 register.

After it is initialized, the RTI service routine occurs once every 8.192 ms. On each occurrence, the change in position is read from the HCTL-2016 by accessing the bytes pointed to by *pulsesHigh and *pulsesLow (addresses $2001 and $2003 respectively). The HCTL-2016 is then immediately reset. These two bytes are combined to form the signed integer that represents the change in position since the last RTI. This is effectively the rotational velocity measured in encoder counts per RTI (roughly revolutions per 679 seconds).

The absolute position of the motor is found by integrating rotational velocity over time. In the discrete domain, this is done by taking the summation velocities at each time step. This value would quickly overflow a 16-bit register after less than one full rotation. For this reason we sum the velocities after dividing them by 16. With this method, up to ten revolutions can be completed without overflow. At this resolution, there are still about 130 counts between each number on the dial. Dividing by 16 provides an accurate enough representation of position while allowing for multiple rotations without overflow. However, simply dividing by 16 introduces error in the position count due to the truncation of the least significant nibble. The truncated nibble must be stored as well, and added to the next velocity value before division. If this is not done, a small amount of error will accumulate in opposite directions for clockwise and counter clockwise rotations. The number of counts in one rotation is approximately 82900/16 = 5181 and is represented by the defined constant: ONE_ROTATION.

After determining the current position and velocity, the error in velocity and position are calculated. The error in velocity (actually the error in speed) is calculated as the difference between the desired speed and the absolute value of the velocity. This value is multiplied by a gain factor of 2. The error in position is calculated as the difference between the current position and the target position.

The error in position is used to determine the desired velocity depending on the programs current state. If the motor is not currently active (that is, it should not be moving), then the desired velocity is 0 regardless of the error in position. It is assumed that the target position value (and therefore positional error value) is only valid when the motor is active. When the motor is active and feeling a notch, the desired velocity is always 200. However, when the motor is active and moving to a new position, the desired velocity is directly related to the error in position according to the simple velocity profile shown in Figure 1. In general, this means that the motor spins at a relatively fast constant velocity when far (more than a half of a rotation) from the goal. As it approaches the goal, velocity decreases linearly with distance to goal. When it is within three numbers (27 degrees) of the goal, deceleration is even faster. The velocity and deceleration values were determined experimentally to find a good balance between speed and accuracy.

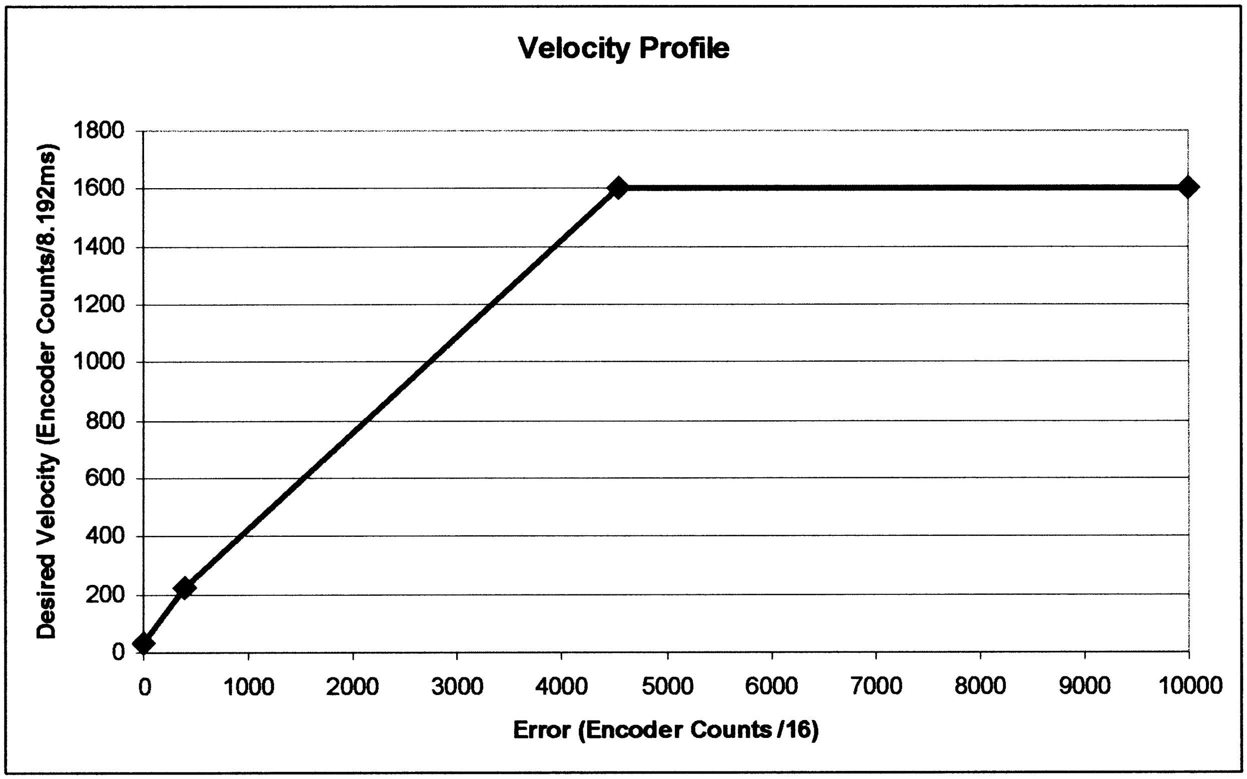


Figure 1: Motor Velocity Profile

In each RTI, the motor PWM is updated according to the error in speed. The scaled error factor is added to the current PWM value so that the motor continually adjusts its PWM to meet the desired output speed. The advantage of this method is that it causes the device to perform uniformly over differences in friction. Thus, a rusty old lock can be cracked as accurately and quickly as a brand new lock. The motor will simple need to apply more torque to meet the needs of the rusty lock. This property of the feedback system is exploited in the lock feeling process discussed later.

To initiate a motor movement, the `MotorMove()` function is called. This function sets a targets position and the direction of motion before enabling the motor by setting the `motorActive` flag. This flag indicates that the motor is active and also indicates the commanded direction: 0 for off, 1 for clockwise, and -1 for counter-clockwise. The calling code must then wait for the flag to be cleared before making the next move.

The `DoCombo()` function makes use of the `MotorMove()` function to enter a three-number combination. This function accepts the three combination numbers and then calculates the corresponding motor positions. The absolute position is set back by three rotations so that the motor will then move forward three rotations, clearing the lock, before entering the combination. Due to the cyclical nature of the numbers on the dial, additional rotations are inserted depending on the size of each number relative to its neighbors. After determining the absolute positions, the dial is moved sequentially to each position using MotorMove(). While waiting for each motion to complete, a while loop monitors the motorKill flag. This flag is set by a keypad interrupt and indicates that the current combination should be aborted. If an abort is detected, the value 2 is returned. Otherwise, this function returns the value of OpenLock().

The NextCombo() function is similar to the DoCombo() function but enters only the last two numbers in a combination. This is useful for when trying multiple combinations in a row when each combination has a common first number. If done correctly, the last two numbers can be entered without disturbing the first.

The OpenLock() function activates the solenoid for a short time while monitoring the status of the success test switch. If the switch is triggered, then the lock was opened and the function returns a 1. If not, the solenoid is deactivated, releasing the latch, and the function returns a 0.

The GetLastNum() function is used to feel the notches in the lock and determine the last number. As described in Section 2.3.1, the motor must measure the width of each of the twelve notches in the lock. A for-loop with twelve iterations is used to accomplish this. Each iteration involves measuring one notch and storing its width and value only if wider than the current widest notch. The motor is then moved approximately one twelfth of a rotation so that the next notch can be felt. When all twelve notches have been measured, the dial is moved back to zero and the last number is returned. As in the combination entering procedures, a motorKill signal causes the routine to end prematurely. In this case, 40 (the first invalid value) is returned.

The FeelNotch() function is used to actually take measurements on the current notch. First, the solenoid is activated. Then, the motor moved left once and right once to stick the dial in one of the notches (if not there already). As mentioned earlier, feeling exploits the fact the motor PWM is continually adjusted to meet the desired speed. If the dial is stuck, the motor torque will increase until it causes movement or it reaches its maximum. During "feeling" a low desired velocity and high torque threshold (PWM value limit) are set. The motor attempts to move until reaching this threshold. Two more of these "feeling" motions are performed in each direction. The motor position is stored in an array after each motion. The width is calculated as the difference between the average left and right side measurements. The dial number is determined by averaging all measurements and converting back to dial coordinates. These results are returned to GetLastNum() via pointers passed into the function.

If any of these function are aborted by the users, the calling program is responsible for detecting the failure return value and then calling MoveToZero(). This function moves the dial back to zero and then clears the motorKill flag.

2.3.3 Keypad

In order to get the Greyhill series 96 4x4 keypad to interface with the rest of the system, a header file was written containing the Key Wakeup Interrupt initialization routine and Interrupt Service Routine.

The keypad itself was connected to the microcontroller with the Key Wakeup Interrupt of Port J to allow the keypad to operate on an interrupt-based method. In the keypad, as shown in Figure 4, there were 8 pins connected, where 4 were used for the row select and 4 were used for the column select. The 4 row select pins were connected to the 4 higher bits of Port J (bits 4-7), which were configured as output bits that were normally held low. When a button was pressed, one of the column select pins was then pulled low, triggering the Key Wakeup Interrupt. The

Key Wakeup Interrupt had to then determine which row was activated and return the correct character for that button.

In the header file, the function `initKeyWakeupInterrupt()` was written for the initialization for the Key Wakeup Interrupt. This function mapped the ISR (`keyWakeupISR`) to PortJKey. Various settings were then set for Port J to accomplish the following: detect falling edges with Port J, select pull-ups for Port J, enable pull-ups, enable bits 0-3 (column selects) for generating interrupts, set up row select bits (4-7) for output, and write lows (0's) to row select bits in Port J. This was all accomplished in the code in Appendix E with the `keypad.h` source code.

The second function of the header file was the Interrupt Service Routine, called `keyWakeupISR()`. First, the state of the Key Wakeup Flags (contents of register `_H12KWIFJ`) was saved. The values of the 4 low order bits on Port J were then isolated by using the following line, which provides a mask for the lower 4 bits with the 0x0F:

```
  lowOrderBits = keyWakeupFlags & 0x0F;
```

Next, the ISR parsed through the different rows to determine which key was pressed. To do this, a high value was first written to row 1 (`_H12PORTJ = 0x10`), and the lower 4 bits were checked. Then, each consequent row was then checked (up to and including row 4). If the row written to was the row in which the button was pressed, the value on the low bits of Port J would be 1111. If this were the case, the function `determineKey()` checked the previously saved 4 low bit values to see which column the key was pressed in. If a key of column 1 was originally selected, the low bits would read 0x1. If column 2 was the selected column, the saved low bits would have a value of 0x2. If column 3 was selected, low bits would be 0x4. Also, if column 4 was selected, the low bits would be 0x8.

The `determineKey()` function then determined the value corresponding to the row and column of the key pressed. For example, if the row pressed was 1 and the column was 1, the key pressed was 'D'. If the column was 2, the key pressed was '#', if the column was 3, the key pressed was a '0', and if the column was 4, the key pressed was a '*'. In the next row (if the button was pressed in row 2), if the column was 1, the key pressed was 'C', if the column was 2, the key pressed was '9', etc. The value of the key was saved as a global variable so that the value could also be accessed from within the main program including the header file.

The key '*' was also used as a global exit of the system. In other words, when this key was pressed, two additional flags were set. The flag `exitState` was set to a 1 to indicate that the program was in a state to exit. Also, the flag `motorKill` was set to a 1 to stop the motor in any function it is executing and return to the zero notch on the dial.

After the determination was made of which key was pressed, the output bits of Port J were all reset to low outputs (the 4 high bits were set to low). The function waited until the key was released (which caused the lower 4 bits of `_H12PORTJ` to be 1111), and then the Key Wakeup Flags were cleared by writing a 1 to the bit in the flag register that corresponded to the interrupt that was processed. This was done by writing the value of `_H12KWIFJ` back to itself, such as with the following: `_H12KWIFJ=_H12KWIFJ;    // Clear the flag`

The value 0xFF was not simply written to the `_H12KWIFJ` register, since more than one interrupt may have set the flags and would then be lost because they were not processed yet by other ISRs, and it is better practice to clear only the bit of the interrupt the ISR specifically handles.

In order to utilize the header file for the keypad with the rest of the system, the `initKeyWakeupInterrupt()` function was called to initialize the Key Wakeup Interrupt, and the keyPressed variable was set to be 'Z' initially (this value cannot be achieved using the keypad as its values currently stand). The user was prompted for input to be entered entirely with the keypad by using the following type of code:

```
// loop until a key is pressed on keypad
while (keyPressed == 'Z') {}
selection = keyPressed;
keyPressed = 'Z';
```

Multiple digit numbers were entered by checking each digit entered by the user and keeping a running total of the values. As a new digit was entered, the previous value was multiplied by 10 and then the new digit was added to this value. This continued until some other key was pressed other than a digit from 0 to 9. The code to accomplish this can be found in Appendix E.

2.3.4 LCD

The microprocessor communicated with the LCD panel using functions written in a lab-provided header file. These functions were called from the main.c source code to display various messages to the user. First, the LCD screen had to be initialized, the address had to be set to zero, and the display and cursor were turned on by calling the following, respectively:

```
OpenXLCD(0x3F);
WriteCmdXLCD(0x80);
WriteCmdXLCD(0x0C);
```

Two functions were also written in main.c to simplify calls to print messages to the LCD screen. These two functions are called clearDisplayAndPrint(char *str) and clearDisplayAndPrint2(char *str1, char *str2). The first of these two functions prints a message on a single line, whereas the second of these functions outputs a two-line message on the LCD screen. For single characters, the function writeChar(char cursorLoc, char c) was written to easily specify a particular character to be written at a given cursor location on the LCD screen. Also, in order to output a given combination that the system was using to try to open the lock, the function outputCombination() was written, which also provided the formatting for the combination. These functions can be found in Appendix E in the main.c source code.

2.3.5 Main Program

The code which integrates the code for the motor control, the LCD, and the keypad can be found in the main.c source code found in Appendix E. The user control of the system was based on menu options and selections. The selection the user made through the keypad determined the function performed by the motor.

When the system is first started, the user is presented with the opening message "Welcome to the UnLock Helper" and is informed that the key '*' can be used at any time while running the system to exit and return to the main menu. Views of the LCD screen displaying these messages can be viewed in Appendix B Figure 14. The main menu is displayed giving two options:

```
[A] Enter combo
[B] Crack it
```

If the option was selected to enter a combination, the user would be prompted to enter the three numbers of the combination:

```
Enter combo:
__-__-__
```

After each two-digit number was entered, the value would be verified to be in the acceptable range of 0 to 39. If the number entered by the user was outside of this range, an error message would be output to the LCD screen indicating this:

```
Number entered
must be 0 to 39!
```

and the cursor would be returned to the starting digit of that number in the combination.

Backspace functionality was added to allow the user to erase a previous digit entered or delete a previously entered number of the combination and reenter it by pressing the 'A' key. During this time that the user is prompted for the combination, only the digits '0'-'9', 'A' (for backspace), and '*' (to exit and return to the main menu) are allowed to be entered. This constraint is established by iterating through a `while` loop until the user has entered valid key. This code can be viewed in Appendix E in the `inputDigit(char, char, char)` function, which takes in the cursor location, which number is being input, and which digit of the number will be entered next. This allows the individual digits of the combination to be tracked.

The function `inputNumber(char, char, char)` combines the two digits entered to represent a number and converts them into a decimal value. When the second digit is entered, the first converted number is multiplied by 10 and added to the second converted number in order to achieve this.

Once the combination is retrieved from the user, the motor is repositioned to zero and the combination is tried by calling the function `DoCombo(char, char, char)`. If the function returns a 0, the lock has not been opened and an appropriate message ("Could not open...") is displayed with the unsuccessful combination. If the lock has been opened successfully, the function will return a 1, indicating that the proper combination was entered, and a message is output to the user: "`Opened Lock:`" with the combination displayed.

From this, the main menu is then displayed and the user makes the selection to either enter a combination or crack the lock. In the case where the user enters a 'B' at the main menu, the system will attempt to find the combination by trying 100 combinations (as opposed to all of the possible combinations) until the lock opens. Refer to Section 2.3.1.

3 Results

The Master Lock Combination Cracker overall successfully accomplishes all of the original goals set for the system: to open a Master Lock given a combination from the user and to crack the lock by finding the combination independently. Three different Master Locks were used to test the system, including an older lock and two newer locks, whose combinations were `1-3-25`, `2-28-18`, and `30-0-22`, respectively. The system was consistently able to open the locks given the appropriate combination.

Also, the combination-cracking functionality was tested on each of the locks. The entire combination cracking process was timed at approximately 4 minutes and 40 seconds. This is time required to feel the notches are try all 100 remaining combinations. The feeling process takes about 40 seconds and combination testing takes about 4 minutes. This maximum run time is independent of the particular lock in use. On average the run time is expected to be about 2 minutes and 40 seconds assuming equal distribution of combinations over all locks.

For the locks with the combinations `1-3-25` and `2-28-18`, the system was able to easily find the combinations and open the locks very quickly since the combinations start with low numbers.

In fact, one of the locks was opened on the first attempt resulting in a 45 second total run time. The final lock took considerably more time to crack (around 4 minutes). We noticed that the success rate for cracking this lock was not as high. We had no failures attempting to find the last digit in any of the locks. We also had no failures in attempting to crack the first two locks. However, failure occurred when attempted to crack the final lock. The reason for this failure is not entirely certain. It could have been due to an accumulation of positional error due to slippage in the dial socket. It also could have been due to overheating of the solenoid causing a reduction in the pull force. Finally, the code could contain a bug which causes this particular combination to be entered incorrectly. More testing would be required to determine exactly what the cause of this problem was, but we ran out of time.

In one respect, the system proved to be more robust than we had originally planned for. We had designed the system such that the dial must be initial placed at the zero location as a point of reference. It turns out that our code will crack the lock regardless of the initial position. However, each digit in the returned combination is shifted off by the amount of the initial error. We were able to crack our third lock by starting at a different location such that the combination was found in less time.

By using the algorithm for finding the possible combinations based on the knowledge of the last number in the combination, possible combinations are brought down from 64,000 possible combinations to only 100 combinations. The average number of combinations that need to be tried by the device to crack any given lock is E[x]=50, where x = 1..100. Given that trying the possible one hundred combinations takes four minutes, each combination on average takes approximately (4 min)(60 sec/min)/100 combinations = 2.4 seconds per combination. If the device had to go through all 64,000 possible combinations, the system would have to run for 2560 minutes, or more than 42 hours! This time would be far worse if combinations were not tried in the ideal order and the lock was cleared between each combination. The Combination Cracker reduces the total time it would take to run through all possible combinations by a factor of 640, and thus implements a much more efficient code-cracking algorithm.

4 Discussion

All of the initial expectations and goals for the Master Lock® Combination Cracker were met by the target date of the end of the semester of Fall 2002. The desired functionality for the device and system was implemented completely, and the system runs reliably with most locks. Given more time and resources, a smaller, more portable device would have been developed which provides the same features as the prototype that was actually built. A smaller, cordless apparatus would be more easily usable for this application, since in most cases the lock would be attached to the object it was locking at the time. Also, with an extended timeline, more testing would be done to verify that the dial socket that attached to the motor to the lock was secure enough. Slight inaccuracies in the motor rotating the dial were likely due to the fact that there was slippage in the connection with the socket. If more resources were available, other lightweight, tighter-fitting options for the socket would have been tested with the other components of the device. However, given the time and supply constraints on the project, the resulting system that was developed and implemented as a Combination Cracker was highly satisfactory and sufficiently performed all of the desired features to open and crack a Master Lock®. The project proved to be challenging and educational, and overall it was an excellent learning experience for the integration of all elements of microprocessor systems.

5 References

Master Lock "Master Lock: Lost Combination FAQ's." 2002. 8 Dec. 2002 <http://www.masterlock.com/general/faqs_lostcom.shtml>.

Rosenberg, Lee. "ECSE-4790 Microprocessor Systems: Motorola 68HC12 User's Manual". Rev. 1.1, 10 Aug. 2000.

6 Bibliography

Hillson, Nathan. "Master Lock Combination Cracking." 19 June 2000. 8 Dec. 2002 <http://www.people.fas.harvard.edu/~hillson/master_lock.html>.

MicroMo Electronics, <http://www.micromo.com>

All Electronics Corporation, <http://www.allelectronics.com>

Jameco Electronics, <http://www.jameco.com>

Allied Electronics, <http://www.alliedelec.com>

Deltrol Corportation <http://www.deltrol.com>

Appendix A: Electrical Schematics

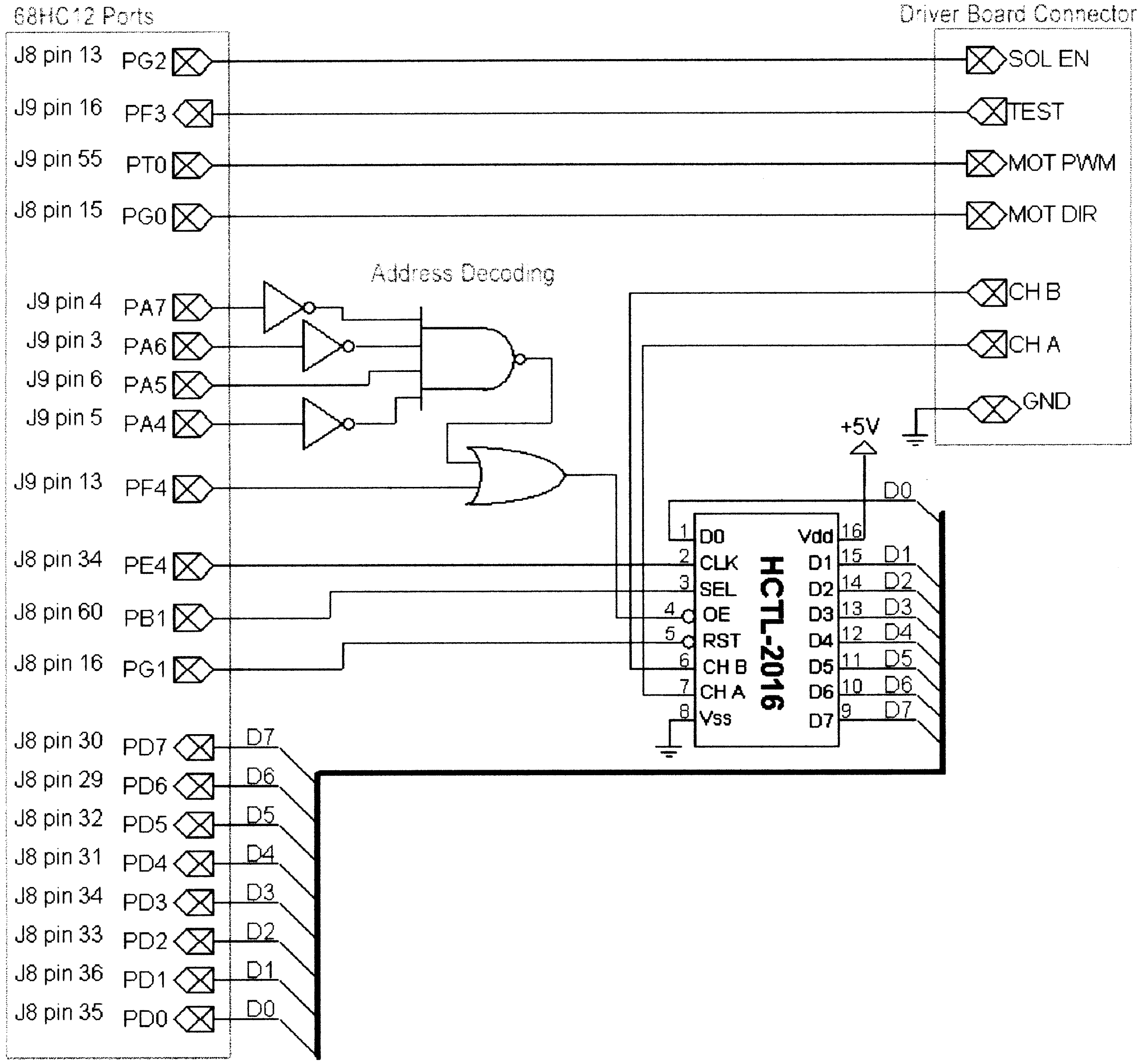


Figure 2: Protoboard Circuitry - Motor Control

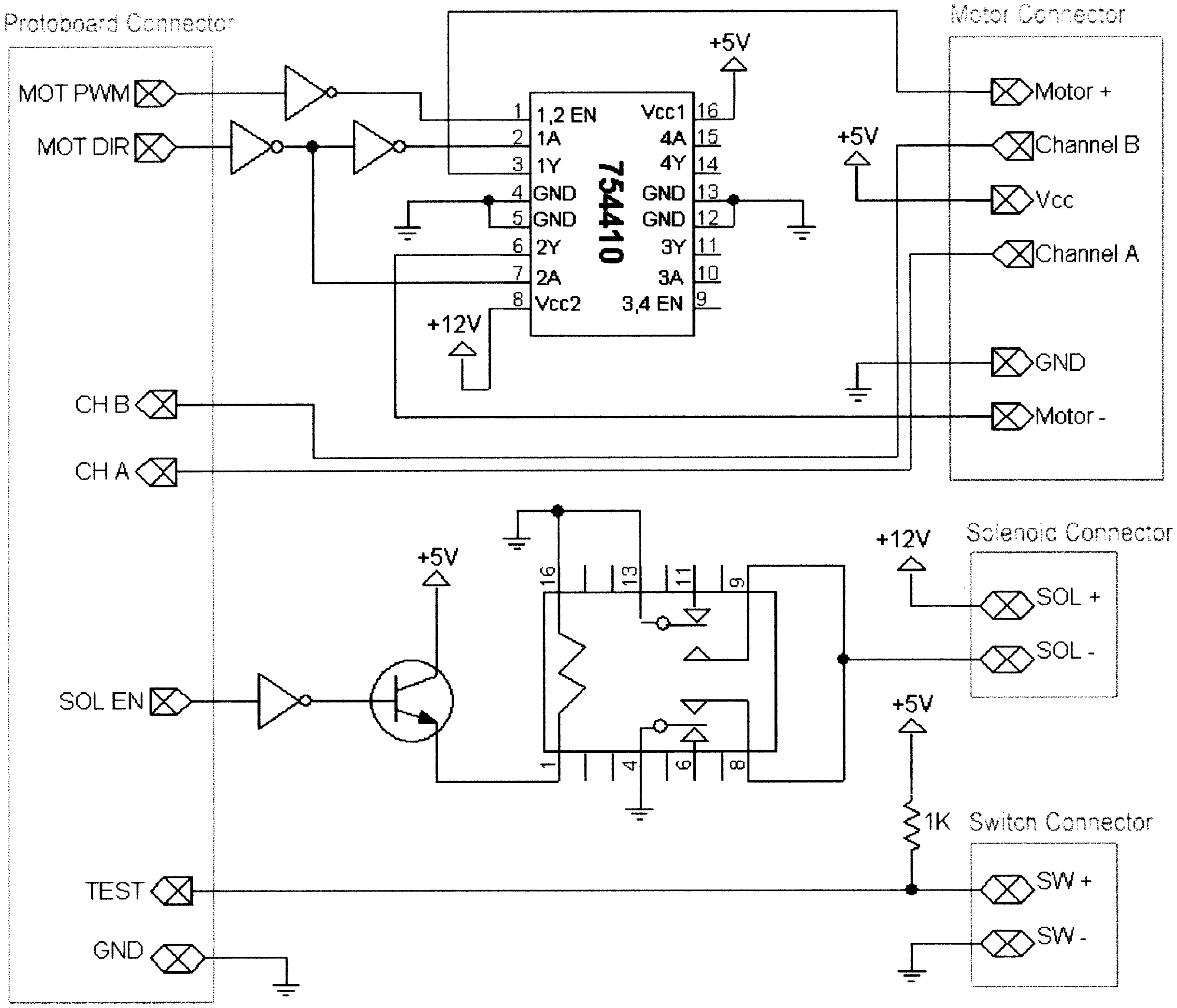


Figure 3: Driver Board Circuitry

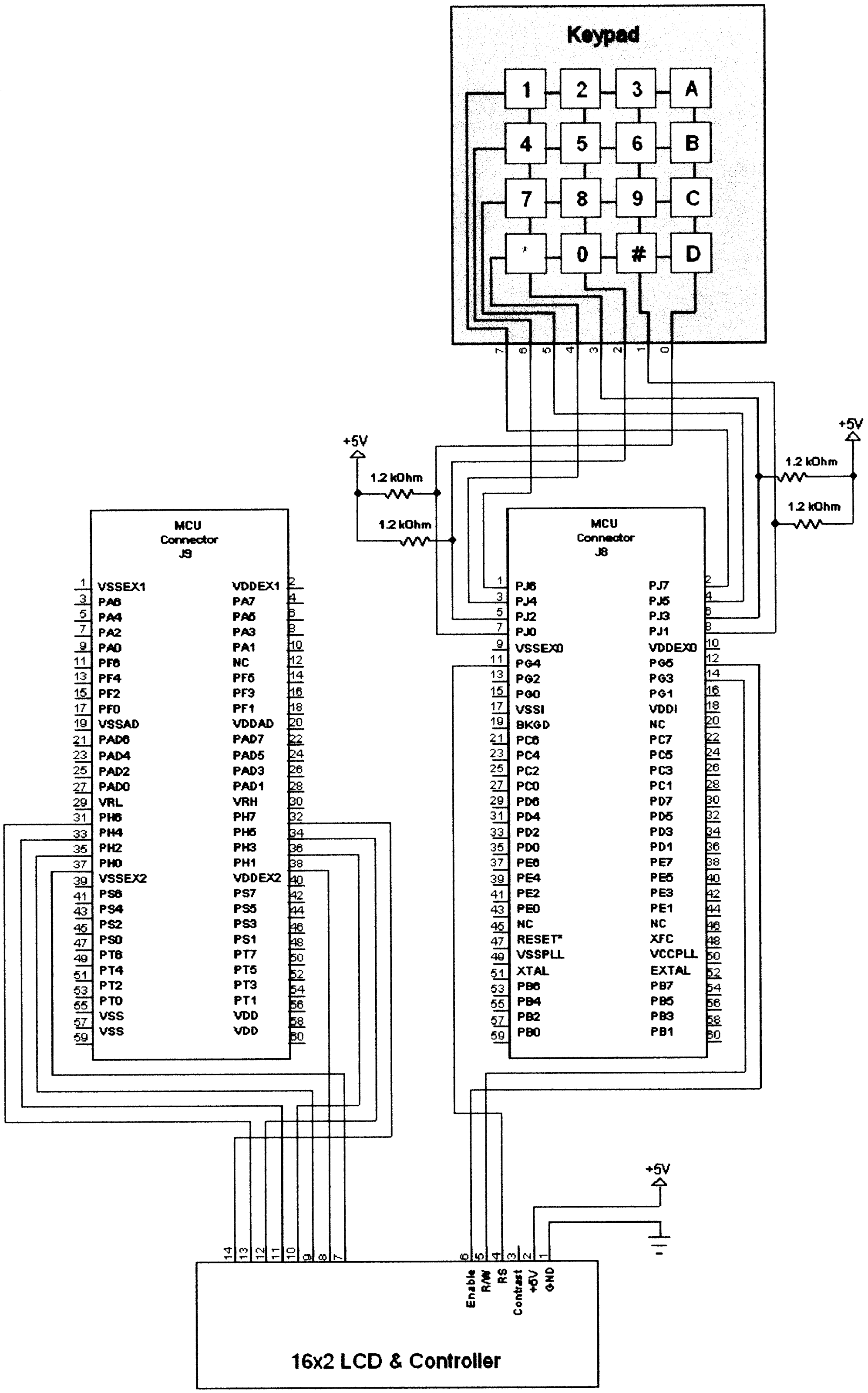


Figure 4: LCD and Keypad Circuitry

Appendix B: Images

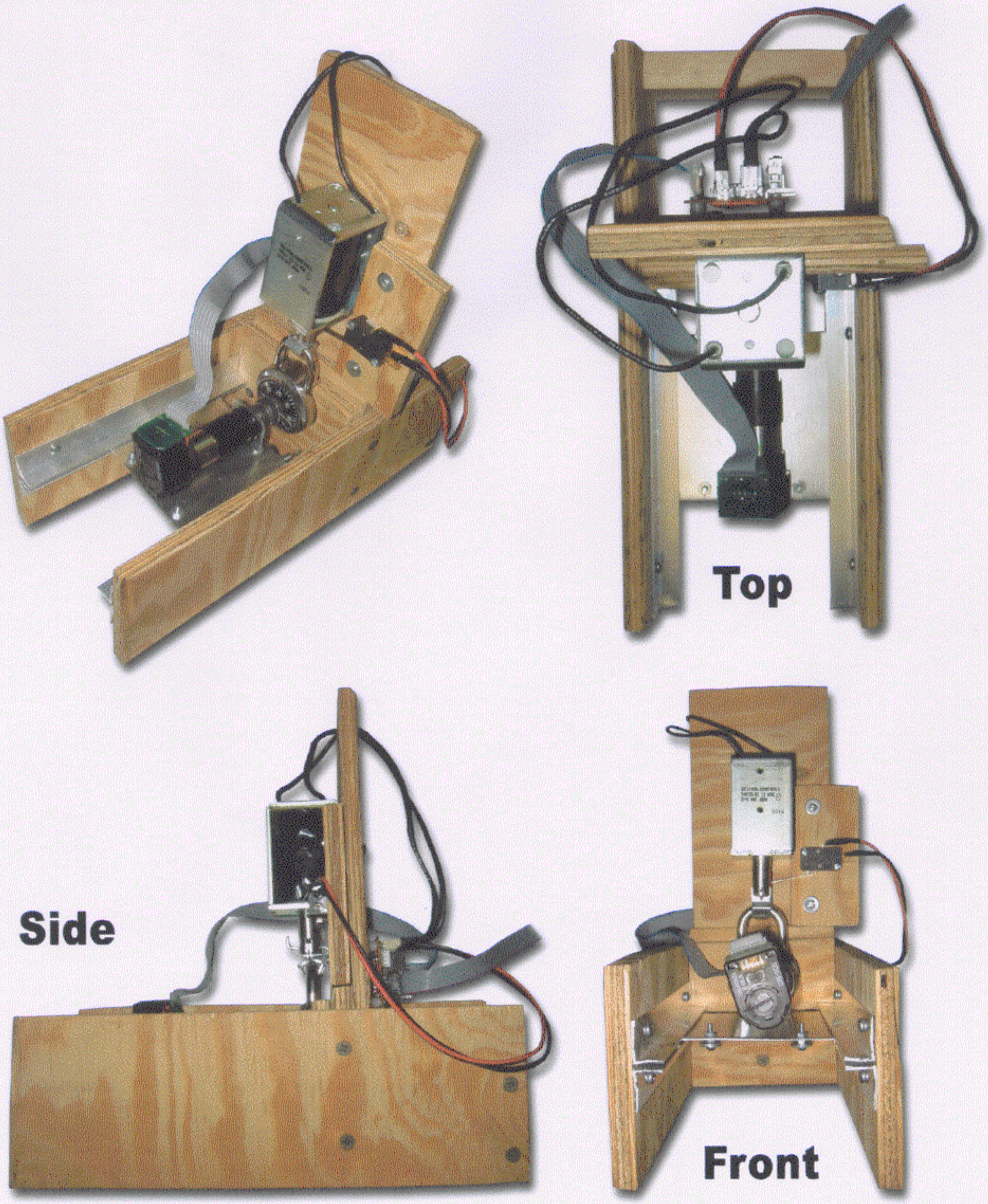


Figure 5: Multiple Views of Prototype

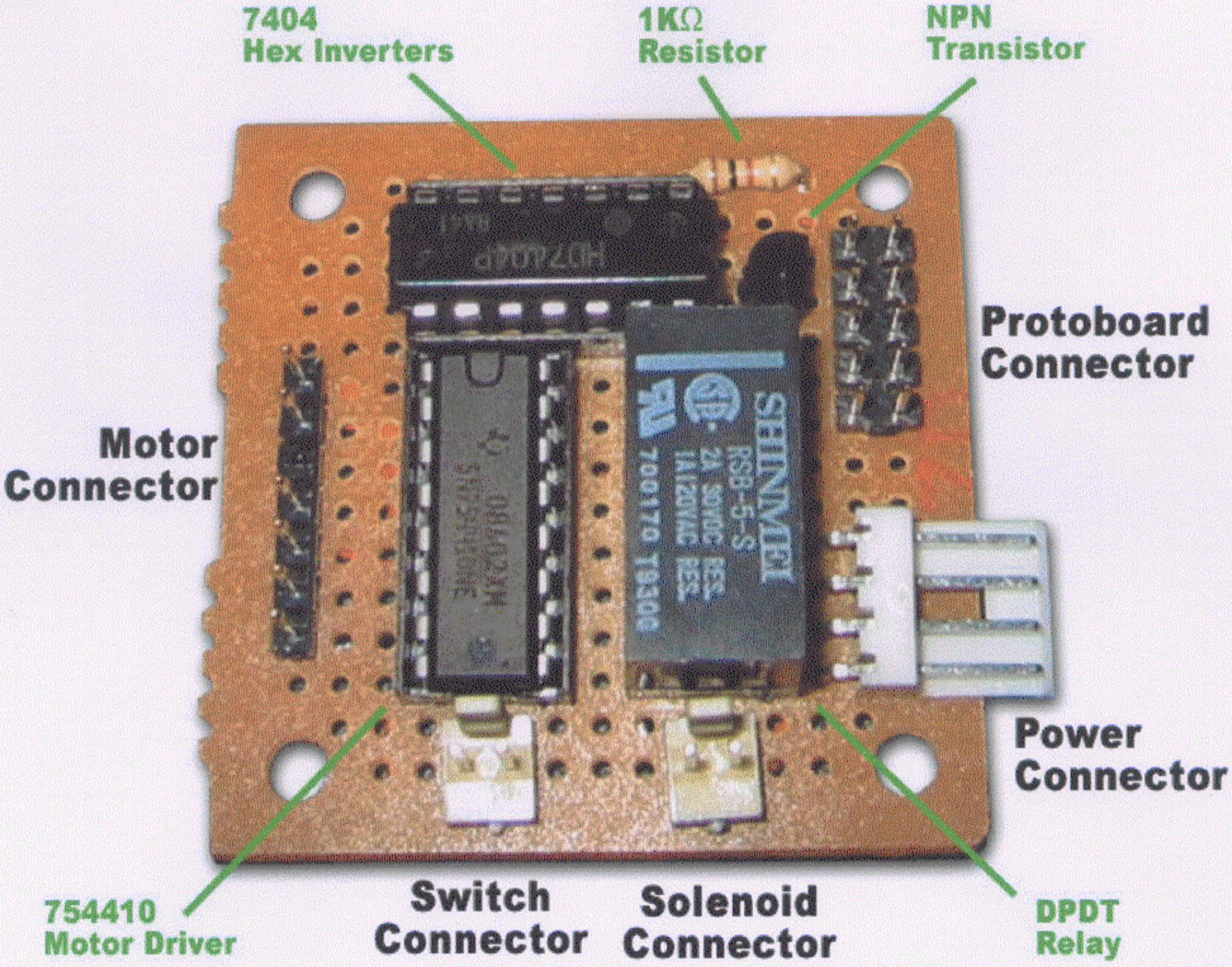
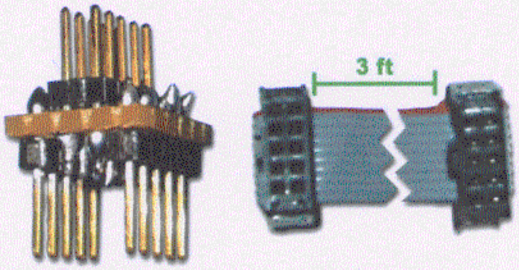


Figure 6: Driver Board Layout



Protoboard to 10-Pin
Ribbon Cable Adaptor

10-Pin
Ribbon Cable

Figure 7: Ribbon Cable and Protoboard Adaptor

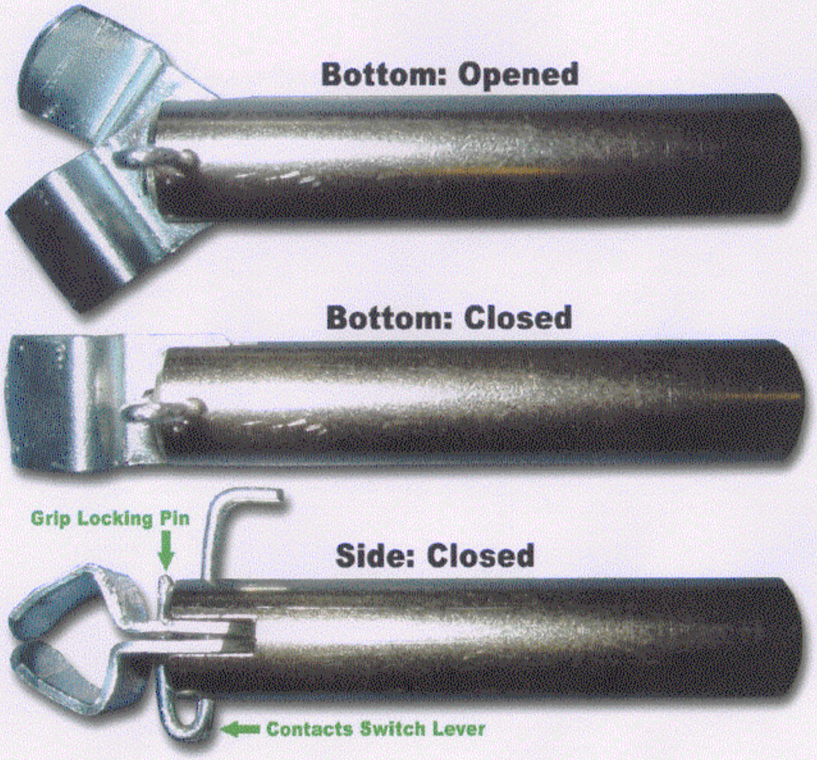


Figure 8: Multiple Views of Solenoid Plunger

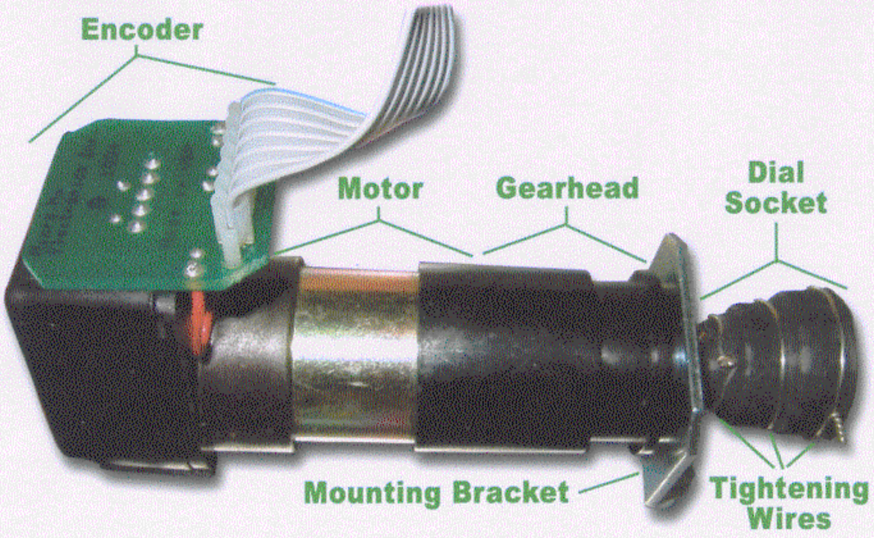


Figure 9: Motor Assembly

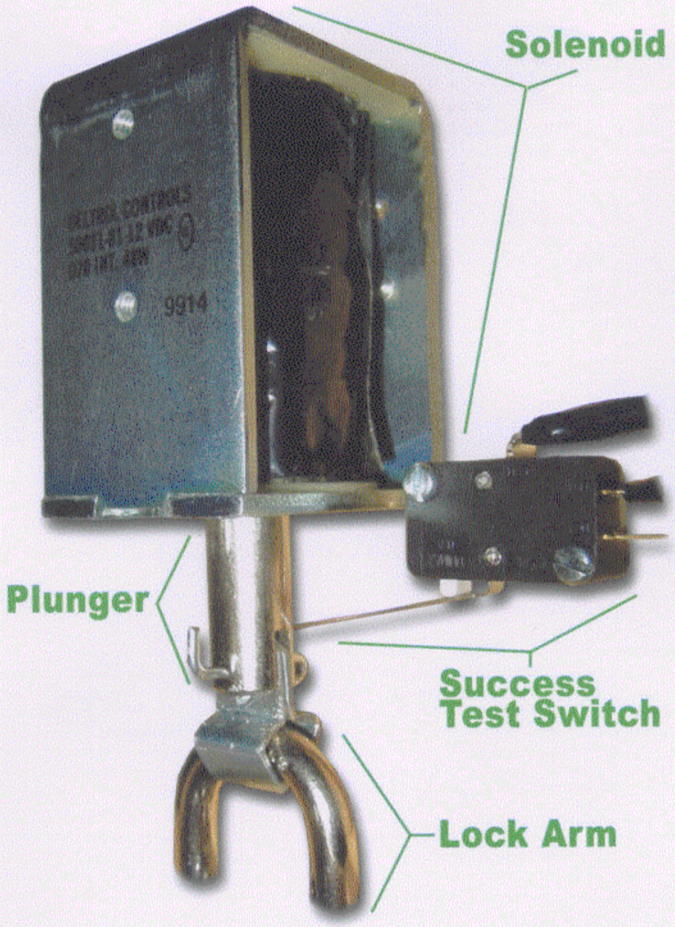


Figure 10: Solenoid Assembly

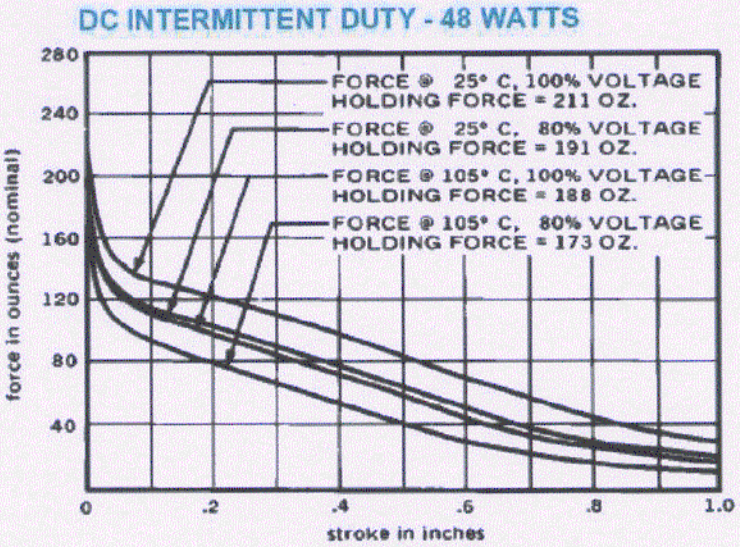
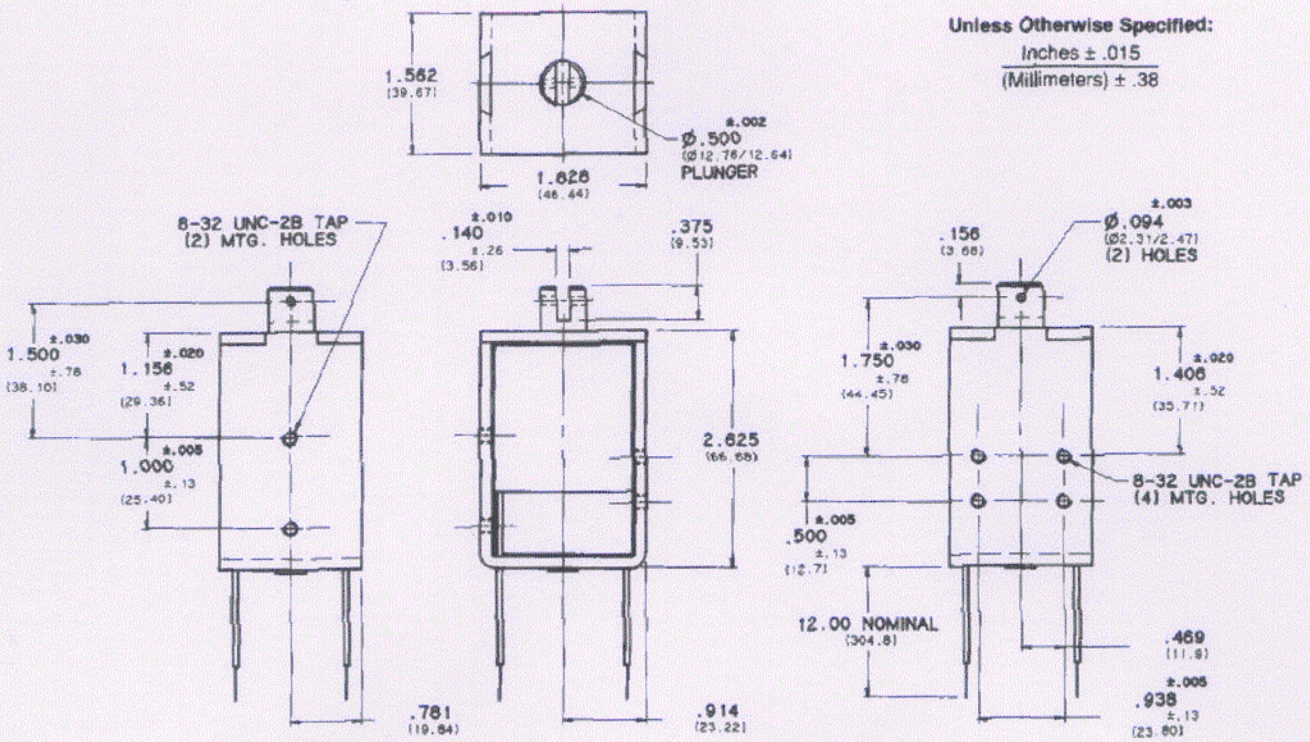


Figure 11: Solenoid Force Curves



Unless Otherwise Specified:
Inches ± .015
(Millimeters) ± .38

STANDARD POWER RATINGS:

DUTY	NOMINAL DC POWER
Continuous	13 W
Intermittent	48 W
Pulse	130 W

Figure 12: Solenoid Specifications

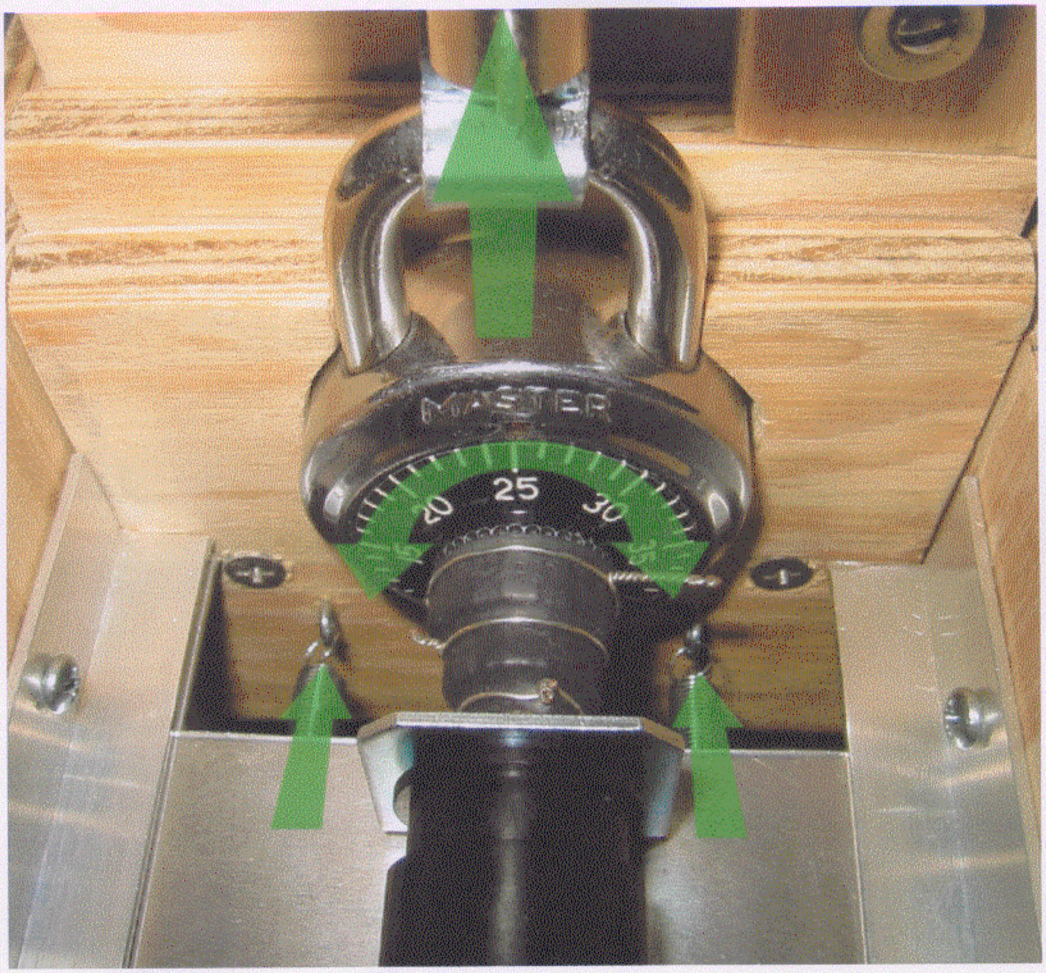
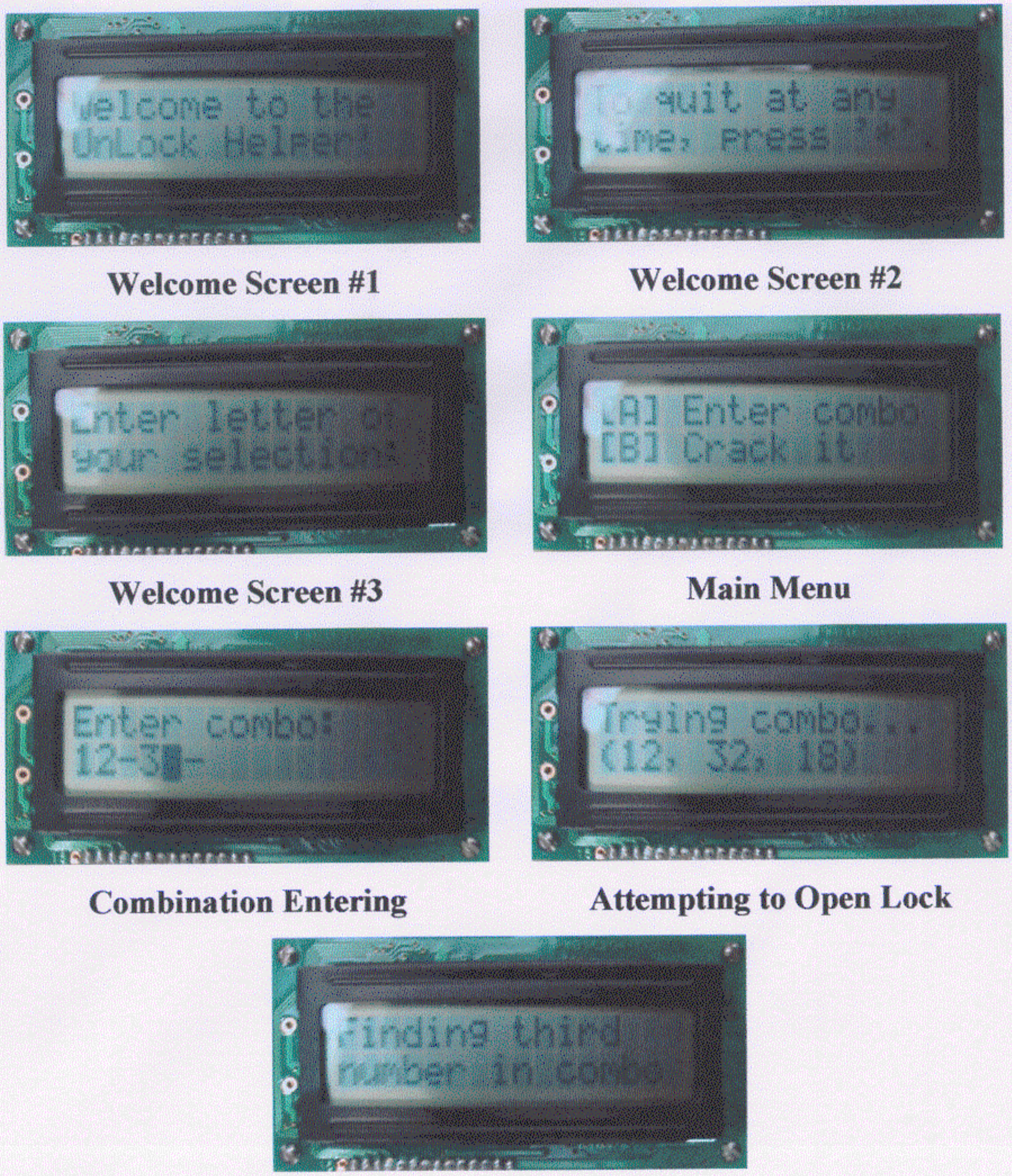


Figure 13: Forces Asserted



Welcome Screen #1

```
Welcome to the
UnLock Helper!
```

Welcome Screen #2

```
To quit at any
time, press '*'.
```

Welcome Screen #3

```
Enter letter of
your selection:
```

Main Menu

```
[A] Enter combo
[B] Crack it
```

Combination Entering

```
Enter combo:
12-3█-
```

Attempting to Open Lock

```
Trying combo...
(12, 32, 18)
```

Cracking Lock

```
Finding third
number in combo
```

Figure 14: LCD Screenshots

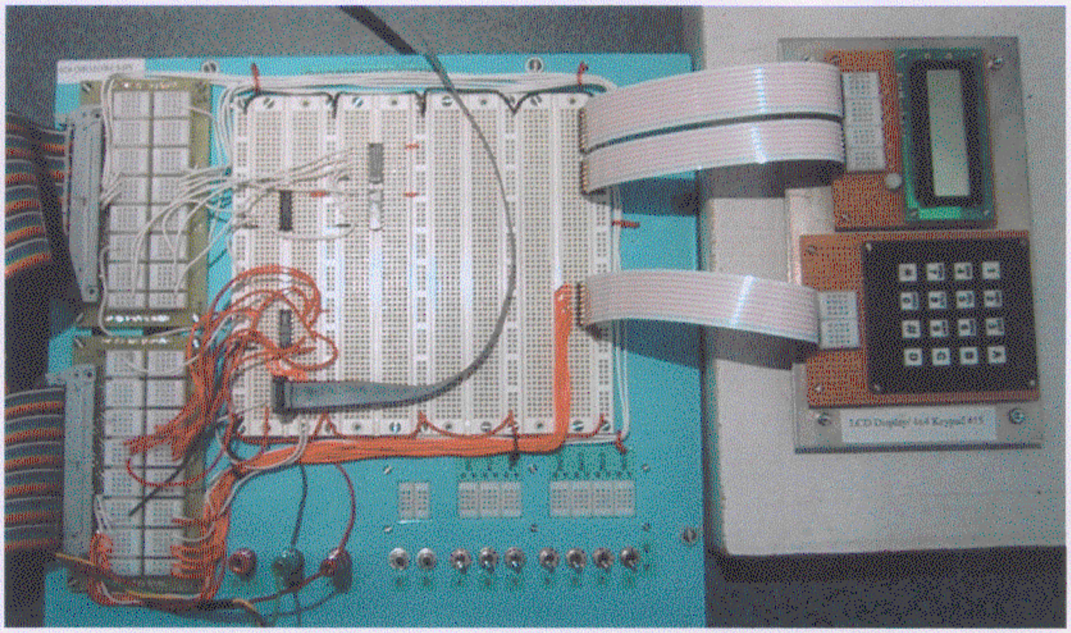


Figure 15: Protoboard, LCD, and Keypad

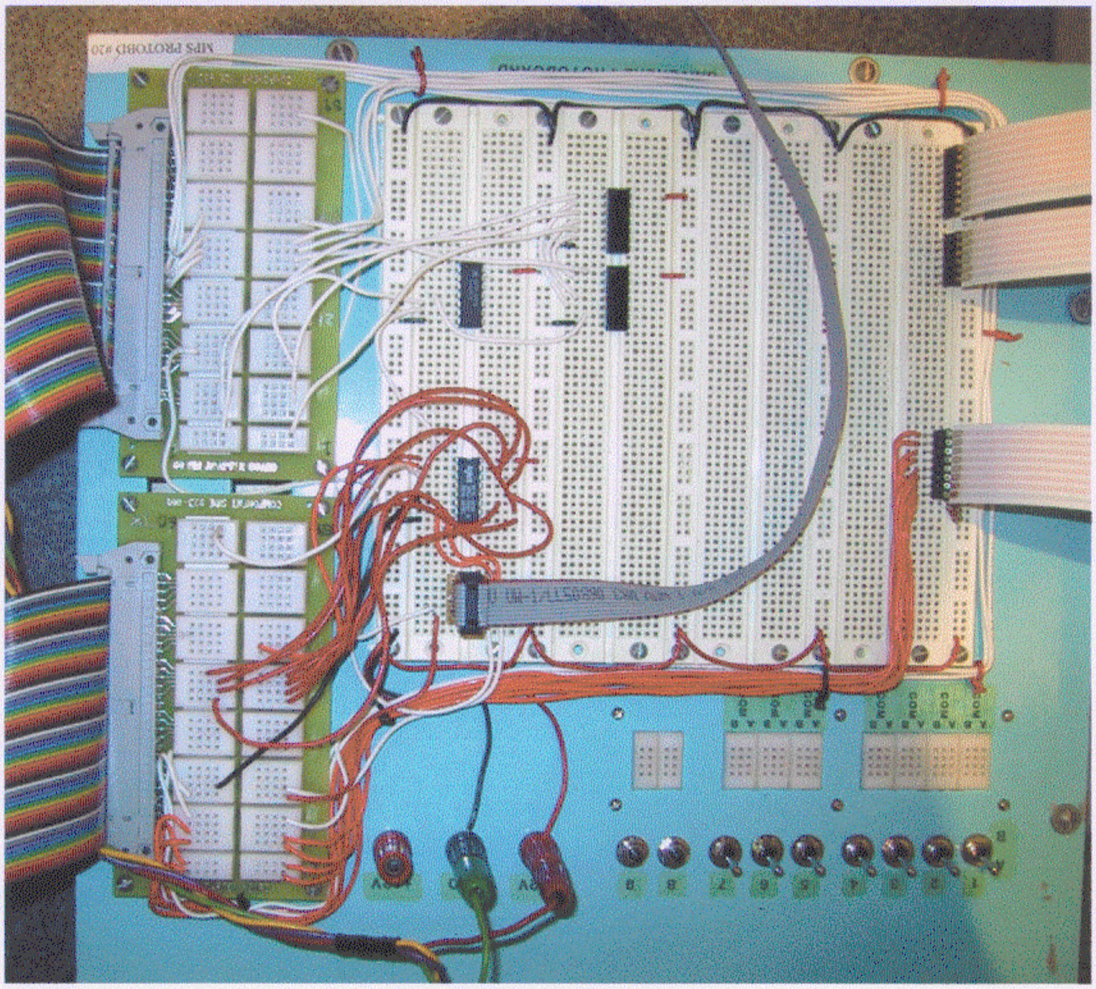


Figure 16: Protoboard

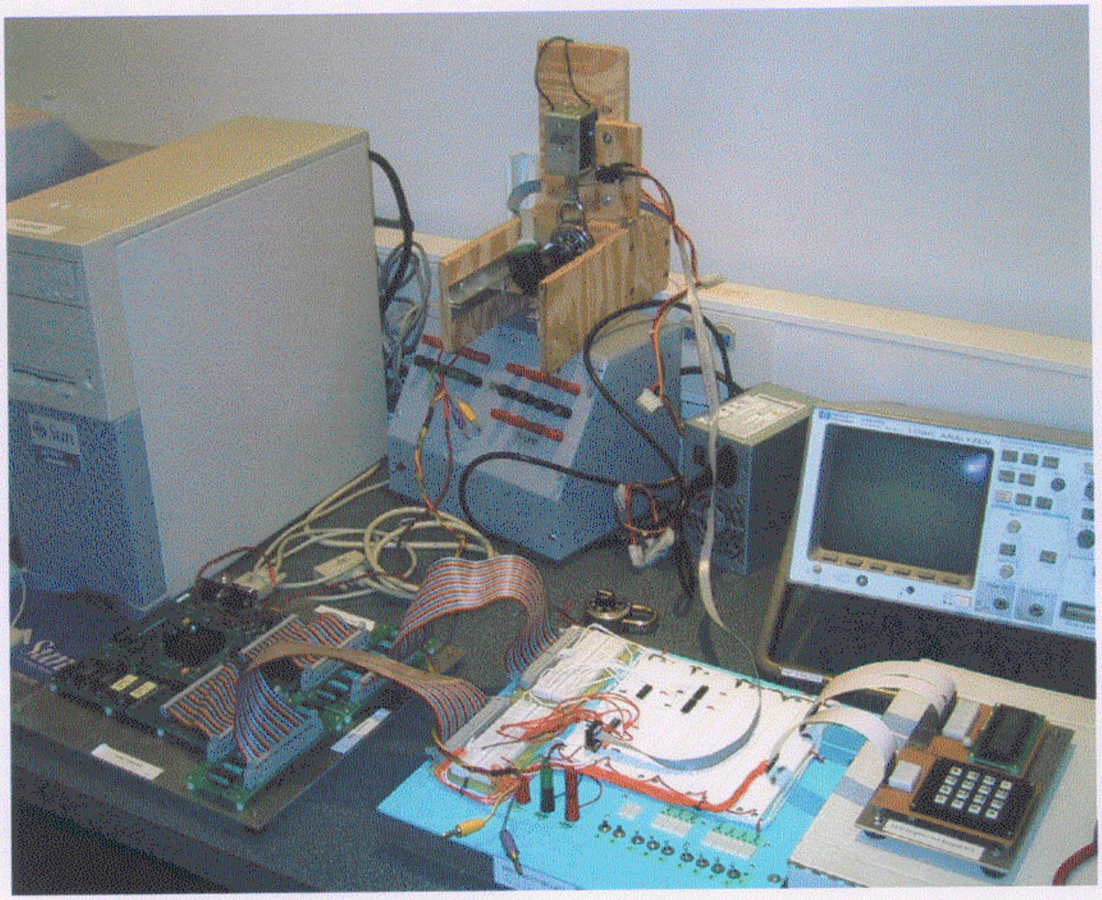


Figure 17: Overall System

Appendix C: Flowcharts and Block Diagrams

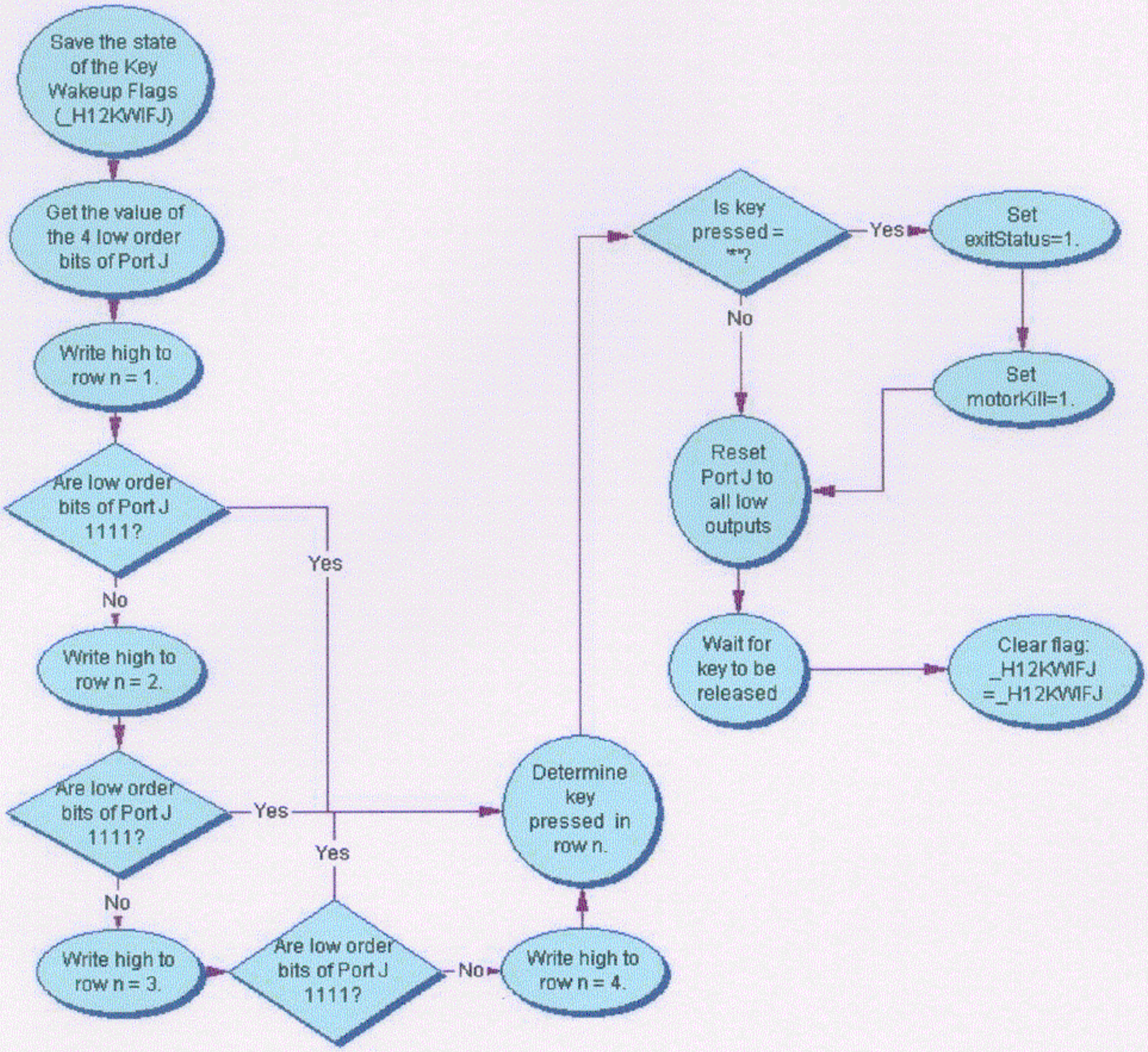

Save the state of the Key Wakeup Flags (_H12KWIFJ)
Get the value of the 4 low order bits of Port J
Write high to row n = 1.
Are low order bits of Port J 1111?
Yes
No
Write high to row n = 2.
Are low order bits of Port J 1111?
Yes
Yes
No
Write high to row n = 3.
Are low order bits of Port J 1111?
No
Write high to row n = 4.
Determine key pressed in row n.
Is key pressed = "*"?
Yes
No
Set exitStatus=1.
Set motorKill=1.
Reset Port J to all low outputs
Wait for key to be released
Clear flag: _H12KWIFJ =_H12KWIFJ

Figure 18: Keypad Flowchart

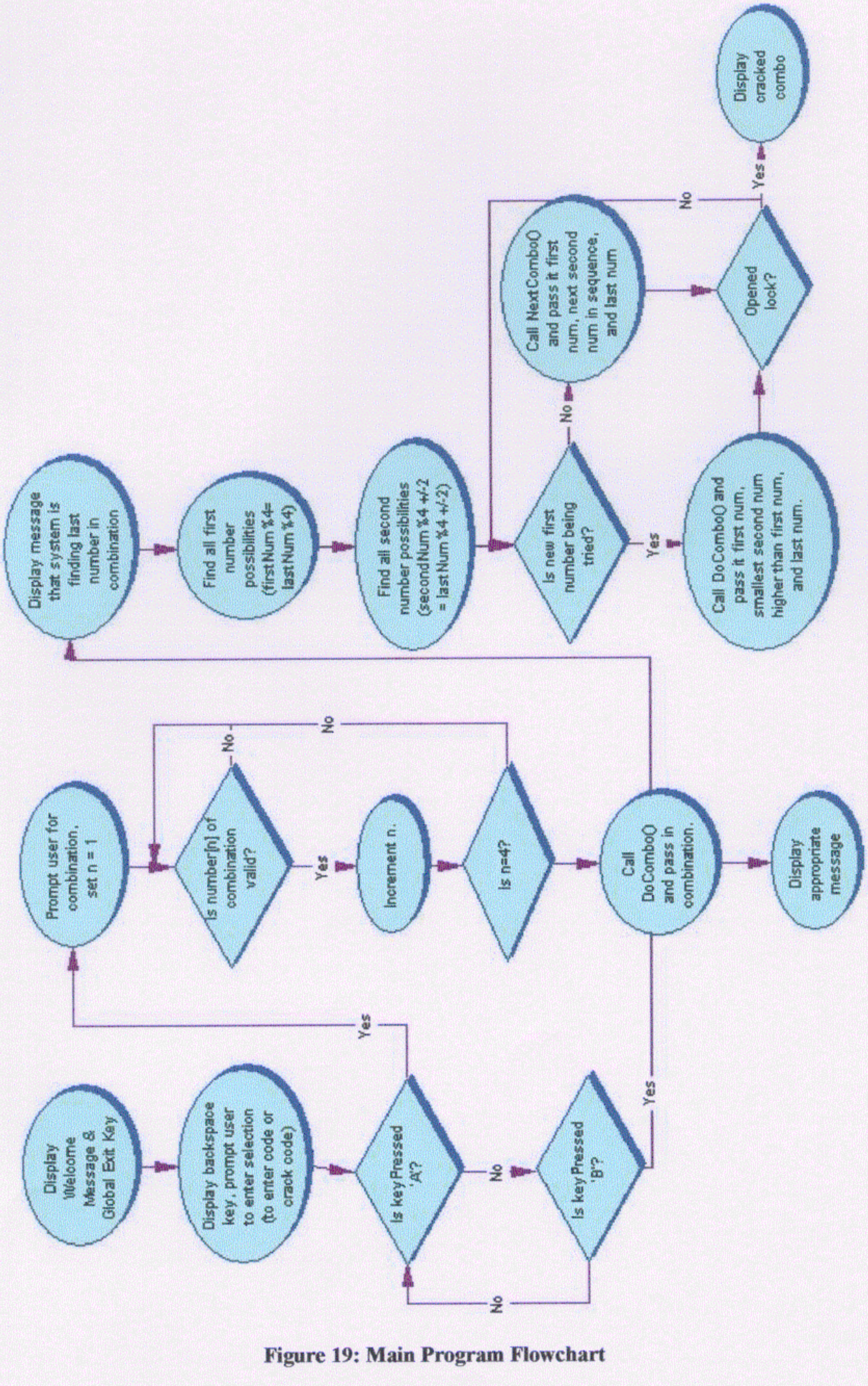
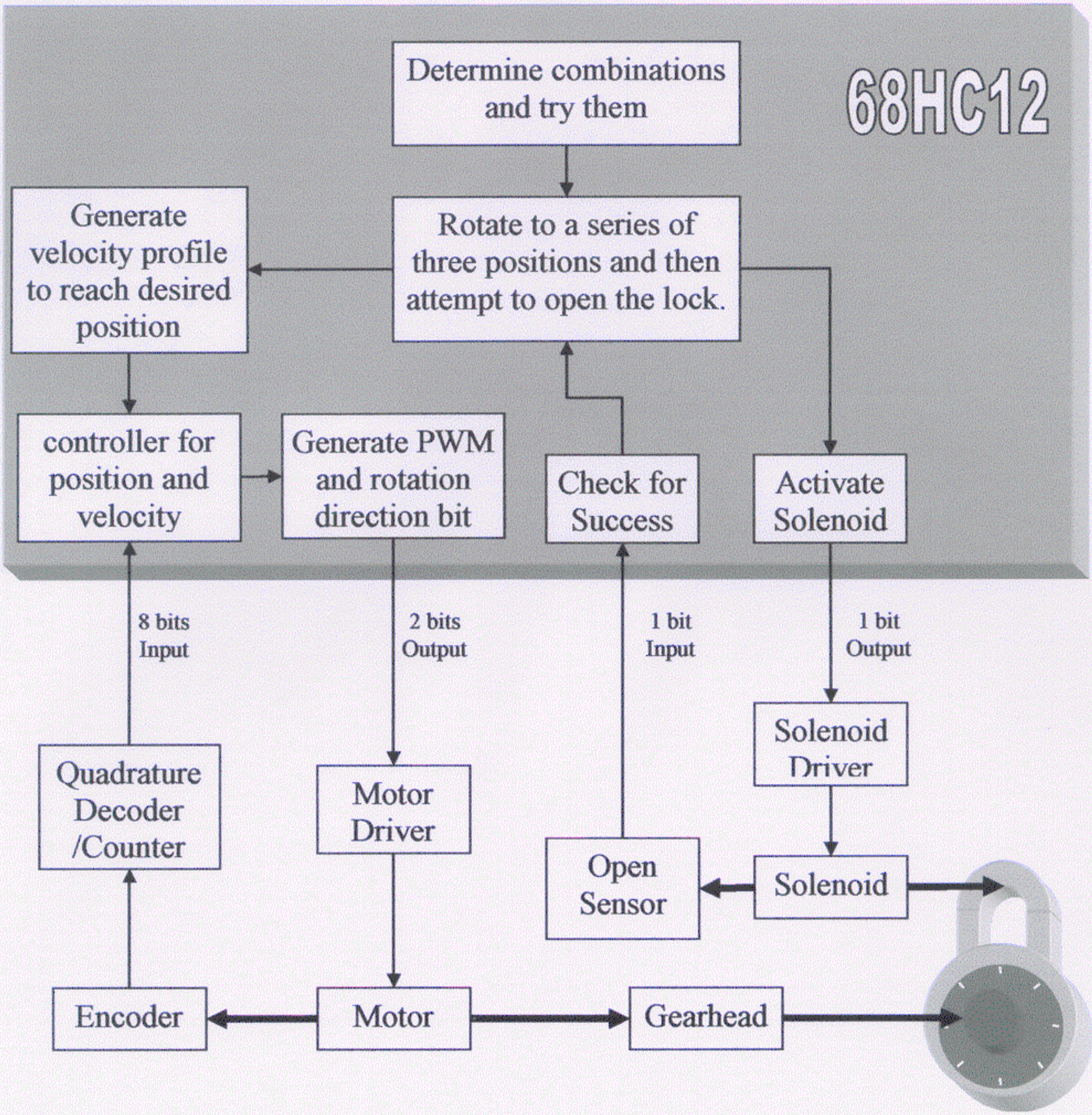

Display Welcome Message & Global Exit Key
Display backspace key, prompt user to enter selection (to enter code or crack code)
Is keyPressed 'A'?
No
Yes
Is keyPressed 'B'?
No
Yes
Prompt user for combination, set n = 1
Is number[n] of combination valid?
Yes
No
Increment n.
Is n=4?
No
Call DoCombo() and pass in combination.
Display appropriate message
Display message that system is finding last number in combination
Find all first number possibilities (firstNum %4= lastNum %4)
Find all second number possibilities (secondNum %4 +/-2 = lastNum %4 +/-2)
Is new first number being tried?
Yes
No
Call DoCombo() and pass it first num, smallest second num higher than first num, and last num.
Call NextCombo() and pass it first num, next second num in sequence, and last num
Opened lock?
Yes
No
Display cracked combo

Figure 19: Main Program Flowchart



68HC12
Determine combinations and try them
Rotate to a series of three positions and then attempt to open the lock.
Generate velocity profile to reach desired position
controller for position and velocity
Generate PWM and rotation direction bit
Check for Success
Activate Solenoid
8 bits Input
2 bits Output
1 bit Input
1 bit Output
Quadrature Decoder /Counter
Motor Driver
Solenoid Driver
Open Sensor
Solenoid
Encoder
Motor
Gearhead

Figure 20: System Block Diagram

Appendix D: Parts List

No.	Part Name	Source	Unit Price
1	68HC812A4 w/ EVB	Borrow from the MPS Lab	NA
2	PC w/ Introl Compiler	Borrow from the MPS Lab	NA
3	Protoboard	Borrow from the MPS Lab	NA
4	12VDC motor w/ attached gearhead & encoder	Donated by MicroMo Electronics, Inc.	NA
5	12VDC 48W push-type solenoid	All Electronics	$4.50
6	2 – SN754410NE motor/solenoid drivers IC	Allied Electronics	$3.70
7	HCTL-2016 quadrature decoder counter IC	Allied Electronics	$16.45
8	LCD and keypad unit	Borrow from the MPS Lab	NA
9	Master Locks	Purchased from Wal-Mart	$3.99
10	Various mechanical parts for interfacing the motor and solenoid with the padlock	Jameco Electronics, Home Depot, etc.	>$10.00

Table 3: Parts List

Appendix E: Source Code Listings

Source Code for main.c

```
// main.c
// Integrating code for the motor control, lcd, and keypad

#include <hc812a4.h>
#include <introl.h>
#include <dbug12.h>
#include <lcd12.h>
#include <keypad.h>

#define CURSOR_CHAR 0xFF
#define BLANK_CHAR 0x20
#define BACKSPACE_CHAR 'A'
#define EXITFUNC_CHAR '*'

// Function prototypes
void initialize(void);
void printWelcome(void);
void promptUser(void);
char displayOptions(void);
void getCombination(void);
void crackCombination(void);
char findPossibleCombos(void);
char tryCombination(char num1, char num2, char num3, char firstTry);

int inputNumber(char cursorLoc, char n, char hasFirstDigit);
char inputDigit(char cursorLoc, char nOfCombo, char nOfNumber);
void writeChar(char cursorLoc, char c);
void outputCombination();
void clearDisplayAndPrint(char *str);
void clearDisplayAndPrint2(char *str1, char *str2);
void delay(void);

void exitProgram(void);

// Global variables
char number[3];
char charCombo[3][2];
//bool validCombo;

// Menu options/prompting messages
char welcomeMessage[2][16] = {"Welcome to the ","UnLock Helper!"};
char promptForType[4][16] = {"Enter letter of", "your selection:",
                             "[A] Enter combo", "[B] Crack it"};
char idBackspace[2][16] = {"Press 'A' to", "backspace."};
char idAbort[2][16] = {"To quit at any", "time, press '*'."};
char promptForCombo[2][16] = {"Enter combo:", "  -  -  "};

// Status messages
char tryingCombo[1][16] = {"Trying combo..."};
char findingLastNum[2][16] = {"Finding third", "number in combo"};
char successfulOpen[1][16] = {"Opened Lock:"};
char unsuccessfulOpen[1][16] = {"Could not open:"};
char successfulCrack[1][16] = {"Cracked Combo:"};
char unsuccessfulCrack[2][16] = {"Could not crack", "lock combo."};

// Error messages
char outOfRange[2][16] = {"Number entered", "must be 0 to 39!"};
char invalidNumber[2][16] = {"Invalid number", "entered."};
char invalidSelection[2][16] = {"Sorry, that is", "not an option."};

void __main()
{
  // Initialize hardware
  initialize();
  // Print welcome message
  printWelcome();

  // Infinite loop
  while(1)
  {
    exitState = 0;
    keyPressed = 'Z';
    promptUser();
  }
}

// Initializes registers and global variables
void initialize(void)
{
  // Motor initializations
  _H12DDRG = 0xFF;
  MotorInit();

  // Initialize the LCD screen
  OpenXLCD(0x3F);

  // Set address to 0
  WriteCmdXLCD(0x80);

  // Turn on display and cursor
  WriteCmdXLCD(0x0C);

  // Initialize the key wakeup interrupt
  initKeyWakeupInterrupt();

  // Initialize the keyPressed variable (global variable in keypad.h)
  keyPressed = 'Z';

  // Initialize the exitState (global var in keypad.h) to 0 (not exiting)
  exitState = 0;
}

// Output welcome message
void printWelcome(void)
{
  // Display welcome message
  clearDisplayAndPrint2(welcomeMessage[0], welcomeMessage[1]);

  // Delay
  delay();

  // Display abort key
  clearDisplayAndPrint2(idAbort[0], idAbort[1]);

  // Delay
  delay();
}

// Function that prompts user for input
void promptUser(void)
{
  int digit = 0;
  int selection = 0;
  char openedLock = 0;

  // Get selection from user
  selection = displayOptions();

  // Check exit state (to return to the menu options again)
  if (exitState == 1)
  {
    return;
  }

  // Check if selection made was for entering a combination
  if (selection == 'A')
  {
    getCombination();
    if (exitState == 1)
    {
      return;
    }
    RESET_POS();
    openedLock = tryCombination(number[0], number[1], number[2], 1);
    if (openedLock == 0)
    {
      clearDisplayAndPrint(unsuccessfulOpen[0]);
      outputCombination();
      delay();
    }
  }
  // Check if selection made was for cracking the combination
  else if (selection == 'B')
  {
    crackCombination();
  }
  // User entered invalid selection.
  else
  {
    // Display error message
    clearDisplayAndPrint2(invalidSelection[0], invalidSelection[1]);
    delay();
  }

  // Check exit state (to return to the menu options again)
  if (exitState == 1)
  {
    return;
  }
}

// Function that displays the options for the user to select from
char displayOptions(void)
{
  char selection = '0';

  // Prompting message
  clearDisplayAndPrint2(promptForType[0], promptForType[1]);

  // Pause while message is displayed
  delay();

  // Display choices
  clearDisplayAndPrint2(promptForType[2], promptForType[3]);

  // Loop until a key is pressed on keypad
  keyPressed = 'Z';
  while (keyPressed == 'Z') {}
  selection = keyPressed;
  keyPressed = 'Z';

  return selection;
}

// Get the combination from the user
void getCombination(void)
{
  char i, j;
  int inputStatus = 0;
  char cursorLoc = 0xC0;
  char tempLoc;
  char hasFirstDigit = 0;  // whether backspace was used to get to previous number in combo

  // Prompt for combination
  clearDisplayAndPrint2(idBackspace[0], idBackspace[1]);
  delay();
```

```
clearDisplayAndPrint2(promptForCombo[0], promptForCombo[1]);
i = 0;
while (i < 3)
{
  if (inputStatus > -1)
  {
    charCombo[i][0] = ' ';
    charCombo[i][1] = ' ';
    number[i] = 0;
  }

  inputStatus = inputNumber(cursorLoc, i, hasFirstDigit);

  // Check exit state (to return to the menu options again)
  if (exitState == 1)
  {
    return;
  }

  hasFirstDigit = 0;

  if (inputStatus == -1)
  {
    if (i > 0)
    {
      --i;
      hasFirstDigit = 1;
      cursorLoc -= 0x03;
    }
  }
  else if (inputStatus == 1)
  {
    // verify that number is within the range
    while (((number[i] < 0) || (number[i] > 39)) && (inputStatus > -1))
    {
      // Print out-of-range error message
      clearDisplayAndPrint2(outOfRange[0], outOfRange[1]);
          delay();
      clearDisplayAndPrint2(promptForCombo[0], promptForCombo[1]);

      // Output the valid numbers read from user
      tempLoc = 0xC0;
      for (j = 0; j < i; j++)
      {
        SetDDRamAddr(tempLoc);
        WriteDataXLCD(charCombo[j][0]);
        WriteDataXLCD(charCombo[j][1]);
        tempLoc += 0x03;
      }

      // Input the number from the user
      charCombo[i][0] = ' ';
      charCombo[i][1] = ' ';
      number[i] = 0;
      inputStatus = inputNumber(cursorLoc, i, hasFirstDigit);

      // Check exit state (to return to the menu options again)
      if (exitState == 1)
      {
           return;
      }
    }

    if (inputStatus == 1)
    {
      ++i;
      cursorLoc += 0x03;
    }
  }
}
```

```
}

// Crack the combination of the lock
void crackCombination(void)
{
  char foundCombo = findPossibleCombos();

  // Check exit state (to return to the menu options again)
  if (exitState == 1)
  {
    return;
  }

  if (foundCombo == 0)
  {
    // failure
    clearDisplayAndPrint2(unsuccessfulCrack[0], unsuccessfulCrack[1]);
  }

  if (foundCombo == 1)
  {
    // success
    clearDisplayAndPrint(successfulCrack[0]);
    outputCombination();
  }

  delay();
}

// Find the possible combinations
char findPossibleCombos(void)
{
  char possible1stNums[10];
  char possible2ndNums[10];
  char lastNum, lastNumMod4, lastNumMod4_2;
  char doDoCombo = 1;
  char successful = 0;
  char i, j, k, num2Index;

  // initialize variables:
  // find third number
  clearDisplayAndPrint2(findingLastNum[0], findingLastNum[1]);
  lastNum = GetLastNum();

  if (lastNum == 40)
  {
    MoveToZero();
    return 0;
  }

  lastNumMod4 = lastNum%4;
  lastNumMod4_2 = (lastNumMod4 >= 2) ? lastNumMod4-2 : lastNumMod4+2;

  // Reset the program
  RESET_POS();

  // find the possible 1st and 2nd numbers in the combination
  for (i = 0; i < 10; ++i)
  {
    possible1stNums[i] = i*4 + lastNumMod4;
    possible2ndNums[i] = i*4 + lastNumMod4_2;
  }

  // try the combinations
  for (i = 0; i < 10; ++i)
  {
    doDoCombo = 1;
    num2Index = (lastNumMod4_2 > lastNumMod4) ? i : ((i == 9) ? 0 : i+1);

    for (j = 0; j < 10; ++j)
    {
      number[0] = possible1stNums[i];
      number[1] = possible2ndNums[num2Index];
      number[2] = lastNum;

      for (k = 0; k < 3; ++k)
      {
        charCombo[k][0] = ((number[k] - (number[k] % 10))/10) + '0';
        charCombo[k][1] = (number[k] % 10) + '0';
      }

      successful = tryCombination(possible1stNums[i], possible2ndNums[num2Index],
                                  lastNum, doDoCombo);

      // Check exit state (to return to the menu options again)
      if (exitState == 1)
      {
        return 0;
      }

      doDoCombo = 0;
      if (successful == 1)
      {
        return 1;
      }
      num2Index = (num2Index == 9) ? 0 : num2Index + 1;
    }
  }

  return 0;
}

// Attempts to open the Master Lock with a given combination (num1, num2, num3)
char tryCombination(char num1, char num2, char num3, char firstTry)
{
  char openedLock = 0;
  DB12->printf("%d\rCombo: %d-%d-%d\n\r", num1, num2, num3);
  // Call function to have motor try a combination
  clearDisplayAndPrint(tryingCombo[0]);
  outputCombination();

  // try combo differently if first try of first number
  if (firstTry == 1)
    openedLock = DoCombo(num1, num2, num3);
  else
    openedLock = NextCombo(num2, num3);

  // Check exit state (to return to the menu options again)
  if (exitState == 1 || openedLock == 2)
  {
    MoveToZero();
    return openedLock;
  }

  if (openedLock == 1)
  {
    clearDisplayAndPrint(successfulOpen[0]);
    outputCombination();
    delay();
  }

  return openedLock;
}

// Inputs a number from the user
// (hasFirstDigit is used to indicate if the backspace was used to return to the second
// digit of a previous number in the combination)
int inputNumber(char cursorLoc, char n, char hasFirstDigit)
{
  char digit;
  char gotFirst = hasFirstDigit;
  char gotSecond = 0;
  number[n] = (hasFirstDigit == 0) ? 0 : charCombo[n][0] - '0';

  while (gotSecond == 0)
  {
    // Input first digit of number
    if (gotFirst == 0)
    {
      digit = inputDigit(cursorLoc, n, 0);

      // Check exit state (to return to the menu options again)
      if (exitState == 1)
      {
        return 0;
      }

      if (digit == BACKSPACE_CHAR)
      {
        writeChar(cursorLoc, BLANK_CHAR);
        return -1;
      }
      number[n] = digit;
      gotFirst = 1;
    }

    // Input second digit of number
    if (gotFirst == 1)
    {
      ++cursorLoc;
      digit = inputDigit(cursorLoc, n, 1);
      // Check exit state (to return to the menu options again)
      if (exitState == 1)
      {
        return 0;
      }

      if (digit == BACKSPACE_CHAR)
      {
        writeChar(cursorLoc, BLANK_CHAR);
        --cursorLoc;
        gotFirst = 0;
        continue;
      }
      number[n] = (number[n] * 10) + digit;
      gotSecond = 1;
    }
  }

  return 1;
}

// Inputs one digit
char inputDigit(char cursorLoc, char nOfCombo, char nOfDigit)
{
  char digit;
  char validDigit = 0;

  while (validDigit == 0)
  {
    // Place the cursor in its proper place
    writeChar(cursorLoc, CURSOR_CHAR);

    // Loop until a key is pressed on keypad
    keyPressed = 'Z';
    while (keyPressed == 'Z') {}
    digit = keyPressed;
    keyPressed = 'Z';

    // Check exit state (to return to the menu options again)
    if (exitState == 1)
    {
      return 0;
    }

    if ((digit >= '0') && (digit <= '9'))
    {
      charCombo[nOfCombo][nOfDigit] = digit;
      validDigit = 1;
    }

    // Backspace
    if (digit == BACKSPACE_CHAR)
    {
      return (BACKSPACE_CHAR);
    }

    // If valid digit was entered, output on LCD screen
    if (validDigit == 1)
    {
      SetDDRamAddr(cursorLoc);
      WriteDataXLCD(digit);
    }
  }

  return (digit - '0');
}

// Writes a given character at a given location on the LCD screen
void writeChar(char cursorLoc, char c)
{
  SetDDRamAddr(cursorLoc);
  WriteDataXLCD(c);
}

// Outputs the combination in format (##, ##, ##) to LCD screen
void outputCombination()
{
  int i;
  int place = 1;
  char comboFormatted[16] = "(##, ##, ##)    ";


  for (i = 0; i < 3; i++)
  {
    comboFormatted[place] = charCombo[i][0];
    place++;
    comboFormatted[place] = charCombo[i][1];
    place += 3;
  }

  // Write combination used on second line of LCD screen
  WriteCmdXLCD(0xC0);
  WriteBuffer(&comboFormatted);
}

// Clear the display and print a string
void clearDisplayAndPrint(char *str)
{
  // Clear display
  WriteCmdXLCD(0x01);

  // Set address to 0
  WriteCmdXLCD(0x80);

  // Write string to screen
  WriteBuffer(str);
}

// Clear the display and print strings on 2 lines
void clearDisplayAndPrint2(char *str1, char *str2)
{
  // Clear display
  WriteCmdXLCD(0x01);

  // Set address to 0
  WriteCmdXLCD(0x80);

  // Write first string to screen
  WriteBuffer(str1);

  // Move cursor to next line
  WriteCmdXLCD(0xC0);

  // Write second string to screen
  WriteBuffer(str2);
}

// Generates a delay
void delay(void)
{
  int j, k;

  for (j = 0; j < 10; j++)
    for (k = 0; k < 60000; k++);
}

// Exit program
void exitProgram(void)
{
  // Clear display
  WriteCmdXLCD(0x01);

  DB12->main();
}
```

Source Code for motor.h

```
// motor.h
// Header file for motor control stuff
// 12/1/02

#ifndef __MOTOR_H
#define __MOTOR_H

#include <hc812a4.h>
#include <introl.h>

#define MOTOR_CW 1
#define MOTOR_CCW -1
#define MOTOR_OFF 0

#define MOTOR_PWM _H12TC0
#define ONE_ROTATION 5181 // number of pulses in one rotation

/******* macros *******/
#define MOTOR_DIR_CW()  { _H12PORTG = (_H12PORTG | 0x01);}
#define MOTOR_DIR_CCW() { _H12PORTG = (_H12PORTG & ~0x01);}
#define SOLENOID_UP() {       _H12PORTG = (_H12PORTG | 0x04); }
#define SOLENOID_DOWN() { _H12PORTG = (_H12PORTG & ~0x04); }
#define GET_MOTOR_DIR() ((_H12PORTG & 0x01)?1:-1)
#define RESET_PULSE_CNT() { _H12PORTG = (_H12PORTG & ~0x02); \
                            _H12PORTG = (_H12PORTG | 0x02); }
#define RESET_POS() { motorPos = -ONE_ROTATION;lowNibble=0;}


#define ABS(val) ((val > 0)?val:-val)

/******* global variables  *******/
signed int motorSpeed, motorPos;
signed int desiredSpeed;
signed int target;
signed char motorActive, motorFeel, motorKill;
unsigned char lowNibble;

// addresses for reading HCTL-2016 chip
const char *pulsesHigh = 0x2001;
const char *pulsesLow  = 0x2003;

/******* prototypes *******/
void MotorMove(signed char dir, signed int goal);
void SetMotorPWM(signed int pwmVal);
char DoCombo(char n1, char n2, char n3);
char NextCombo(char n2, char n3);
char OpenLock();
char GetLastNum();
void FeelNotch(signed int *sep, signed char *num);
void MoveToZero();
__mod2__ void RTIInt();

// initialize motor control
void MotorInit()
{
  // initialize variables
  motorSpeed = 0;
  motorPos = 0;

  desiredSpeed = 0;
  target = 0;
  motorActive = 0;
  motorFeel = 0;
  motorKill = 0;
  lowNibble = 0;

  _H12TIOS = 0x81;   // set bit 0 for OC
  _H12TC0  = 1;      // OC0 at 30000
  _H12TCTL1= 0x00;    // do nothing with these bits
  _H12TCTL2= 0x03;    // set OC0 to high on compare
  _H12TC7  = 0x0000;  // OC7 at 0x0000
  _H12OC7M = 0x01;    // OC7 effects OC1
  _H12OC7D = 0x00;    // reset OC0 on OC7 compare
  _H12TSCR = 0x80;    // Enable Timer

  _H12CSCTL0 = 0xF0; // upper half of Port F for chip select
  _H12DDRF = 0xF0;    // set lower have of Port F for input

  // initially deactivate solenoid
  _H12PORTG = (_H12PORTG & ~0x04);

  MOTOR_DIR_CW();      // set initial motor direction
  RESET_PULSE_CNT(); // clear inital encoder count
  DB12->SetUserVector(RTI,RTIInt);
  _H12RTICTL = 0x84; //activate RTI every ~8.192 msec
}

// move the dial to the goal position
void MotorMove(signed char dir, signed int goal)
{
  // set the direction
  if(dir == MOTOR_CW){
    MOTOR_DIR_CW();
  }else if(dir == MOTOR_CCW){
    MOTOR_DIR_CCW();
  }
  // initialize the target value
  target = goal;
  // enable the motion
  motorActive = dir;
}

// set motor PWM based on a signed value
void SetMotorPWM(signed int pwmVal)
{
  // set direction
  if(pwmVal < 0){
    MOTOR_DIR_CCW()
    pwmVal = ABS(pwmVal);
  }else{
    MOTOR_DIR_CW()
  }

  // set magnitude
  if(pwmVal == 0)
    MOTOR_PWM = 1;
  else
    MOTOR_PWM = pwmVal;
}

// do a complete combination
// return open lock status or 2 for cancelled
char DoCombo(char n1, char n2, char n3)
{
  int c;
  signed int pos1, pos2, pos3;
  // calculate positions
  pos1 = -130*n1;
  pos2 = -ONE_ROTATION-130*n2;
  pos3 = -130*n3;
  if(n2 < n1){
    pos2 -= ONE_ROTATION;
    pos3 -= ONE_ROTATION;
  }
  if(n3 < n2) pos3 -= ONE_ROTATION;

  //back up 3 rotations for clearing
  motorPos -= 3*ONE_ROTATION;
  // move to first number
  MotorMove(MOTOR_CW, pos1);
  while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 2;
    }
  }
  // move to second number
  MotorMove(MOTOR_CCW, pos2);
  while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 2;
    }
  }
  // move to third number
  MotorMove(MOTOR_CW, pos3);
  while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 2;
    }
  }

  return OpenLock();
}

// move to the next combo in the sequence
// without clearing
// return open lock status or 2 for cancelled
char NextCombo(char n2, char n3)
{
  int c;
  signed int pos2, pos3;
  // calculate positions
  pos2 = -ONE_ROTATION-130*n2;
  pos3 = -130*n3;
  if(n3 < n2) pos3 -= ONE_ROTATION;

  // move to second number
  MotorMove(MOTOR_CCW, pos2);
  while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 2;
    }
  }
  // move to first number
  MotorMove(MOTOR_CW, pos3);
  while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 2;
    }
  }

  return OpenLock();
}

// try to open the lock
// return 1 for success, 0 for failure
char OpenLock()
{
  unsigned int i;
  char opened = 0;
  SOLENOID_UP(); // pull on latch
  for(i=0; i<65000; i++){
    // check for success signal
    if(!(_H12PORTF & 0x08)){
      opened=1;
    }
  }
  SOLENOID_DOWN();// release latch
  return opened;
}

// feel notches to find the last number
// return 40 if cancelled
char GetLastNum()
{
  unsigned int i;
  char tempNum, lastNum = -1;
  signed int currSep, maxSep=0;
  motorPos = ONE_ROTATION;

  // for each of the 12 notches
  for(i=0;i<12;i++){
    FeelNotch(&currSep, &tempNum);
    // store maximum seperation and corresponding number
    if(currSep > maxSep){
      maxSep = currSep;
      lastNum = tempNum;
    }
    DB12->printf("\r%d\rNotch #%d\tNum:%d\tSep:%d \n\r",
                 i,tempNum,currSep);
    // move to the next approx. location
    MotorMove(MOTOR_CCW, motorPos-390);
    while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 40;
    }
  }

  DB12->printf("\r%d\rLast Num: %d \n\r",lastNum);

  //move back to zero
  MotorMove(MOTOR_CCW, 0);
  while(motorActive != MOTOR_OFF){
    if(motorKill){
      motorActive = MOTOR_OFF;
      return 40;
    }
  }

  return lastNum;
}

// take measurements of the nearest notch
void FeelNotch(signed int *sep, signed char *num)
{
  signed int notchEnd[4];
  float val1, val2, separation;

  SOLENOID_UP(); // pull up on the latch
  motorFeel = 1; // set motorFeel flag

  // move back and forth one to drop into a notch
  MOTOR_DIR_CCW();
  motorActive = MOTOR_CCW;
  while(motorActive != MOTOR_OFF);

  MOTOR_DIR_CW();
  motorActive = MOTOR_CW;
  while(motorActive != MOTOR_OFF);


  // move back and forth twice while taking measurements
  MOTOR_DIR_CCW();
  motorActive = MOTOR_CCW;
  while(motorActive != MOTOR_OFF);
  notchEnd[0] = motorPos;

  MOTOR_DIR_CW();
  motorActive = MOTOR_CW;
  while(motorActive != MOTOR_OFF);
  notchEnd[1] = motorPos;

  MOTOR_DIR_CCW();
  motorActive = MOTOR_CCW;
  while(motorActive != MOTOR_OFF);
  notchEnd[2] = motorPos;

  MOTOR_DIR_CW();
  motorActive = MOTOR_CW;
  while(motorActive != MOTOR_OFF);
  notchEnd[3] = motorPos;

  motorFeel = 0;
  SOLENOID_DOWN(); // release the latch

  // calculate the average seperation
  *sep = ((notchEnd[1]+notchEnd[3])-(notchEnd[0]+notchEnd[2]));
  // calculate the nearest number on the dial
  *num = (char)((notchEnd[0]+notchEnd[1]+notchEnd[2]+notchEnd[3]+260)/520);
  *num = 40 - *num;
  // correct for negative numbers
  if(*num < 0) *num += 40;
}

// reposition the lock at 0
// used after a cancellation
void MoveToZero()
{
  if(motorPos > -ONE_ROTATION) //choose shortest direction
    MotorMove(MOTOR_CCW, -ONE_ROTATION);
  else
    MotorMove(MOTOR_CW, -ONE_ROTATION);
  while(motorActive != MOTOR_OFF);
  motorKill = 0;       // reset motorKill flag
}

// real time interrupt service routine
__mod2__ void RTIInt()
{
  signed int temp1, temp2;
  signed int ePos, eSpeed;
  unsigned int newSpeed;
  // read the encoder count
  unsigned char valH = *pulsesHigh;
  unsigned char valL = *pulsesLow;
  RESET_PULSE_CNT();

  // determine the speed in counts per RTI
  motorSpeed = (((unsigned int)valH)<<8) + valL;
  // update with previous least significant nibble
  temp1 = motorSpeed + lowNibble;
  // store least significant nibble
  lowNibble = (unsigned char)(0x000F & temp1);
  // divide by 16
  temp2 = temp1>>4;
  // correct for sign changes
  if (temp1 < 0)
    temp2 |= 0xF000;
  // integrate to find position
  // (sumation of scaled velocity)
  motorPos += temp2;

  // find the error in speed
  eSpeed = 2*(desiredSpeed - ABS(motorSpeed));
  // find the error in position
   ePos = target - motorPos;
```

Source Code for motor.h

```
  if(motorActive != MOTOR_OFF){
    // if feeling notches
    if(motorFeel){
      desiredSpeed = 200;
      // check for cutoff force
      if(MOTOR_PWM > 20000){
        motorActive = MOTOR_OFF;
        desiredSpeed = 0;
      }
    }
    // if doing combinations
    else{
      // set the direction of the error
      // relative to the motor direction
      ePos *= motorActive;
      // select appropriate velocity based on error
      if(ePos > 0){
        if(ePos > 4545){
          desiredSpeed = 1600;
        }else if(ePos > 390){
          desiredSpeed = 95+ePos/3;
        }else{
          desiredSpeed = 30+ePos/2;
        }
      }else{ // the goal was reached
        motorActive = MOTOR_OFF;
        desiredSpeed = 0;
      }
    }
    // update the PWM to correct for speed error
    SetMotorPWM(motorActive*(MOTOR_PWM + eSpeed));
  }
  // if waiting for a command
  else{
    // if inbetween feeling notch commands
    if(motorFeel){
      SetMotorPWM(0); // stop the motor
    }else{
      // set the direction to oppose error
      if(ePos > 0){
        MOTOR_DIR_CW();
      }else{
        MOTOR_DIR_CCW();
      }
      desiredSpeed = 0; // desire no motor motion
      // update the PWM to oppose error
      SetMotorPWM(GET_MOTOR_DIR()*(MOTOR_PWM + eSpeed));
    }
  }

  // reset the flag
  _H12RTIFLG=0x80;
}

#endif
```

Source Code for keypad.h

```
#include <hc812a4.h>
#include <dbug12.h>
#include "motor.h"

void initKeyWakeupInterrupt(void);
__mod2__ void keyWakeupISR(void);
void determineKey(char r, unsigned int lowBits);

// Global Variables
char exitState;
char keyPressed;

void initKeyWakeupInterrupt(void)
{
  // setup the interrupt service routine for a key on port J
  DB12->SetUserVector(PortJKey, keyWakeupISR);

  _H12KPOLJ=0x00;      // falling edge sets flag
  _H12KWIFJ=0xFF;      // clear any flags that may be set
  _H12PUPSJ=0xFF;      // pull up
  _H12PULEJ=0xFF;      // pull up enabled all bits
  _H12KWIEJ=0x0F;      // enable bits 0-3 (column selects) for generating interrupts
  _H12DDRJ=0xF0;       // Set up row select bits (4-7) for output
  _H12PORTJ=0x00;      // Write lows 0's to row select bits
}

__mod2__ void keyWakeupISR(void)
{
  unsigned int keyWakeupFlags, lowOrderBits;

  // Save the state of the Key Wakeup Flags
  keyWakeupFlags = _H12KWIFJ;

  // Get the value of the 4 low order bits on Port J
  lowOrderBits = keyWakeupFlags & 0x0F;

  // Parse through the different rows to determine which key was pressed.
  // Write a high to each row individually and then look at the value on
  // the low nibble of Port J.  If the row a 1 is written to is the row
  // in which the button was pressed, then the value on the low bits
  // of Port J will be (----1111).

  // Write high to row 1
  _H12PORTJ=0x10;
  if ((_H12PORTJ & 0x0F) == 0x0F)
  {
    determineKey(1,lowOrderBits);
  }
  else
  {
    // Write high to row 2
    _H12PORTJ = 0x20;
    if ((_H12PORTJ & 0x0F) == 0x0F)
    {
      determineKey(2,lowOrderBits);
    }
    else
    {
      // Write high to row 3
      _H12PORTJ = 0x40;
      if ((_H12PORTJ & 0x0F) == 0x0F)
      {
        determineKey(3,lowOrderBits);
      }
      else
      {
        // Write high to row 4
        _H12PORTJ = 0x80;
        if ((_H12PORTJ & 0x0F) == 0x0F)
        {
          determineKey(4,lowOrderBits);
        }
      }
    }
  }

  _H12PORTJ=0x00;         // reset port J to all low outputs

  // wait for the key to be released
  while ((_H12PORTJ & 0x0F) != 0xF) {}

  _H12KWIFJ=_H12KWIFJ;    // Clear the flag
}

// Determine which column corresponds to the low bit settings
// and determine the corresponding character to the (row, column)
void determineKey(char r, unsigned int lowBits)
{
  // variable declarations
  int row, column;
  row = r;
  column = 0;

  // determine which column the key was pressed in
  if (lowBits == 0x1)
  {
    column = 1;
  }
  else if (lowBits == 0x2)
  {
    column = 2;
  }
  else if (lowBits == 0x4)
  {
    column = 3;
  }
  else if (lowBits == 0x8)
  {
    column = 4;
  }

  // output an error, if the column cannot be determined
  if (column == 0)
  {
    DB12->printf("Error!!!  Column not determined correctly.\n\r");
    DB12->printf("%x\rLow bits: %x\n\n\r",lowBits, lowBits);
  }
  // if the button was pressed in row 1, determine from the column, which key was pressed
  else if (row == 1)
  {
    if (column == 1)
    {
      keyPressed = 'D';
    }
    else if (column == 2)
    {
      keyPressed = '#';
    }
    else if (column == 3)
    {
      keyPressed = '0';
    }
    else if (column == 4)
    {
      keyPressed = '*';
      exitState = 1;
      motorKill = 1;
    }
  }
```

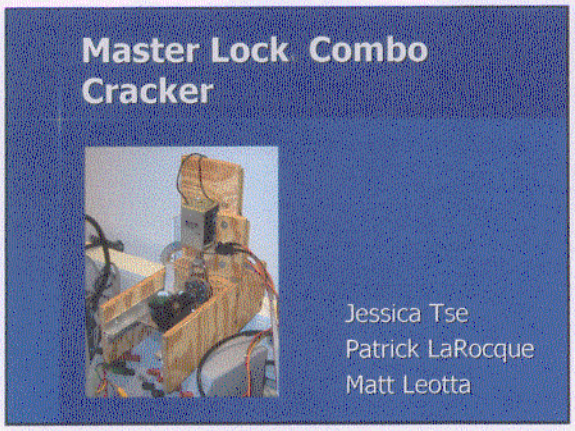
Source Code for keypad.h

```
  // if the button was pressed in row 2, determine from the column, which key was pressed
  else if (row == 2)
  {
    if (column == 1)
    {
      keyPressed = 'C';
    }
    else if (column == 2)
    {
      keyPressed = '9';
    }
    else if (column == 3)
    {
      keyPressed = '8';
    }
    else if (column == 4)
    {
      keyPressed = '7';
    }
  }
  // if the button was pressed in row 3, determine from the column, which key was pressed
  else if (row == 3)
  {
    if (column == 1)
    {
      keyPressed = 'B';
    }
    else if (column == 2)
    {
      keyPressed = '6';
    }
    else if (column == 3)
    {
      keyPressed = '5';
    }
    else if (column == 4)
    {
      keyPressed = '4';
    }
  }
  // if the button was pressed in row 4, determine from the column, which key was pressed
  else if (row == 4)
  {
    if (column == 1)
    {
      keyPressed = 'A';
    }
    else if (column == 2)
    {
      keyPressed = '3';
    }
    else if (column == 3)
    {
      keyPressed = '2';
    }
    else if (column == 4)
    {
      keyPressed = '1';
    }
  }
}
```

Appendix F: Attachments

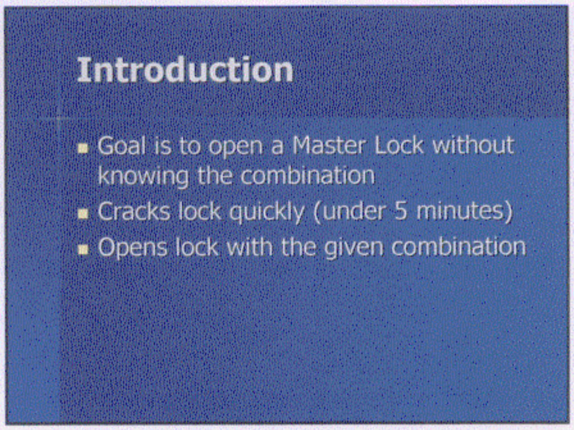
The following documents have been attached to this paper:

- Slides used for the project demonstration
- The original graded project proposal memo.



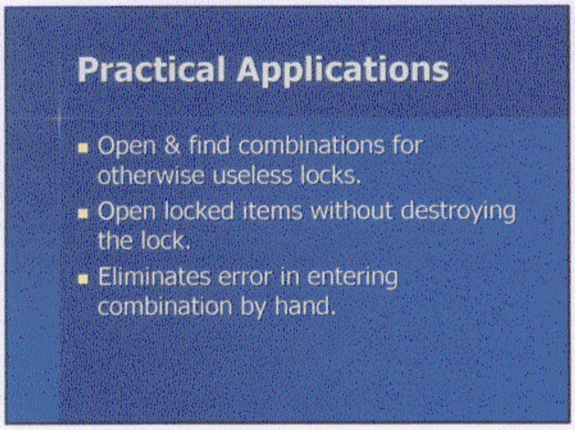
Master Lock Combo Cracker

Jessica Tse
Patrick LaRocque
Matt Leotta



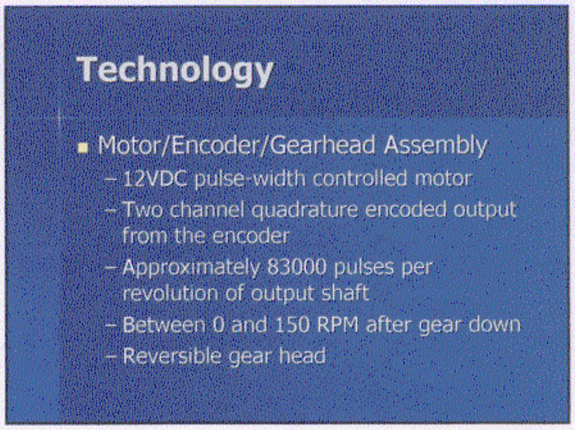
Introduction

- Goal is to open a Master Lock without knowing the combination
- Cracks lock quickly (under 5 minutes)
- Opens lock with the given combination



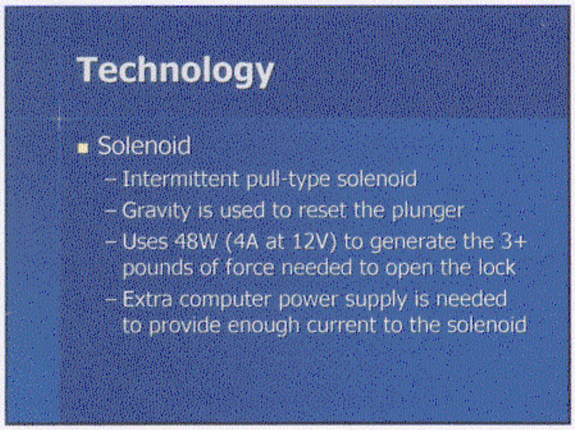
Practical Applications

- Open & find combinations for otherwise useless locks.
- Open locked items without destroying the lock.
- Eliminates error in entering combination by hand.



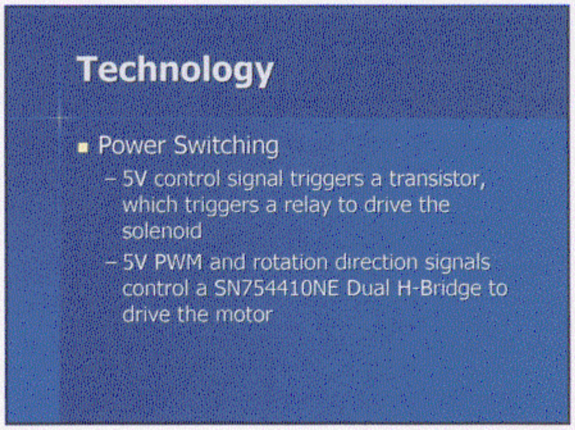
Technology

- Motor/Encoder/Gearhead Assembly
  - 12VDC pulse-width controlled motor
  - Two channel quadrature encoded output from the encoder
  - Approximately 83000 pulses per revolution of output shaft
  - Between 0 and 150 RPM after gear down
  - Reversible gear head



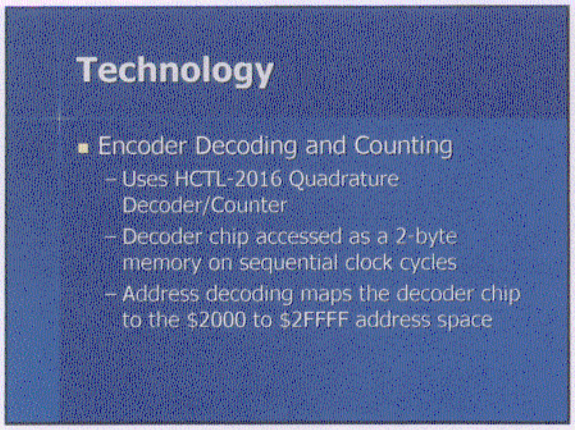
Technology

- Solenoid
  - Intermittent pull-type solenoid
  - Gravity is used to reset the plunger
  - Uses 48W (4A at 12V) to generate the 3+ pounds of force needed to open the lock
  - Extra computer power supply is needed to provide enough current to the solenoid



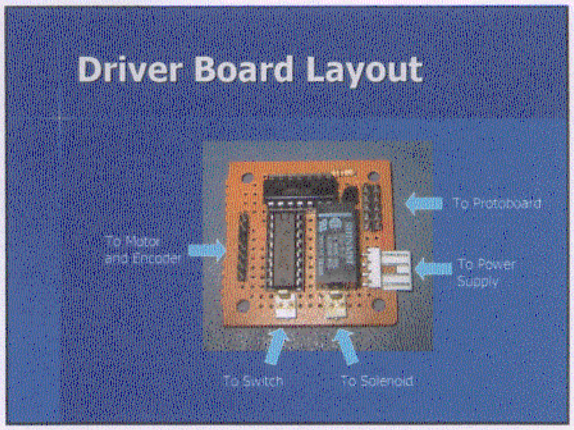
Technology

- Power Switching
  - 5V control signal triggers a transistor, which triggers a relay to drive the solenoid
  - 5V PWM and rotation direction signals control a SN754410NE Dual H-Bridge to drive the motor

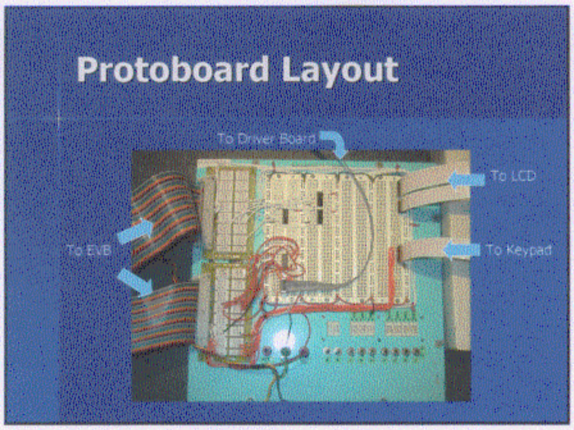


Technology

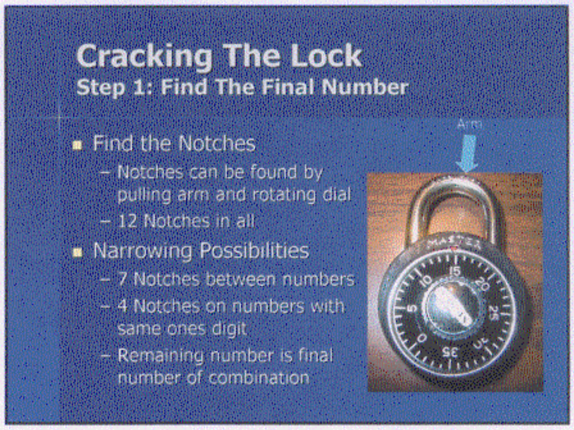
- Encoder Decoding and Counting
  - Uses HCTL-2016 Quadrature Decoder/Counter
  - Decoder chip accessed as a 2-byte memory on sequential clock cycles
  - Address decoding maps the decoder chip to the $2000 to $2FFFF address space



Driver Board Layout

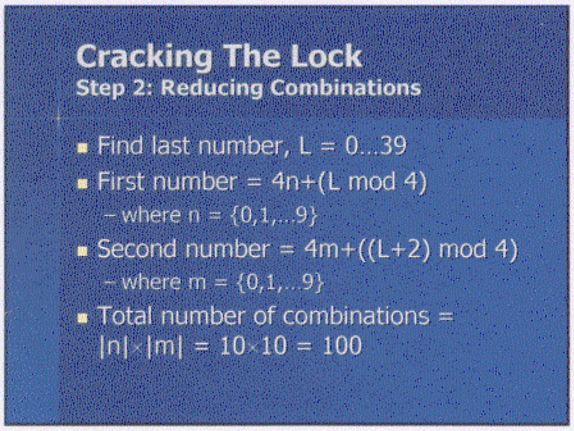


Protoboard Layout



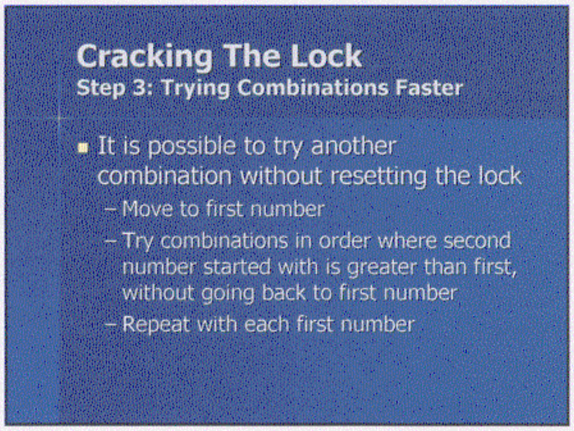
Cracking The Lock
Step 1: Find The Final Number

- Find the Notches
  - Notches can be found by pulling arm and rotating dial
  - 12 Notches in all
- Narrowing Possibilities
  - 7 Notches between numbers
  - 4 Notches on numbers with same ones digit
  - Remaining number is final number of combination



Cracking The Lock
Step 2: Reducing Combinations

- Find last number, $L = 0 \ldots 39$
- First number $= 4n+(L \bmod 4)$
  - where $n = \{0,1,\ldots 9\}$
- Second number $= 4m+((L+2) \bmod 4)$
  - where $m = \{0,1,\ldots 9\}$
- Total number of combinations $= |n| \times |m| = 10 \times 10 = 100$



Cracking The Lock
Step 3: Trying Combinations Faster

- It is possible to try another combination without resetting the lock
  - Move to first number
  - Try combinations in order where second number started with is greater than first, without going back to first number
  - Repeat with each first number

5

MEMO

TO: PROF. R. P. KRAFT

FROM: PATRICK LAROCQUE, MATT LEOTTA, AND JESSICA TSE

SUBJECT: MPS PROJECT PROPOSAL: AUTOMATIC MASTER LOCK OPENER

DATE: 10/15/2002

We intend to design and construct a device to automatically open Master Lock® brand combination padlocks. The device will use a motor and solenoid to manipulate the lock, a keypad and LCD display for user I/O, and a Motorola 68HC12 microcontroller to interface these components and provide a software-based PID controller for the motor. This will require a fair amount of mechanical, electrical, and software design. The completed system will open any Master Lock® either by accepting the three-number combination from the user or by experimentally determining the combination through trial and error. We will also attempt to significantly reduce the time required to find the combination by using a known code-cracking methodology. However, this enhancement will require precise interaction between the solenoid and motor to "feel" the notches in the rotating cam within the lock. Depending on the quality of the parts we obtain and the precision of the feedback available this may prove to be impossible. We are proposing this project because it is challenging and will make use of several topics covered in this course.

Problem Statement

The state of today's society calls for secure storage of one's material property. From this, there is inherently a need for small personal locks; one of the most common of which is the combination Master Lock®. In certain cases, the lock owner may not be able to open the lock easily, due to arthritis or general lack of manual dexterity. In other cases, the combination to the lock may be forgotten and the owner would like to open the lock without damage and retrieve the combination so the lock may be reused. The typical lock owner would only need this service on rare occasions, so this device might be more fitting for a locksmith. Other users of this device could be owners of storage facilities or health fitness centers. In both businesses, customers often are allowed to use there own locks to temporarily store person belongings. When customers fail to remove their locks after the allotted time span, the owner is forced to cut them off. This device provides an alternative in which the owner may remove the lock and then reuse or resell it at a later date.

Proposed Project, Goals, and Approach

The need presented above can be satisfied by developing and implementing a prototype that will automatically open a lock when it is given a corresponding combination and that will have the ability to determine the correct combination by trial and error (possibly enhanced with known code-cracking methodology). For users lacking manual dexterity, the device will enable them to enter three numbers representing the combination.

Attached is a flowchart that shows the flow of control for the motor and solenoid subsystems. The device will rotate the dial of the Master Lock® using a small gearhead motor with an attached encoder for feedback. The feedback loop will consist of a PID controller in software for accurate positioning and velocity control. The solenoid will push up on the hasp (the metal loop) of the lock to attempt to open it. A simple sensor (possibly a pushbutton) will allow the microcontroller to determine if the lock was successfully opened. The user's interface to the system will consist up the LCD panel and keypad available in the MPS lab.

There are several alternatives that could be considered for this project. A more complicated approach to opening the lock would involve using pneumatics instead of a solenoid. With pneumatics it might be easier to produce the required force for opening the lock. It might also be easier to maintain this force while the motor is used to detect notches in the lock's cam. However, pneumatics require a tank to store the compressed air, and actually use solenoid valves to control the air flow. This would only add to the bulk of the device. They are also too slow for quickly checking multiple combinations. Another alternative would be to use a stepper motor to control the position of ~~the lock~~'s dial. This would simplify the code for motor control. However, a stepper motor might have difficultly covering all 40 possible dial combinations, and eliminating the encoder would also make sensing ~~the lock~~'s cam notches impossible.

Although the world will not benefit much from this device (except in the specific instances discussed above), we will benefit by learning how to meeting the difficult challenges this project sets forth. Additionally, we will be able to recover the combinations from some old Master Locks®.

Plan of Activities

We have already begun to design and order parts for the construction of this project. Attached is a detailed task time-line – in the form of a Gantt chart – that shows the details of how we have divided up the work for this project. We will be purchasing several of the components for our design and borrowing others from the MPS lab. The following is a list of the major components and equipment we will need and how we plan to obtain them:

No.	Part Name	Source
1	68HC812A4 w/ EVB	Borrow from the MPS Lab
2	PC w/ Introl Compiler	Borrow from the MPS Lab
3	Protoboard	Borrow from the MPS Lab
4	12VDC motor w/ attached gearhead & encoder	Donated by MicroMo Electronics, Inc.
5	12VDC solenoid	Order online
6	2 – SN754410NE motor/solenoid drivers IC	Order online
7	HCTL-2016 quadrature decoder counter IC	Order online
8	LCD and keypad unit	Borrow from the MPS Lab
9	Master Locks	Purchased from Wal-Mart
10	Various mechanical parts for interfacing the motor and solenoid with the padlock	Order online and purchase from hardware stores

EST. PRICE?

Evaluation

To ensure that the defined problem is solved by the said project, a demonstration of a working prototype will include the following:

- opening a lock with a given combination
- opening a lock with an unknown combination

I WOULD RECOMMEND A STEPPER MOTOR WITH BUILT-IN ENCODER TO SIMPLIFY MOTOR DESIGN UNLESS YOU HAVE ALREADY DONE SOME MOTOR CONTROLLER WORK.

Project Assignment Key

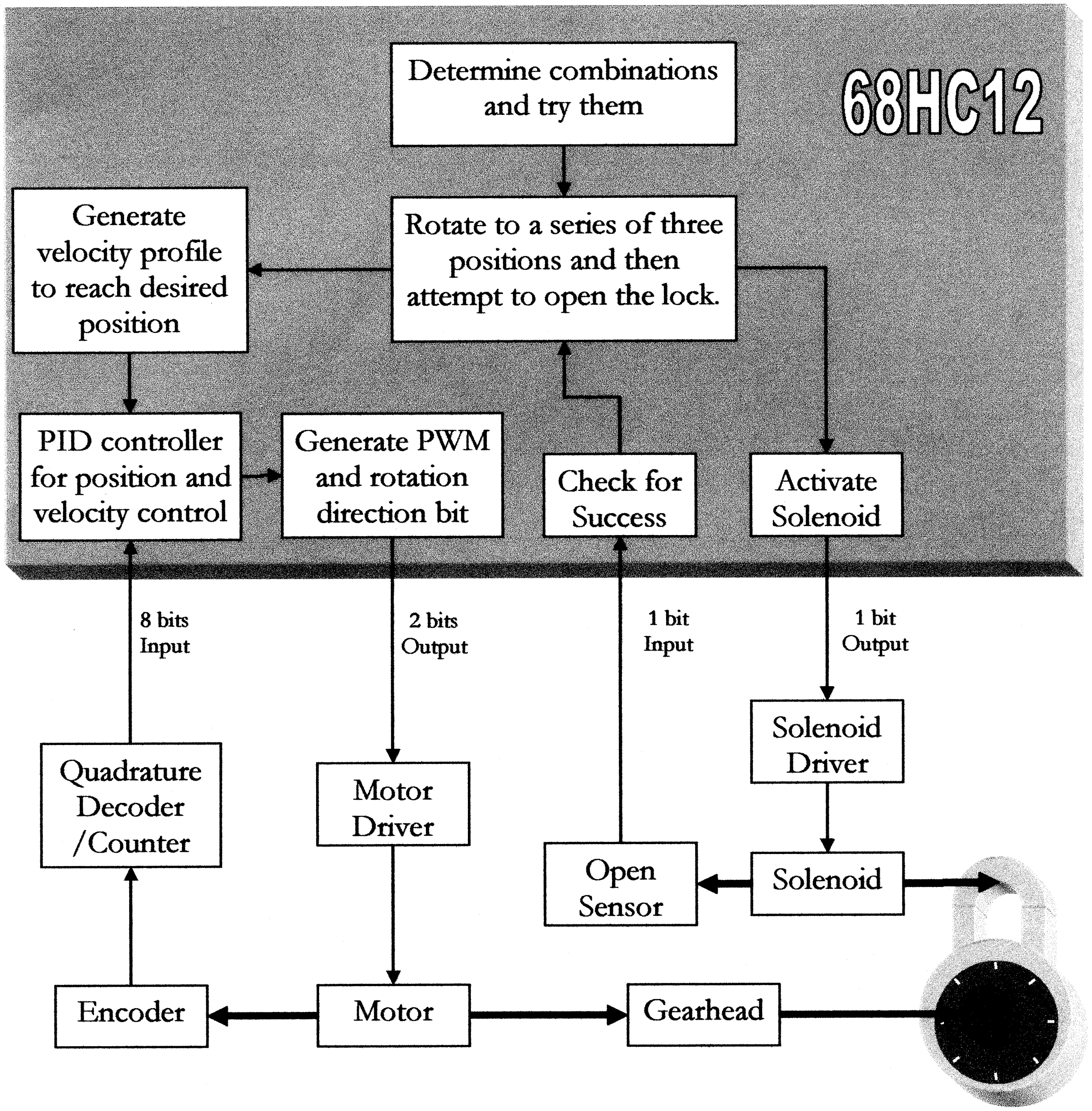
🟥 = Matt Leotta
🟦 = Patrick LaRocque
🟩 = Jessica Tse
🟨 = All

Start date: 10 / 07 / 2002

Project: AUTOMATIC MASTER LOCK OPENER

Current Week	⬇	⬇							
Weeks **Tasks**	10/07 to 10/14	10/14 to 10/21	10/21 to 10/28	10/28 to 11/04	11/04 to 11/11	11/11 to 11/18	11/18 to 11/25	11/25 to 12/02	12/02 to 12/09
Part Ordering	🟨	🟨	🟨	🟨					
-Motor, Controller & Decoder Chips	🟥	🟥	🟥	🟥					
-Solenoid		🟦 🟩	🟦 🟩	🟦 🟩					
Topic Research	🟨	🟨	🟨						
Hardware Assembly			🟨	🟨	🟨	🟨	🟨	🟨	
-LCD and Keypad Assembly			🟩	🟩	🟩				
-Solenoid Assembly					🟦	🟦	🟦	🟦	
-Motor Assembly					🟥	🟥	🟥	🟥	
Software Development			🟨	🟨	🟨	🟨	🟨	🟨	
-User Interface			🟩	🟩	🟩	🟩			
-Solenoid Control					🟦	🟦	🟦	🟦	
-Motor Control					🟥	🟥	🟥	🟥	
-Code Integration							🟩	🟩	
Interim Project Report					🟨	🟨			
Final Report								🟨	🟨
Testing						🟨	🟨	🟨	🟨

FLOWCHART OF LOCK MANIPULATION TASKS


68HC12
Determine combinations and try them
Rotate to a series of three positions and then attempt to open the lock.
Generate velocity profile to reach desired position
PID controller for position and velocity control
Generate PWM and rotation direction bit
Check for Success
Activate Solenoid
8 bits Input
2 bits Output
1 bit Input
1 bit Output
Solenoid Driver
Quadrature Decoder /Counter
Motor Driver
Open Sensor
Solenoid
Encoder
Motor
Gearhead