To the Graduate Council:

I am submitting herewith a thesis written by John Clement Wilson entitled, "Design of a Microcontroller-Based Electronic Pill Counting Device". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Robert Bodenheimer, Major Professor

We have read this thesis
and recommend its acceptance:

Donald W. Bouldin

Frank H. Speckhart

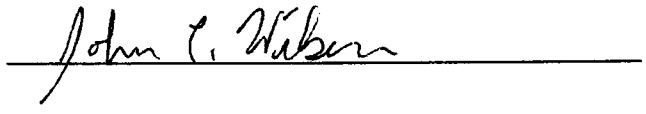
Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledge of the source is made.

Requests for permission for extensive quotation from or reproduction of this thesis in whole or in parts may be granted by the copyright holder.

Signature 

Date 11/15/91

Design of a Microcontroller-Based Electronic Pill Counting Device

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

John Clement Wilson

December 1991

Copyright © 1991 by John Clement Wilson

All Rights Reserved

ACKNOWLEDGEMENTS

There are many people who deserve credit for helping to make my experience at the University of Tennessee one of educational and personal growth. I appreciate the guidance that Drs. Robert Bodenheimer and Frank H. Speckhart have provided me in my work on the pill counter. Their advice of both an academic and personal nature has lead to an enjoyable educational experience. The opportunity to work with these professors, not only on my thesis, but in the development and teaching of the "Control of Electro-Mechanical Devices" course has been informative and exciting.

Thanks to the mechanical engineering graduate students who have provided a place to work and friends to enjoy.

Thanks to my parents for always allowing me to move in the direction I have wanted. And, thanks to Paula for following me to Tennessee to allow me to continue my education. I never would have come without you.

ABSTRACT

This thesis covers the author's design and implementation of the electronics and software used to operate a microcontroller-based pill counter. Development was carried out in two phases. First, a personal computer was used to test the conceptual design. With some modifications to improve the device, a prototype was developed using the Motorola 68HC11EVB. This document describes the systems used to make up the pill counter including stepping motor drivers, photoelectric counters, keypad and LCD display.

The theory behind the operation of the pill counter is to separate a bulk of pills into a single-file "stream" where they are counted using optical means. Separation of the pills is accomplished first by an augured drum, and second by a rotating cone platform. The drum serves as a reservoir for pills and when turning, slowly feeds pills onto the rotating cone, which carries them to an exit chute for counting. The 68HC11 microcontroller has many interesting features that are used to the benefit of the pill counter, and are described in this thesis. Software development followed modern software engineering techniques, specifically those of structured programming, that are described in this document. The motivational factors behind these techniques are also discussed.

TABLE OF CONTENTS

1.0 Introduction 1
1.1 Conceptual Design 4

2.0 PC Implementation 6
2.1 Stepping Motor Control 7
2.2 Counters 8
2.3 Operation 9

3.0 68HC11 Evaluation Board Implementation 11
3.1 The 68HC11 and Evaluation Board 11
3.2 System Overview 13
3.3 Keypad 13
3.4 Liquid Crystal Display 17
3.5 Stepping Motor Drive System 19
3.6 Counter Circuitry 20

4.0 Software 22
4.1 Programming Language 25
4.2 Conceptual Design Phase 25
4.3 Evaluation Board Phase 26

5.0 Conclusion 30

List of References 33

Appendices 35
Software C Code 36
Operational Description 72

Vita 75

LIST OF FIGURES

Figure 1-1: Drawing showing alternative pill counter. 3
Figure 1-2: Conceptual drawing of the pill counter. 5
Figure 2-1: Diagram of personal computer and pill counter used in the conceptual design. 7
Figure 2-2: PC interface to the IMS stepping motor drivers. 9
Figure 3-1: Block diagram of the pill counter (68HC11EVB version). 14
Figure 3-2: Pin connections from the 68HC11EVB to the external electronics. . . . 15
Figure 3-3: Schematic of keypad connections. 16
Figure 3-4: Schematic of connections to the LCD display module. 18
Figure 3-5: Diagram showing timing requirements for the LCD module. 19
Figure 3-6: Photoelectric counters connected to the 68HC11EVB. 21
Figure 4-1: Software life cycle. 23
Figure 4-2: PC main menu. 26
Figure 4-3: Pill counter operational flowchart (68HC11 Version). 29

CHAPTER 1

1.0 Introduction

Odds are, if you went to fill a prescription at your local pharmacy, you would find the pharmacist counting out pills manually, with a broad plastic utensil. Or, in a more advanced pharmacy, the pills would be slowly administered into a bulky electronic machine that passes them through a series of funnels. The most advanced machines for counting pills, found only in large facilities such as hospitals, take up an entire wall with cartridges custom fit to dispense only a certain size pill.

Most automated pill counters available today are designed to have modules for every size and shape of pill, tablet, or capsule made. This takes up a large amount of space and is too costly for the average pharmacy. Other counters have only one mechanism which must be calibrated for a specific tablet size. Recalibration is often difficult and slow.[1]

In a search for patents related to the counting of pills, none of the pill counters were microprocessor controlled. They all used purely mechanical means or extensive solid-state circuitry to count the pills. Most of the pill counters relied heavily on complicated mechanical pieces to operate and were designed for only one size of pill.

As an example, one patent uses a constant RPM motor to rotate the pill bowl.[2] A solid-state counter switches a mechanical gate to divert the pills once the programmed number of pills has been reached. While this device was able to handle different sizes of pills, experimentation showed that it was inaccurate for certain shapes of pills.

On another pill counter, the pharmacist is required to slowly shake pills out of a bottle onto the top of a helical inclined plane.[3] Pills slide down the plane and pass over a detector. The detector is connected to a combination counter and three-digit display. This pill counter requires the constant attendance of the pharmacist, not only to watch the count, but to administer the pills as well.

Figure 1-1 shows another more automated pill counter that uses two counter-rotating turn-tables to put pills into an adjustable-width passage.[4] At the end of the passage is a photoelectric sensor used for counting. A mechanical gate connected to the counter, switches in front of the pill path to deflect pills into an overflow bin once the desired quantity of pills has been reached. This pill counter requires adjustment of the discharge passage width each time a different size pill is to be counted. A pill "decapper", which prevents the stacking of pills, also must be adjusted by the pharmacist.

The need for a more efficient pill counter with the ability to easily handle universal size pills became apparent through these observations. This laid the foundation for the development of an electronic device that automatically counts a specified number of small objects. Objects may range in size and shape, but the device is specifically designed for the counting of various size pharmaceutical pills, tablets, capsules, etc. The device is referred to as the pill counter.

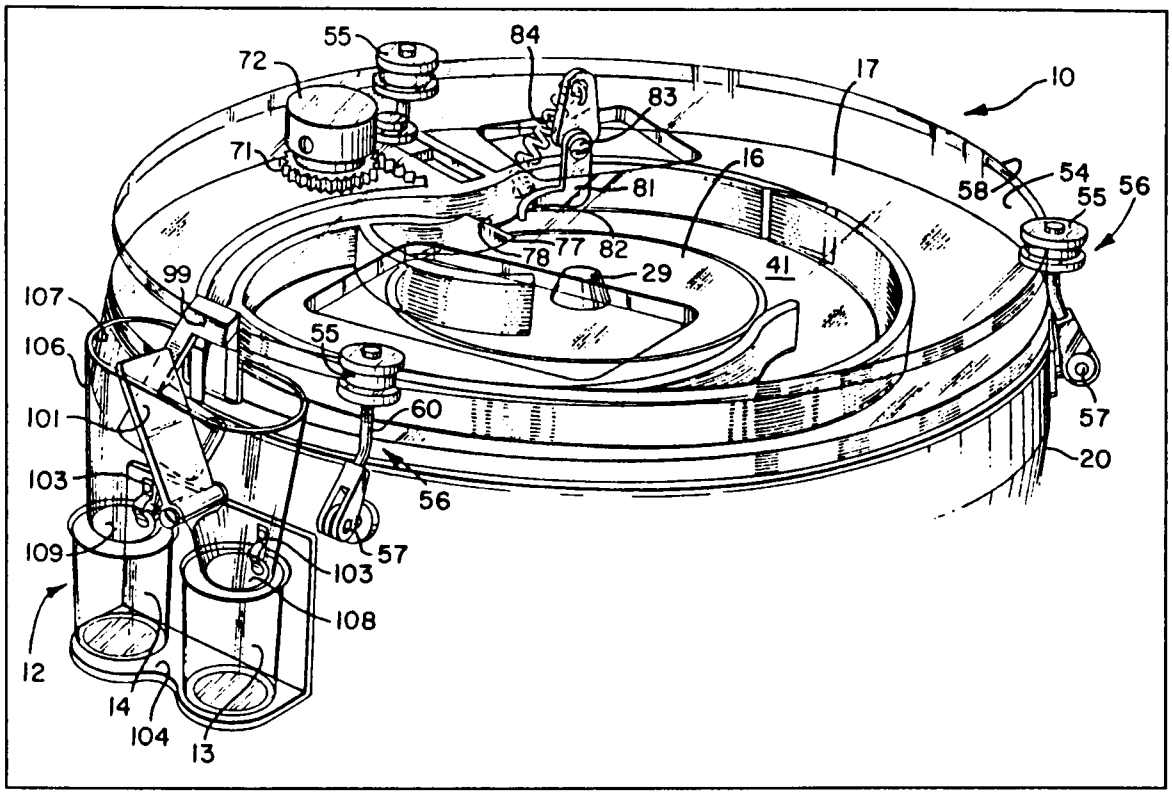


Figure 1-1: Drawing showing alternative pill counter.

Source: United States Patent Number 3,837,139. Inventor: Harry G. Roseberg.

1.1 Conceptual Design

Dr. Frank H. Speckhart, a professor in mechanical and aerospace engineering at the University of Tennessee, originated the idea for the mechanical aspects of the pill handling. He designed and built the mechanical hardware that makes up the pill counter, thus providing the framework from which the electronics and software were developed. The following describes the conceptual design.

The theory behind the operation of the pill counter is to separate a bulk of pills into a single-file "stream" where they are counted using optical means. Separation of the pills is accomplished first by an augured drum, and second by a rotating cone platform as shown in Figure 1-2. The motion of the pills begins at the base of the drum. Pills are augured up the drum onto the cone where they are rotated around until they fall down through the exit chute where they are counted. The operation continues until the specified number of pills have fallen down the chute.

This thesis describes the development and implementation of the electronics and software that are used to operate the pill counter. Chapter 2 presents the conceptual design and the testing of its feasibility with the aid of a personal computer. An explanation of the conversion from conceptual design to a fully operational prototype follows in Chapter 3. Software goals and development through all phases of the project are the topic of Chapter 4.

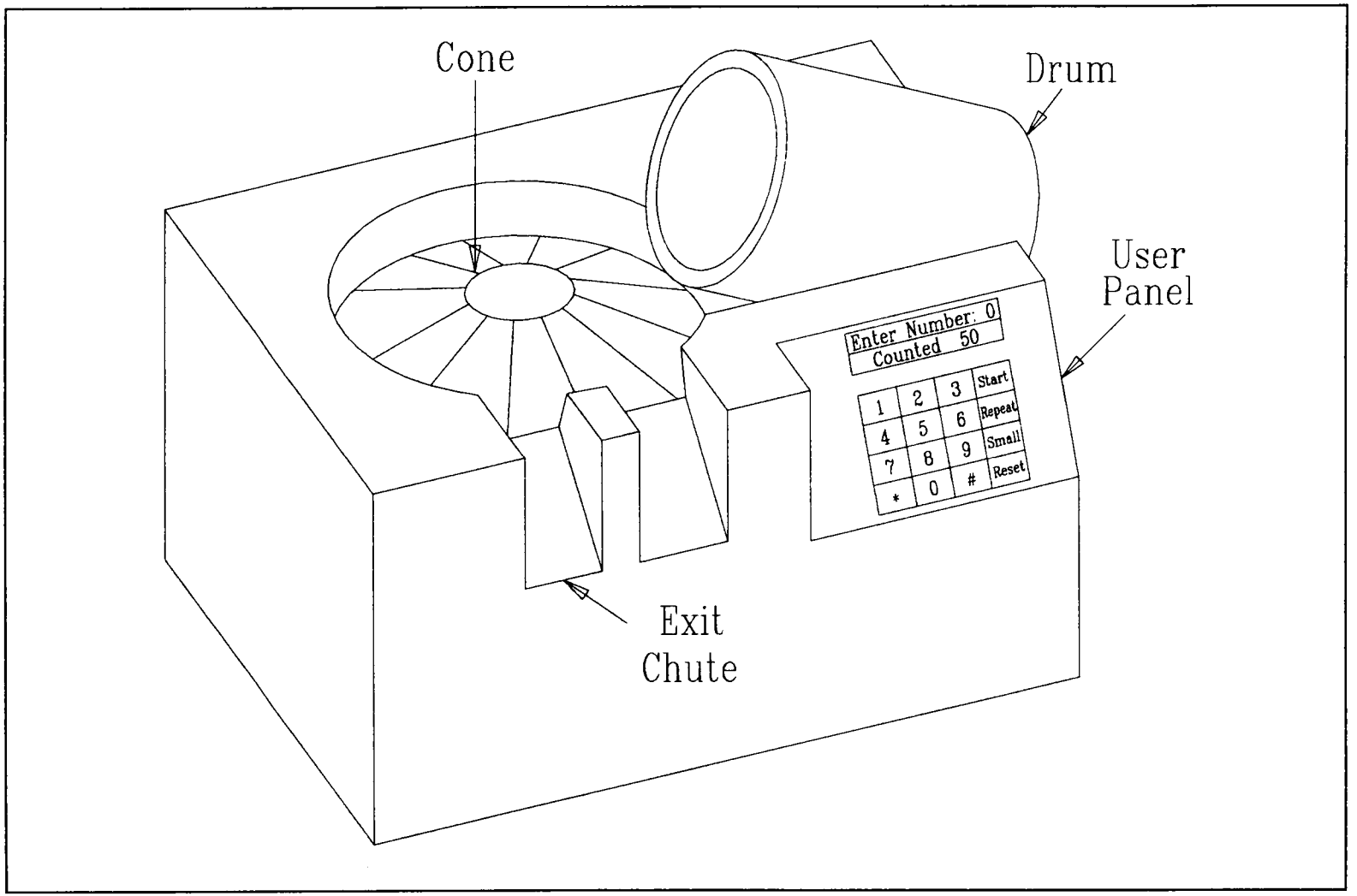


Figure 1-2: Conceptual drawing of the pill counter.

CHAPTER 2

2.0 PC Implementation

In the conceptual design, the following was determined to be the basis of operation of the pill counter. The drum serves as a reservoir for pills and when rotated, slowly feeds pills onto the cone. At the base of the pill counter are two bins. At the completion of one cycle, the left bin holds the desired quantity of pills. The right bin serves as an overflow and so contains all excess pills that were initially put into the drum.

In a design of this magnitude, it is important to know if the concept is feasible. A *flexible* electronic system used to control the mechanical system aids in the development of a complex device such as the pill counter. This type of electronic system not only provides the mechanical systems designer with a way of testing the mechanical system, but also a way to quickly and easily change the characteristics of the machine. A microprocessor-based system provides ample flexibility. The most efficient way to verify the feasibility of the conceptual design is by using a readily available personal computer (PC). Figure 2-1 shows the PC with components used to test the conceptual design. By using the keyboard and monitor to interface to the user, additional hardware for this purpose was not needed.

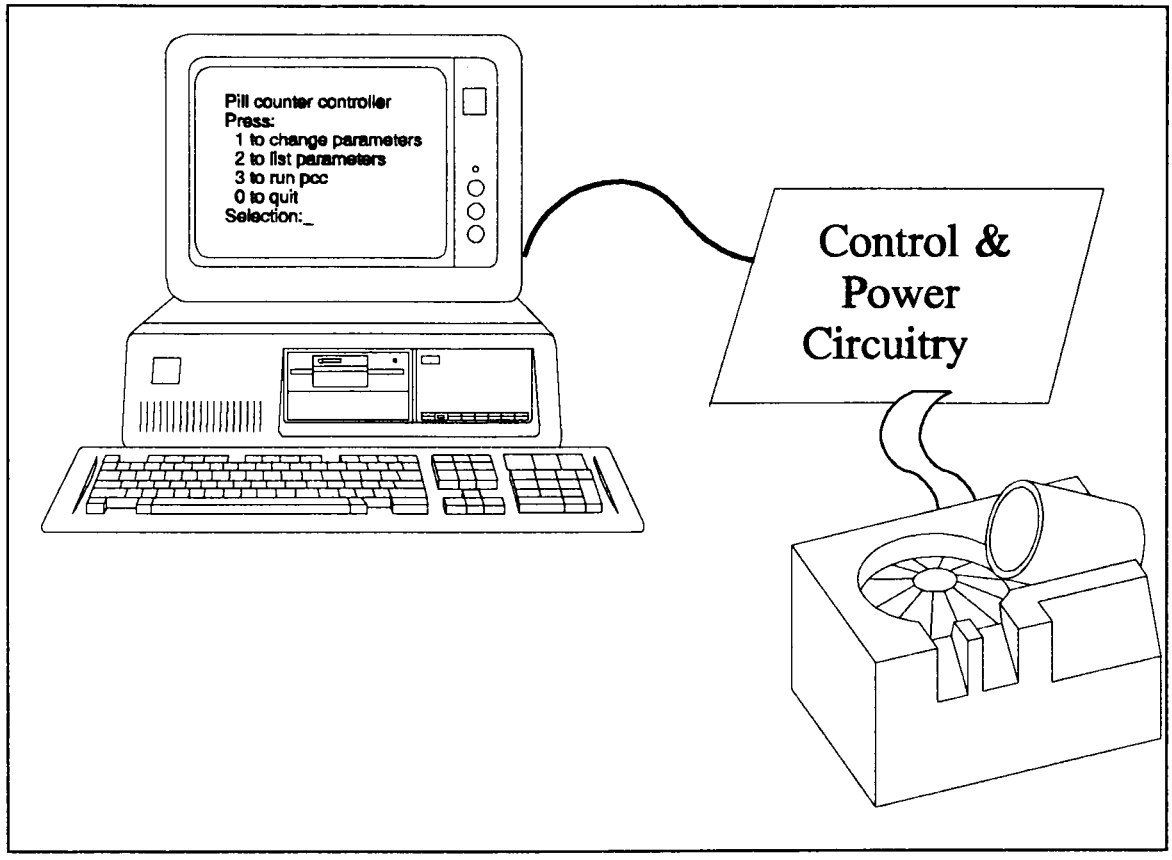


Figure 2-1: Diagram of personal computer and pill counter used in the conceptual design.

2.1 Stepping Motor Control

The given conceptual design contains two mechanical devices, the drum and the cone, that need to be controlled by the microprocessor. Since the cone's speed and direction change during operation, a stepping motor drive would allow for the most versatile operation. The drum's speed is also variable, so a stepping motor is also used for its drive.

In order to save development time, Intelligent Motion Systems' (IMS) commercial stepping motor drivers were initially used. These drivers could be easily interfaced to the PC and did not require time to design and assemble a discrete logic driver. The IMS

driver has four logic input lines that control the motor. Their labels and functions are as follows:

- Enable - A logic low enables the driver, applying a holding torque to the motor. A logic high disables the driver, so that no current is flowing through the motor. This line is used to disable the motors when the pill counter is not in operation, thereby preventing heat build-up in the motors and drivers.

- Half/Full - This line chooses between half and full step drive. For the 1.8° stepping motors used, there are 200 full steps per revolution and 400 half steps per revolution. Full step motion is rough, allowing for more separation of the pills, and is used with both the cone and drum. Half step motion is smoother than full step and is used with the cone to count out the last few pills.

- CW/CCW - This line chooses the direction of motion for the motor and is used to reverse the cone when emptying the pills into the overflow bin.

- Step Clock - A falling edge on this input steps the motor in the direction and step mode specified by the CW/CCW and Half/Full lines. The enable line must be low for the motor to step.

The interface from the PC to the drivers is shown in Figure 2-2. By controlling the input lines to the IMS drivers, the PC operated the cone and drum motors exactly as desired.

2.2 Counters

The other electronics required to test the conceptual design are the counters. Photoelectric devices provide a very fast response time and are inexpensive. They only give inaccurate results when two pills pass the photosensors side by side. In order to maximize the accuracy of the pill counter, two sets of photosensors were implemented. One LED-phototransistor pair is located at the top of the chute and another pair is located at the bottom of the chute. The microprocessor keeps track of the number of pills that

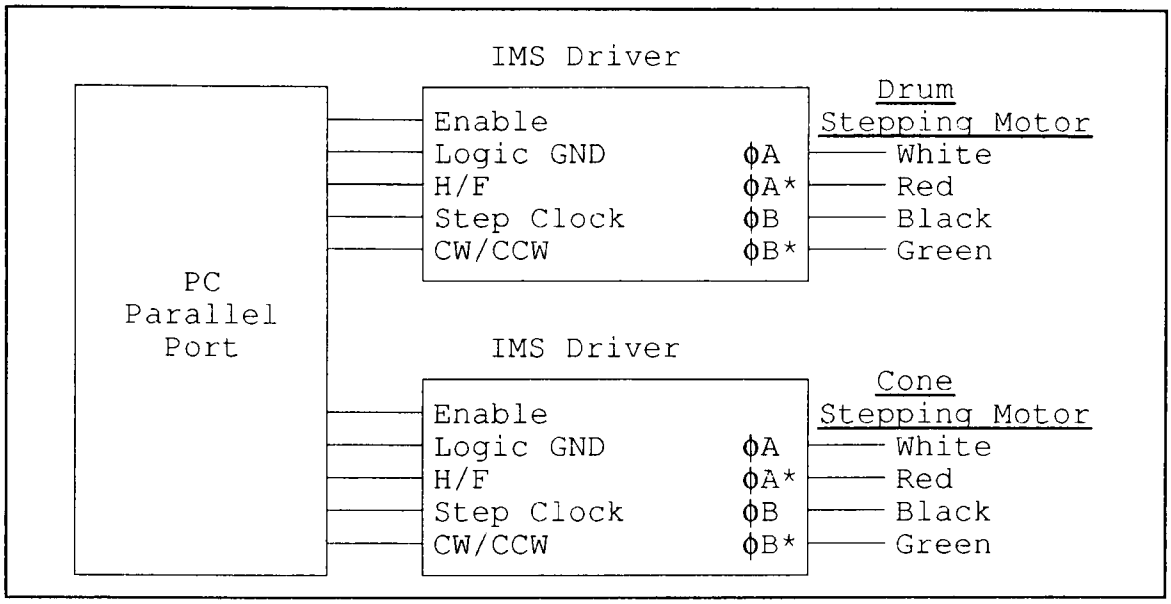


Figure 2-2: PC interface to the IMS stepping motor drivers.

pass through each pair of photosensors and assumes the highest count to be the most accurate. A dual 4-bit counter chip helps the microprocessor keep track of the separate counts.

2.3 Operation

Once the designer has entered the operational parameters and a desired quantity of pills using the computer keyboard, the drum motor begins to turn, dropping pills onto the cone. The cone motor rotates carrying the pills to the exit chute. By alternating between stepping the motor a few steps and stopping for a step, a vibration was introduced into the cone that enhances pill separation. The computer screen is continually updated to display the present count. Once the counted quantity reaches to within two of the desired quantity, the drum and cone slow down to count the final few pills. In

order to clear the device, the drum rotates quickly and the cone rotates in reverse so that extra pills are deposited into the overflow bin.

There are many parameters that affect the accuracy and characteristics of the pill counter, such as the speed of the drum and cone, the vibration mode of the cone, and the length of time that the cone runs in reverse for emptying pills. Any of these parameters can be changed at run-time to enable the mechanical systems designer to fine tune the system for optimal performance without the assistance of the programmer.

During testing, it was discovered that pills would fall out of the drum in an irregular fashion corresponding to the internal threads of the drum. The resulting empty areas on the cone indicate an inefficient counting process. A more even distribution of pills is achieved if the drum is rotated slowly when the pills are falling onto the cone, and rotated quickly when the pills are not coming out of the drum. A proximity detector was added to the system to tell the PC when pills are not falling out of the drum. The PC now knows when to speed up the rotation of the drum to move past the area where pills are not coming out.

With this addition, the conceptual design of the pill counter proved to work as desired, so design continued with the objective of making the device more self-contained. It was evident that a separate microcomputer from the personal computer would be selected to run the pill counter.

CHAPTER 3

3.0 68HC11 Evaluation Board Implementation

The personal computer supplied an excellent platform for the development and testing of the conceptual design of the pill counter. However, the PC is too large and expensive to use for this task. A self-contained unit using an embedded microprocessor would yield a more attractive product. The Motorola 68HC11 is the microprocessor chosen to operate the pill counter.

Other components had to be integrated with the self-contained unit to take the place of the PC's keyboard and monitor. A 16-key keypad allows the user to specify the number of pills to be counted and the mode of operation. A 2-line by 16-character liquid crystal display (LCD) shows the status of the machine and provides feedback to the user.

3.1 The 68HC11 and Evaluation Board

A microcontroller is a microprocessor with memory systems and peripheral devices built into one compact integrated circuit. By using the Motorola 68HC11 microcontroller, the amount of external circuitry needed to operate the pill counter is greatly reduced, compared to using a base microprocessor. Using the microcontroller also simplifies and reduces the size of the circuit board needed for the final product.

The 68HC11 is an 8-bit microcontroller unit (MCU) produced using high-density complementary metal-oxide semiconductor (HCMOS) technology. On-chip memory systems include 256 bytes of random-access memory (RAM), 8K bytes of read-only

memory (ROM), and 512 bytes of electrically erasable programmable ROM (EEPROM). Some of the major peripheral functions include an 8-bit analog-to-digital converter, asynchronous serial communications, a timer system, and an 8-bit pulse accumulator system.[5]

The 68HC11 microcontroller has many interesting features that are used to the benefit of the pill counter. First, the 512 bytes of on-chip EEPROM are used to store parameters controlling the speed and length of time the drum and cone motor run. Second, the event counter serves as a means to keep track of the count in one part of the chute. Also, the edge-triggered interrupt line of the input capture system provided a means to create an additional internal counter. A self-strobing feature eliminated complex hardware decoding and software to interface to the liquid crystal display. The features used in the implementation of the pill counter are described in more detail as needed.

To aid in the development of a system based around the 68HC11, Motorola has designed and manufactured an evaluation board called the 68HC11EVB. An easy to connect interface between the 68HC11 and the components that operate the pill counter is provided by this evaluation board. There is a 52-pin connector on the EVB that "looks like" the 52-pin 68HC11. The 68HC11EVB (EVB) is such an accurate emulation of the 68HC11, that once the software program has been developed and the electronics finalized, the EVB can be replaced directly with the 68HC11 microcontroller.

By using the EVB during development, testing of the program and attached hardware is much simpler and faster than using the straight MCU. The EVB is set up for serial communications to a PC, providing a fast method of cross-assembly, downloading,

and testing of programs. BUFFALO, the monitor program used to run the EVB, allows the designer to set break points and step through the downloaded code to aid in the debugging of both the software and electronic hardware.

3.2 System Overview

An overall block diagram of the system is shown in Figure 3-1. The pill counter electronics can be broken down into the control unit (EVB board) and four subsystems: the keypad, liquid crystal display module, stepping motor drive system, and counting circuitry. Each of these subsystems is described below. A detailed diagram showing the connections between the EVB and the external electronics is in Figure 3-2.

3.3 Keypad

A 4x4 matrix keypad provides a straight-forward method for the user to communicate with the pill counter. As seen in Figure 3-3, the keypad is interfaced to Port C of the 68HC11, which provides both input and output lines. Lines 4-7 of Port C are configured as outputs and are connected to the four column-pins of the keypad. Lines 0-3 of Port C are configured as inputs and are connected to the four row-pins of the keypad.

In order to tell if a key has been pressed, bits 4-7 of Port C are set to logical zeros and bits 0-3 are monitored. If any key is pressed, contact is made between a row and column line causing one of the row-pin's voltage to drop as current flows through the pull-up resistor. By reading Port C, the microcontroller can tell which row the key

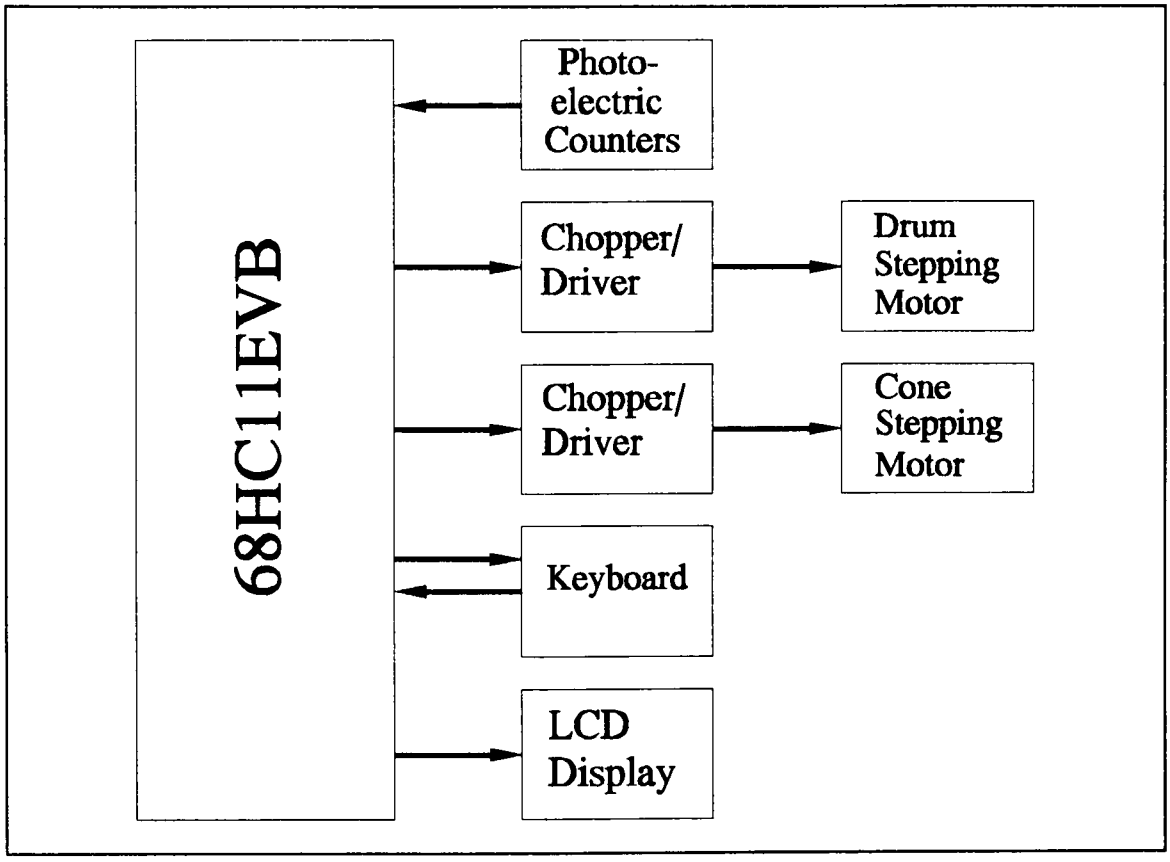


Figure 3-1: Block diagram of the pill counter (68HC11EVB version).

depressed was in. The exact key can be determined by sequentially setting each column low, and checking the row value for a logic zero.

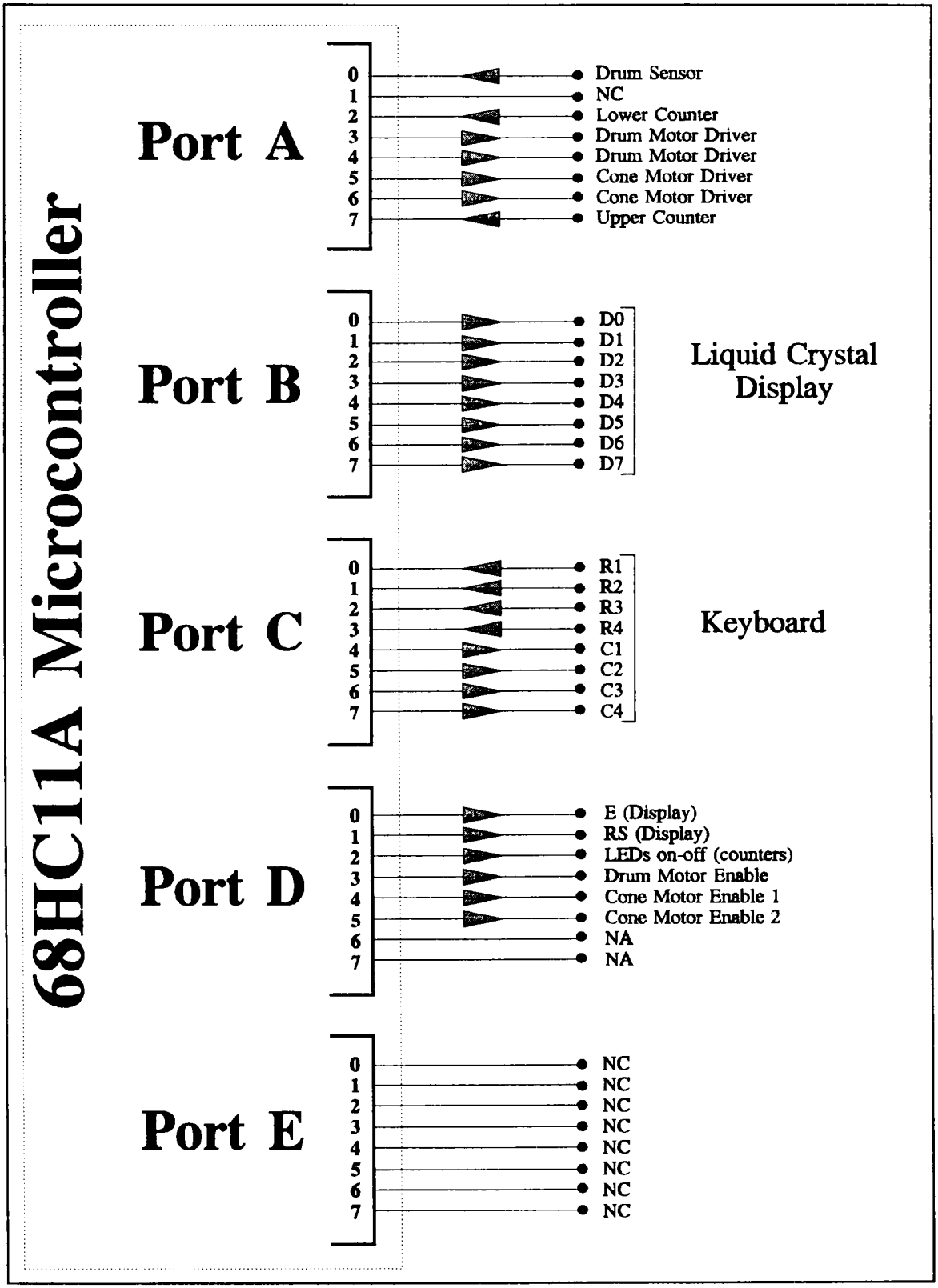


Figure 3-2: Pin connections from the 68HC11EVB to the external electronics.

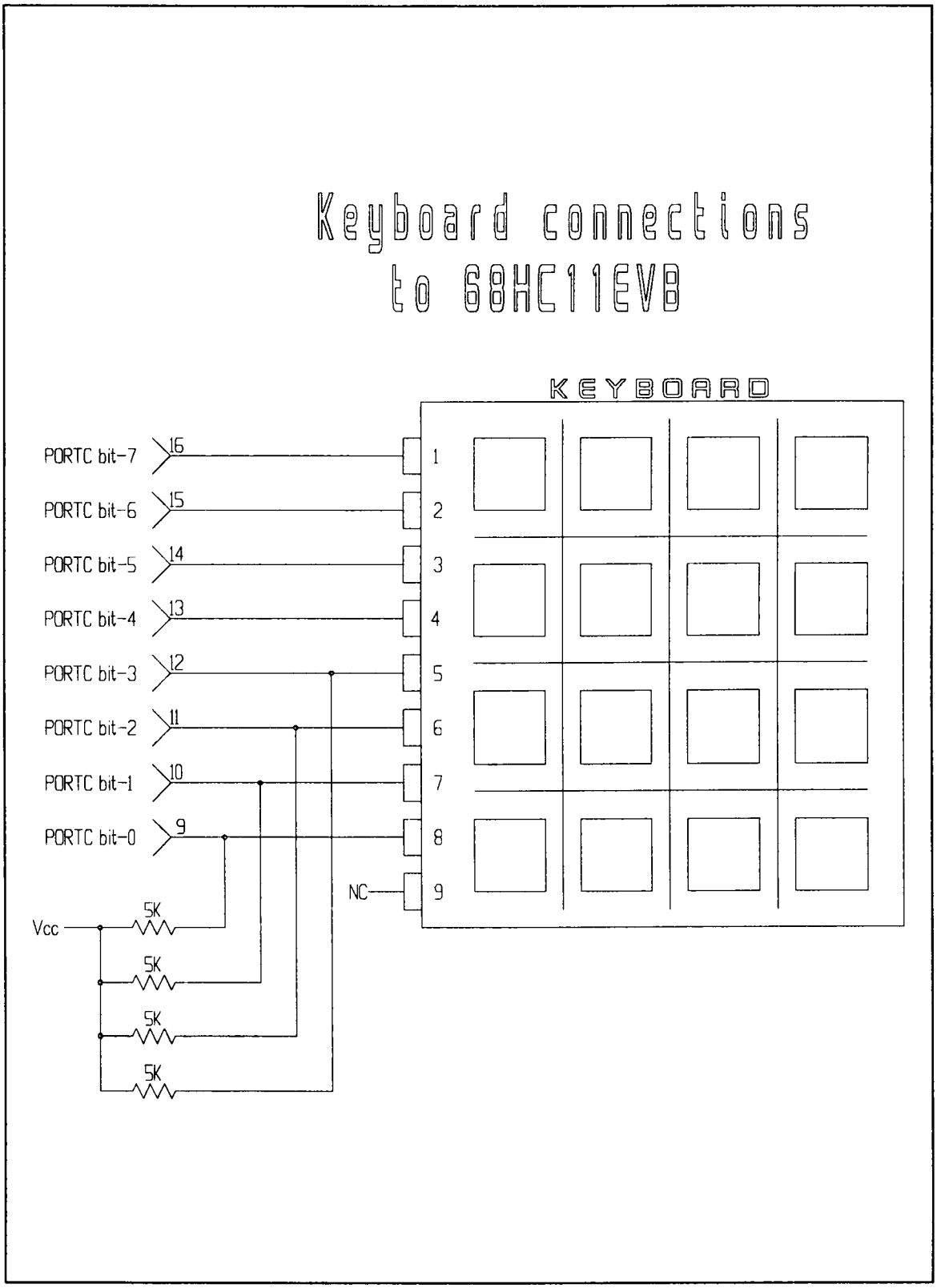


Figure 3-3: Schematic of keypad connections.

3.4 Liquid Crystal Display

The Philips LTN211 is a two line by sixteen (2X16) character LCD module, with each character consisting of a 5X7 dot matrix and cursor. All the controls and memory needed to operate the LCD are built into the module. A simple 8-bit interface to the MCU shown in Figure 3-4 is all that is needed to provide the module with the characters that are to be displayed on the LCD screen. The interface consists of eight data lines (D0-D7), a register select line (RS), a read/write select line (R/W), and an operation enable line (E).[6]

The data bus from the 68HC11 to the LCD module is set up through the 8-bit output Port B. Since Port B is a straight output port, and the timing requirements for writing to the module can be met, the read/write signal on the module was tied low rather than interface it to the MCU. Therefore, the data bus is only in one direction, from the MCU to the LCD module.

The register select signal tells the module whether the incoming data is an instruction or a character to be displayed. Sending instructions allows positioning of the cursor and access to special modes. Bit 1 of Port D is used to control the RS line.

To complete the interface, the final signal needed is the enable line used to latch in the data on the bus. Figure 3-5 shows the timing requirements to ensure the LCD module receives the correct data. The 68HC11 provides a simple strobe mode for the handshake I/O subsystem associated with Port B that meets these timing requirements. By connecting the STRB (strobe-B) pin of the 68HC11 to the E line of the display, values

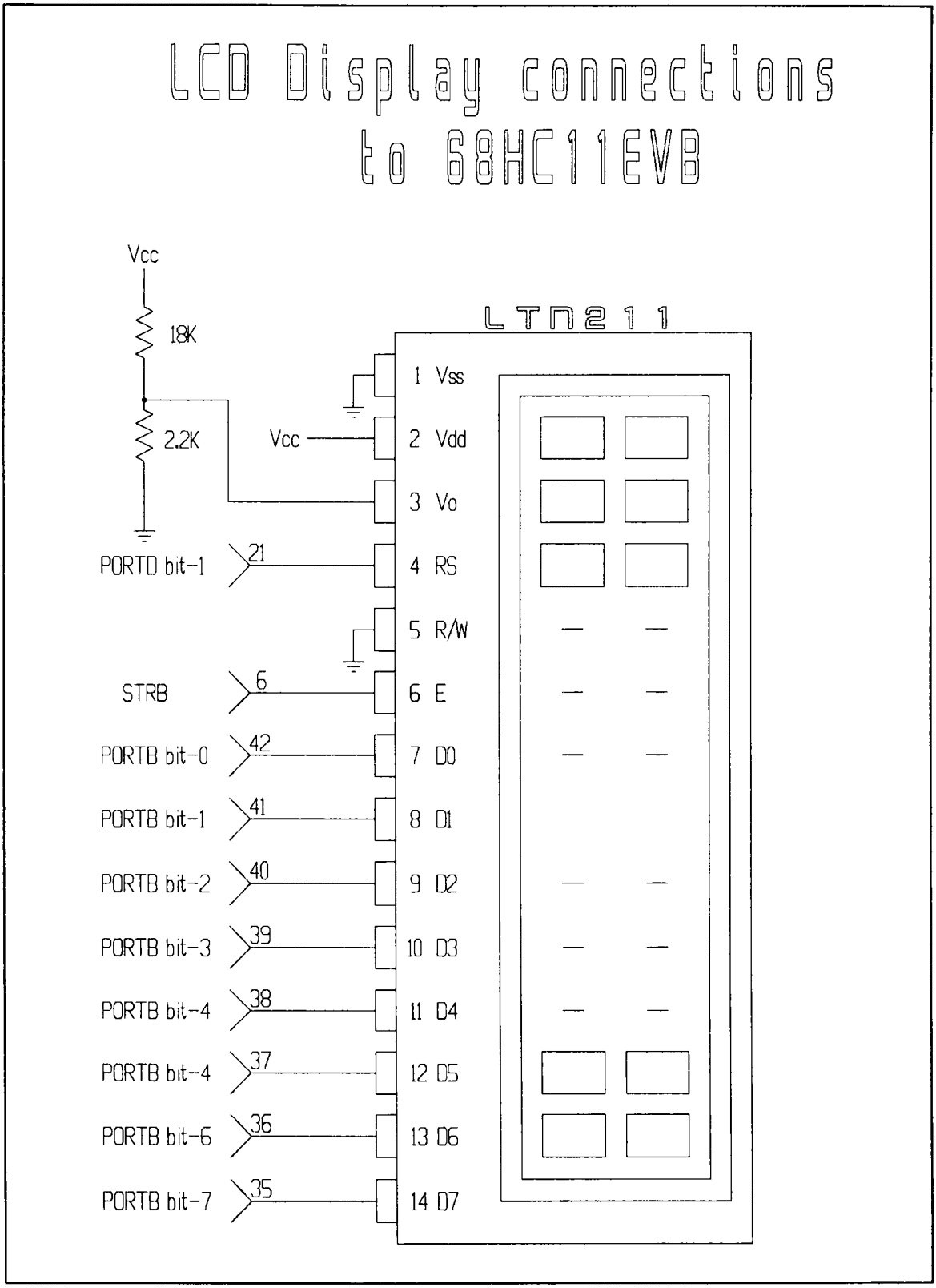


Figure 3-4: Schematic of connections to the LCD display module.

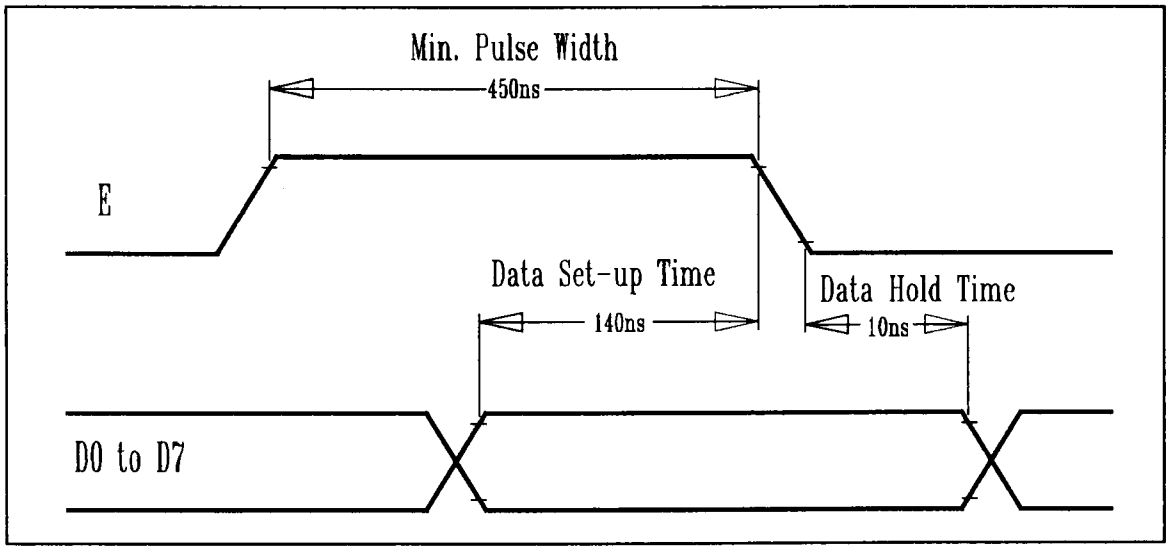


Figure 3-5: Diagram showing timing requirements for the LCD module.

written to Port B are automatically latched into the display module. Once the cursor is positioned, a string of characters written to Port B will appear on the display.

3.5 Stepping Motor Drive System

The stepping motor is the ideal instrument for computerized motion control since each revolution of the rotor is divided into discrete steps. The motors used in this application have 200 steps per revolution. Speeds of the motors are easily controlled by the microprocessor by varying the frequency of the steps. In order to drive the stepping motors in bipolar mode, a single-chip chopper/driver (SGS-Thomson TEA3718) is used for each winding of the motor. These drivers use 12 volts as their source and utilize internal chopping circuitry to limit the current applied to the motors.

There is only a small number of control lines needed to drive each motor. The drivers require two lines to step each motor using full step (200 steps per revolution) as is done on the drum motor. Since the cone motor is driven using both full and half steps

(400 steps per revolution), two additional control lines are needed. The lines from the 68HC11EVB to the chopper/drivers are allocated between Port A and Port D according to Figure 3-2.

3.6 Counter Circuitry

In order to check the quantity of pills falling down the chute into the bin, optical sensors were used. By placing an LED and a phototransistor facing each other on each side of the chute, objects that break the light beam from the LED are detected by the phototransistor. Optically matched pairs of phototransistors and LEDs were chosen so external light from the room would not affect their performance.

Rather than connecting the LEDs through a resistor to V_{cc}, they are powered through a latch controlled by the 68HC11. This connection gives the software program control over whether the LEDs are on or off. The reason this was done will be described in Chapter 4.

The event counter available on the 68HC11 is an ideal choice to count the pills as they come down the chute. The phototransistor is connected through a Schmitt-Trigger invertor to reduce noise before it is connected to the event counter through bit-7 of Port A. In order to increase the accuracy of the total system, two counters are needed, but only one is available on the 68HC11. To create another counter, the input-capture system is used to generate an edge-triggered interrupt each time a pill breaks the light beam. When this occurs, the interrupt service routine increments a variable counter. Figure 3-6 shows how the LEDs and phototransistors are wired.

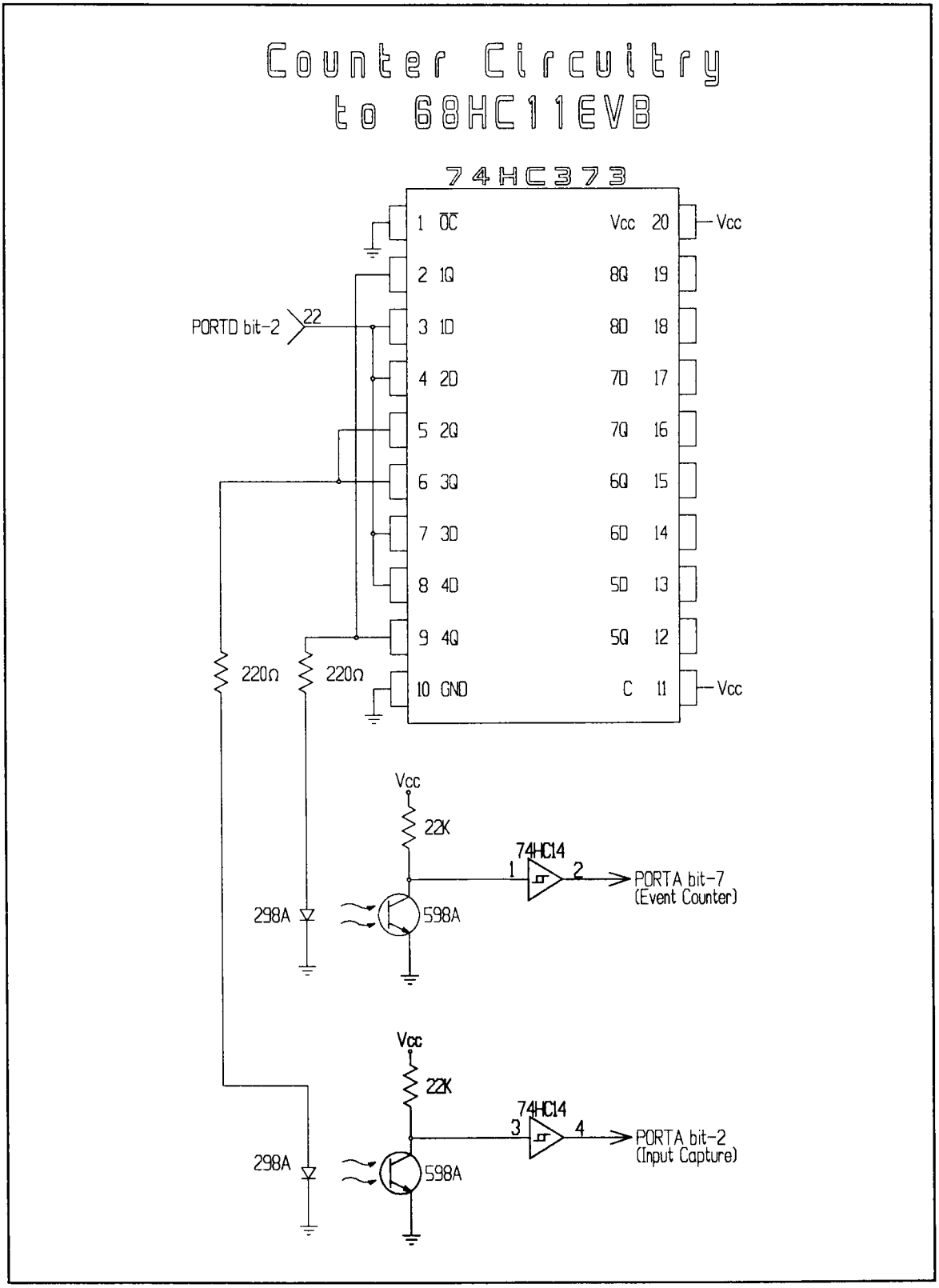


Figure 3-6: Photoelectric counters connected to the 68HC11EVB.

CHAPTER 4

4.0 Software

In order to control the electronics and hardware of the pill counter, a sophisticated software program is required. During operation, the microcontroller not only controls the motors and display, but also must simultaneously monitor the counters and keypad. To accomplish these tasks without a detectable loss of performance, the software must be designed efficiently. The significant complexity of the required software demanded that the program be written in an organized and thoughtful way. Techniques of modern software engineering, specifically those of structured programming, were followed with the utmost attention.

The life cycle of software covers the period of time from the conception of an idea to the retirement of a given software product. A traditional software life cycle model is shown in Figure 4-1.[7] The model shows that 67 percent of all life-cycle costs are attributed to the maintenance phase. Therefore, it is important during the development cycle to create a program that can be maintained easily. When this is done successfully, the resultant program is called *maintainable*. Another important characteristic of a well-written program is *testability*. This is the ease to which test criteria can be established and the code can be evaluated for correctness.[8]

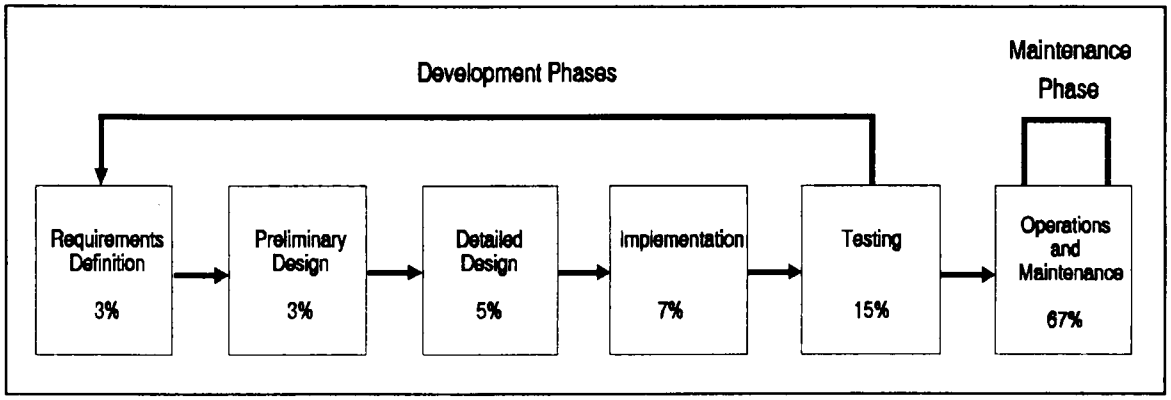


Figure 4-1: Software life cycle.
Source: Lowell Jay Arthur, *Software Evolution* (New York: John Wiley & Sons, Inc., 1988), p. 4.

There are various rules to follow during software development that will aid the programmer in creating a maintainable and testable program. The following paragraphs describe some of the guidelines used in the development of the pill counter software and the motivation behind these guidelines.

From the beginning of software development, code was separated into pre-determined modules and written using a top-down style of programming. The functional modules were developed and tested separately, and later brought together to make up the final product. A description of the individual modules is given in Table 1.1.

Modular programming reduces the complexity of the problem by breaking up a software package into a number of simpler pieces. These pieces are manageable in size, separately testable, and separately modifiable.[9] The software that results from using modular programming techniques is more easily testable and maintainable.

Even though the code developed to operate the pill counter is extensive, it can be maintained by someone other than the author. For example, if a problem occurs with the

Table 1.1 Modules that make up the pill counter software

Name	Description
Main:	Initializes system and begins execution.
Display:	Contains all functions controlling the display.
Keyboard:	Reads keys from the keypad. Converts separate keystrokes into a value and mode; and returns this information.
User:	Controls the user interface.
Control:	Synchronizes all operations while counting pills, including running the drum and cone.
Count:	Contains all functions relating to the counters. Keeps track of the present count and the quantity desired. Returns parameters to the control module to tell when to slow down the drum and cone and when to stop.
Init:	Responsible for initializing electronics and appropriate variables upon power-up.

keypad, then the maintenance programmer can look at the keyboard module without needing to know the details of any other modules. Other techniques add to the maintainability of this software:

- Each function within a module begins with a header that describes its purpose, any input variables, return variables, and/or modifications to global variables.
- Descriptive variable and constant names along with in-line comments make the code much more readable.
- GOTO statements are never used, only structured programming constructs such as IF, CASE, FOR, and WHILE.

- Variables are kept local to the function or module whenever possible. Global variables make code extremely hard to understand because the programmer must search through all the modules to find out where these variables are used or modified.

4.1 Programming Language

Software was developed using the high-level language C. C was chosen for its powerful high-level instruction set, while retaining the ability to manipulate expressions at the bit level. The ability of a language to use the same code on different microprocessors is called *portability*. C is considered a portable language, especially since ANSI (American National Standards Institute) developed a standard for compiler developers to follow. ANSI C compilers are available for both the PC and the 68HC11, so the code developed for the PC was directly transferable to the 68HC11. Another important benefit of C is its popularity and broad usage, adding to maintainability of the software.

4.2 Conceptual Design Phase

During the conceptual design phase, using C with the personal computer provided an easily implemented user-interface with the ability to control the pill counter. An important design goal was to make the system flexible for the mechanical systems designer. By allowing parameters that affected how the pill counter operated to be changed by the user at run-time, the mechanical systems designer was able to fine-tune the system quickly and without programmer assistance. The main menu used for the conceptual design is shown in Figure 4-2.

```
Prescription Pill Counter Controller

  Press:
    1 to change parameters
    2 to list parameters
    3 to run pcc
    0 to quit
  Selection: _
```

Figure 4-2: PC main menu.

The PC aided in the development of additional software that was used in the 68HC11EVB prototype. Using Turbo C, the keyboard and display modules were independently developed and tested. Communication with these peripheral devices was done through the parallel port of the PC. The tested software was later combined with the other modules running on the 68HC11EVB.

4.3 Evaluation Board Phase

In order to take advantage of the C code already developed during the conceptual design phase, it was decided that a C compiler for the 68HC11 was needed. A C cross-compiler that runs on the PC compiles C code into 68HC11 machine code. With a few hardware-specific changes to the existing code and by bringing together the independent modules, a new version of the software was running on the 68HC11EVB within a day.

As mentioned in Chapter 3, the top photodetector was connected to the 68HC11's 8-bit on-chip event counter. Slight modifications in the count module were made to account for this change. At the beginning of each new set of pills, the event counter is

set to zero. During operation, the program monitors the PACNT register to see how many pills have fallen through the top part of the chute. Another counter was needed for the photodetector at the bottom of the chute. The easiest way to implement this was to have a counter chip connected to the photodetector as was done during the conceptual design phase. In order to save having an additional chip on the circuit board, however, a software counter was implemented using an edge-triggered interrupt. Each time a pill falls through the bottom part of the chute, the program is interrupted to increment a variable keeping track of the count. An interrupt service routine was written for this purpose.

The software on the PC allowed the mechanical systems designer to change operational parameters during run-time. This feature had added so much flexibility into the system that it was carried over into the 68HC11 version. Modifications to the software were made so that these parameters are stored into the on-chip EEPROM. By storing them into non-volatile memory, the operational characteristics are remembered when the power is turned off. This feature allows each production pill counter to be tailored to a specific application.

A self-testing system has the ability to independently check itself for component malfunction. This feature increases the reliability of a system by alerting the user of any detected failures. A self-test function was added to the software to check the most critical component of the pill counter, the counter circuitry. Before each new batch of pills, the function flashes the LEDs on and off, and checks the phototransistors for a response. If there is no response due to a failure or an obstruction blocking the light from the LEDs

to the phototransistors, then a message appears on the display alerting the user to the problem. Considering that the lens of the LEDs and phototransistors may become smudged or dusty over time, the diagnostics also serves as a means to ensure that a clean path is maintained, preserving the accuracy of the device.

Another modification made from the PC version of the software was to write a new user interface to utilize the keypad and display. A user-oriented interface was created to replace the interface designed for the development engineers. For example, upon power-up, the display now asks the user for a quantity of pills to count rather than displaying a menu. Further description of how the software interacts with the user is presented in Appendix B.

A special repeat mode was added to allow for different counts of the same pill. In this mode, once the desired count has been reached, the pills remain in the cone and drum instead of being transferred into the overflow bin. A pharmacist would be able to administer 5000 pills into the drum and set it to count out sets of 200. Bulk containers of vitamins, for example, could be filled using this feature, without constant attendance during each count.

A flowchart showing the major tasks and decisions of the software is shown in Figure 4-3. A listing of the C code operating the pill counter with the 68HC11EVB is found in Appendix A.

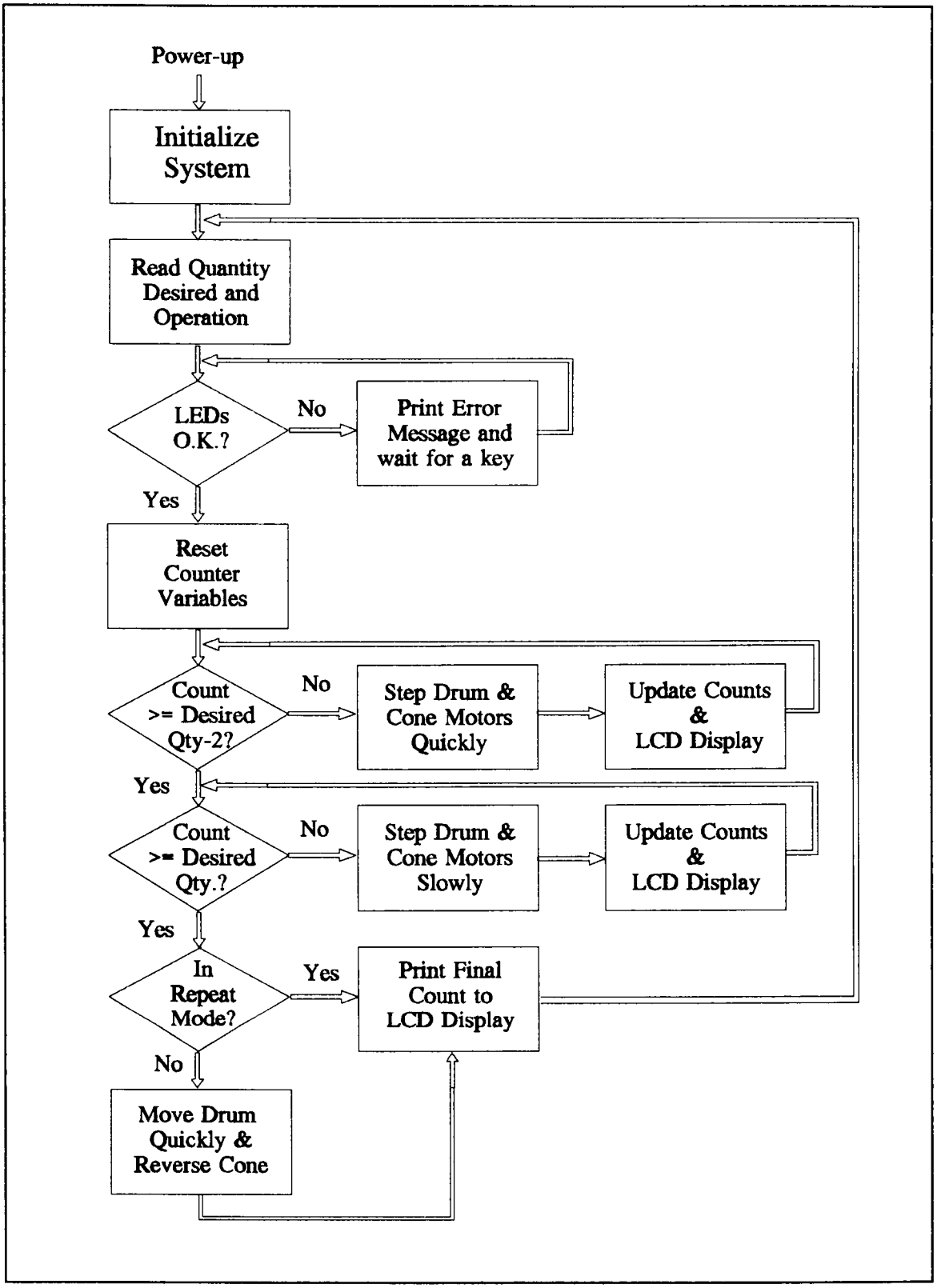


Figure 4-3: Pill counter operational flowchart (68HC11 Version).

CHAPTER 5

5.0 Conclusion

The pill counter described in this thesis is an apparatus that counts pills by putting them in a single-file line and passing them through optical counters. Microprocessor-based electronics were intentionally used instead of conventional solid-state electronics to give the system the greatest flexibility during development and operation. After testing the conceptual design, a prototype of the pill counter was built. This prototype was used to test the correctness of the electronics, while also providing a platform for the development of the mechanical system.

The accuracy of the machine relies upon the ability to separate the pills into a single-file stream as they pass through the counters. This separation is created by the mechanical system which has not fully evolved. With this in mind, some special considerations in the development of the electronic system were taken into account. The software was written to allow future designers to adjust the operational characteristics of the mechanical system through the keypad. Some of the characteristics include:

- the various speeds of the drum,
- the speeds and vibration modes of the cone, and
- the length of time to spend emptying pills into the overflow bin.

By adjusting these parameters through the keypad, a mechanical system can be "tuned" to provide the furthest separation without any software editing or compilation.

With this kind of flexibility, the electronics could easily be connected to another mechanical system that would count items other that pills or tablets. As long as the system uses a stepper motor drive system and the designed optical means of counting, no modifications in the electronics or software are needed. Also, different types of counters could be developed using other drive systems by modifying the existing interfaces and/or software on the pill counter.

Chapter 3 described the use of the 68HC11EVB evaluation board to control the pill counter. Since this was only meant as a prototype, the next logical step towards production is to consolidate the electronics into one circuit board. This can be done by replacing the evaluation board with the 68HC11. A memory chip is not needed if a mask is made for the built-in ROM. This mask can be made by Motorola by sending them the compiled software. The 68HC11s they return can then take the place of the large evaluation board, providing a more compact unit.

If the cost of having a mask made on the 68HC11 (approximately $2500) is prohibitive, then an external ROM can be used to store the program. Unfortunately, the communication lines between the 68HC11 and the ROM need to use some of the ports already in use by the pill counter. Motorola has addressed this problem by creating a Port Replacement Unit (PRU) that allows an external memory chip to be used and still have all of the ports on the 68HC11 available. By using the combination of a PRU and

a ROM, the initial expense of having a mask made is saved. However, this increases the individual cost of each pill counter by adding these extra chips.

Another option to avoid having a mask made is to use a special version of the 68HC11 called the 68HC11F1. The 68HC11F1 has extra ports available for interfacing to a memory chip. This larger 68-pin chip is slightly more expensive than the original 68HC11, but the difference is less than the cost of the PRU. Using an EPROM for the memory on this system allows for easy revisions and updates of the software either before or after production. This option has been implemented as a continuation of this thesis work and has proved to work as intended.

List of References

LIST OF REFERENCES

[1] David J. Bak, "Optical Feedback Calibrates Tablet Counter," *Design News*, 3 Sept. 90, pp. 112-113.

[2] United States Patent Number 4,013,192. Inventor: Austin E. Pillon. Assignee: ITL Industries, Inc.

[3] United States Patent Number 4,580,280. Inventor: C. Douglas Hetrick. Assignee: Bobar Instruments, Inc.

[4] United States Patent Number 3,837,139. Inventor: Harry G. Roseberg.

[5] Motorola Inc., *M68HC11 Reference Manual* (Englewood Cliffs, New Jersey: 1989), pp. 1:1-2.

[6] Philips Corp., *Technical Publication on Character Type LCD Modules*, p. 3.

[7] Lowell Jay Arthur, *Software Evolution* (New York: John Wiley & Sons, Inc., 1988), pp. 3-5.

[8] Michael W. Evans, *The Software Factory: A Fourth Generation Software Engineering Environment* (New York: John Wiley & Sons, Inc., 1989), p. 283.

[9] Ronald L. Krutz, *Interfacing Techniques in Digital Design with Emphasis on Microprocessors* (New York: John Wiley & Sons, Inc., 1988), pp. 252-253.

APPENDICES

APPENDIX A

Software C Code

```
/*****************************************************************************
*  File kbd.c -- contains mostly functions concerned with reading the key
*  pad.  Also includes function init_ports() which initializes the ports on
*  the 68HC11 microcontroller.
*
*  Last modified:        6-13-90
*  Modified by:          John C. Wilson
*/

#include <stdio.h>
#include <register.h>
#include "keys.h"                    /* file where key pad constants are defined */

/* Here are the function in keys.c */
void init_ports();
int key_hit();
int reset_hit();
int key_scan();
int get_entry();

#define INTBUFSIZE 4                 /* buffer size for number entry */

extern num_pills;

int waiting;                         /* if we are waiting for key to be released */


/*****************************************************************************
*   Function key_hit returns a boolean value which tells if a key on the
*   key-pad is being pressed.
*   Returns:  TRUE if any key is being pressed
*             FALSE if no key is being pressed
*/
int key_hit()  {

   PORTC = 0x00;                     /* pull outputs low */

   if ((PORTC & 0x0F) == 0x0F)
      return(FALSE);                 /* no key pressed */
   else
      return(TRUE);                  /* key pressed */
}
```

```
/*****************************************************************************
*   Function reset_hit returns a boolean value which tells if the reset key
*   on the key-pad is being pressed.
*   Returns:  TRUE if the reset key is being pressed
*             FALSE if the reset key is not being pressed
*/
int reset_hit()  {

  PORTC = 0xE0;                         /* output 1110 */

  if ((PORTC & 0x0F) == 0x07)
    return(TRUE);                       /* reset key pressed */
  else
    return(FALSE);                      /* reset key not pressed */
}

/*****************************************************************************
*   Function key_scan returns the value of any new key being pressed.
*   If there is no new key being pressed, the value returned is NO_KEY.
*   The other values returned are:
*         key                   value
*
*         0-9             the integer value of the key
*           strt          START
*           repeat        REPEAT
*           reset         RESET
*           hold          HOLD
*           small         SMALL
*           repeat small  REPEAT_SMALL
*
*   These constants are defined in the file keys.h
*/
int key_scan()  {

  unsigned char key;                    /* value of key pressed */

  if (key_hit() == FALSE)  {            /* if no key is being pressed */
    waiting = FALSE;                    /* no longer waiting */
    return(NO_KEY);
  } else                                /* a key is being pressed */
    if (waiting == TRUE)                /* waiting for old key to be released */
      return(NO_KEY);

   /* here if a new key has been pressed */
  waiting = TRUE;                       /* must wait till key is released */
```

```
PORTC = 0xE0;                    /* output 1110 */
  switch (PORTC & 0x0F) {
   case 0x07 :
      key = RESET;
      break;
   case 0x0b :
      key = 0;
      break;
   case 0x0d :
      key = HOLD;
      break;
   case 0x0e :
      key = REPEAT_SMALL;
  }
PORTC = 0xD0;                    /* output 1101 */
  switch (PORTC & 0x0F) {
   case 0x07 :
      key = 7;
      break;
   case 0x0b :
      key = 8;
      break;
   case 0x0d :
      key = 9;
      break;
   case 0x0e :
      key = REPEAT;
  }
PORTC = 0xB0;                    /* output 1011 */
  switch (PORTC & 0x0F) {
   case 0x07 :
      key = 4;
      break;
   case 0x0b :
      key = 5;
      break;
   case 0x0d :
      key = 6;
      break;
   case 0x0e :
      key = SMALL;
  }
```

```
  PORTC = 0x70;                          /* output 0111 */
    switch (PORTC & 0x0F)  {
     case 0x07 :
        key = 1;
        break;
     case 0x0b :
        key = 2;
        break;
     case 0x0d :
        key = 3;
        break;
     case 0x0e :
        key = START;
    }
  PORTC = 0xF0;                          /* reset outputs to high */
  return(key);

}   /* end key_scan() */
```

```
/*************************************************************************
*  Function get_entry returns the mode of operation the user has selected.
*  The number of pills entered by the user is put into the global variable
*  num_pills.
*/
int get_entry()  {

   char buf[INTBUFSIZE];              /* buffer to hold number */
   int key;                           /* value of key pressed */
   int i = 0;

   disp_entry();                      /* Place cursor for printing */
   itob(num_pills,buf,10);            /* init buf to num_pills */
   disp_print(buf);                   /* print value of num_pills */
```

```
  for (key = key_scan(); 0 == 0; key = key_scan()) {
    switch(key) {
    case 0: case 1: case 2: case 3: case 4:
    case 5: case 6: case 7: case 8: case 9:
      disp_entry();                          /* position cursor */
      if (i < INTBUFSIZE-1) {                /* if room */
        buf[i++] = key + '0';                /* convert to ASCII */
        buf[i] = '\0';                       /* insert null at end */
      }
      disp_print(buf);                       /* print new number */
      break;
    case RESET:
      disp_entry();                          /* position cursor */
      buf[0] = '0';                          /* reset string to zero */
      buf[1] = '\0';
      i = 0;                                 /* reset counter */
      disp_print(buf);
      break;
    case START:
      num_pills = atoi(buf);          /* set num_pills */
      while (key_hit() == TRUE)
        /* wait till key is released */ ;
      return(START);
    case REPEAT:
      num_pills = atoi(buf);          /* set num_pills */
      while (key_hit() == TRUE)
        /* wait till key is released */ ;
      return(REPEAT);
    case SMALL:
      num_pills = atoi(buf);          /* set num_pills */
      while (key_hit() == TRUE)
        /* wait till key is released */ ;
      return(SMALL);
    case REPEAT_SMALL:
      num_pills = atoi(buf);          /* set num_pills */
      while (key_hit() == TRUE)
        /* wait till key is released */ ;
      return(REPEAT_SMALL);
    default:
      break;
    }
  }
}   /* end get_entry */
```

```
        /* File pcc_cnt.c -- Contains all functions relating to the counter */

#include <stdio.h>
#include <register.h>
#include "pcc_cntl.h"

#define PAOVFMASK 0x20        /* mask location of Pulse Accum. overflow flag */

extern num_pills;         /* number of pills to count */

int tot_cnt1;             /* total count of lower counter */
int tot_cnt2;             /* total count of upper counter */
int last_tot_cnt;         /* highest total count */
tinyint last_cnt1;        /* last state of counter chip (lower) */
tinyint last_cnt2;        /* last state of counter chip (upper) */

tinyint cnt2;             /* present state of counter chips */
int tot_cnt;              /* highest preset count */
tinyint num_ovfl;         /* number of Pulse Accumulator overflows */

/*************************************************************************
*  function reset_count resets the counters chips and all software counters
*  in preparation to count another set of pills
*/
void reset_count()  {

   PACNT = 0;                          /* reset event counter */
   tot_cnt1 = 0;
   tot_cnt2 = 0;
   last_cnt1 = PACNT;
   last_cnt2 = 0;
   last_tot_cnt = 0;
   num_ovfl = 0;
}

/* inc_counter is a interrupt service routine that is called when the lower
*  photodetector is blocked.  The photodetector is connected through an
*  inverter to IC1 of PORTA.  When IC1 detects a falling edge, it generates
*  an interrupt.  The service routine increments the software counter (
*  (tot_cnt2), clears the interrupt, and returns.
*/
void inc_counter()  {

   TFLG1 = 0x04;          /* Clear interrupt */
   tot_cnt2++;            /* increments bottom counter */
}
```

```
/*****************************************************************************
*   function print_count displays the total # pills counted on the second
*   line of the LCD display
*/
void print_count()  {

   char buffer[17];                /* character buffer for text */
   char num_buf[5];                /* character buffer for number pills */

   strcpy(buffer,"Counted ");
   itob(tot_cnt,num_buf,10);       /* convert integer to string (number pills) */
   strcat(buffer,num_buf);         /* append number of pills counted */
   clear_two();                    /* move cursor to head of line two */
   disp_print(buffer);             /* print "Counted XXX" */

}

/*****************************************************************************
*   function check_leds tests the counting circuitry to see if it is operating
*   correctly.  The function turns the LEDs on and then back off again, then checks
*   the counters to see if they were incremented.  The function returns:
*         0:  No error detected
*         1:  Error in top LED
*         2:  Error in bottom LED
*   The LEDs are turned off at the end of this function
*/
int check_leds()  {

   last_cnt1 = PACNT;                  /* find present state of event counter*/
   last_cnt2 = tot_cnt2;               /* find present state of counter 2 */

   PORTD = 0x39;                       /* flashes LED's */
   my_delay(10);
   PORTD = 0x3D;
   my_delay(40);

         /* now check if counters were incremented */
   if (PACNT == last_cnt1)
      return(1);                       /* error in LED 1 */
   if (tot_cnt2 == last_cnt2)
      return(2);                       /* error in LED 2 */
   else
      return(0);                       /* no errors */
}
```

```
void check_op()  {

   int i;
   char buffer[16];

   for (i = 0; (i < 80) || (check_leds() != 0); i++)  {
      if (check_leds() == 1) {
         strcpy(buffer,"Check top LED ");
         clear_one();
         disp_print(buffer);
         strcpy(buffer,"& press reset ");
         clear_two();
         disp_print(buffer);
         while(key_hit() == FALSE) ;        /* wait for key to be hit */
         while(key_hit() == TRUE) ;         /* wait for key to be released */
      }
      if (check_leds() == 2) {
         strcpy(buffer,"Check exit LED");
         clear_one();
         disp_print(buffer);
         strcpy(buffer,"& press reset ");
         clear_two();
         disp_print(buffer);
         while(key_hit() == FALSE) ;        /* wait for key to be hit */
         while(key_hit() == TRUE) ;         /* wait for key to be released */
      }
   }
}
```

```
/*****************************************************************************
*  function update_cnt first reads in the values of the counter chips from
*  the printer port.  It compares them with last_cnt1 and last_cnt2 to see
*  if there has been a change since they were last read.  If there has been
*  a change then the appropriate total count variable is incremented.
*  Update_cnt then compares the highest count with num_pills to see if the
*  cone should be slowed down or stopped.  If the count is within 2 of the
*  number of pills to be counted then update_cnt returns SLOW.  If the count
*  is greater than or equal to the number counted, then update_cnt returns
*  STOP, otherwise it returns RUN.
*/
int update_cnt()  {

  char buffer[4];

  if (PACNT != last_cnt1)  {
    if ((TFLG2 & PAOVFMASK) != 0)   {        /* if pulse accum. overflow */
      num_ovfl++;
      TFLG2 = PAOVFMASK;                     /* clear P. A. overflow flag */
    }
    tot_cnt1 = ((int)num_ovfl << 8) + PACNT;
    last_cnt1 = PACNT;
  }
  if (tot_cnt1 > tot_cnt2)
    tot_cnt = tot_cnt1;
  else
    tot_cnt = tot_cnt2;

  if (last_tot_cnt != tot_cnt)  {
    itob(tot_cnt,buffer,10);                 /* convert to string */
    line_two();
    disp_print(buffer);                      /* print updated count */
    last_tot_cnt = tot_cnt;
  }

  if (tot_cnt < (num_pills-2))
    return(RUN);
  else if (tot_cnt >= num_pills)
    return(STOP);
  else  {
    if (tot_cnt1 > tot_cnt2)
      tot_cnt2 = tot_cnt1;
    return(SLOW);
  }
}
```

```
        /* File keys.h */

#define NO_KEY 10
#define RESET 11
#define REPEAT 12
#define START 13
#define HOLD 14
#define SMALL 15
#define REPEAT_SMALL 16
```

```
/*****************************************************************************
*  File disp.c -- contains mostly functions concerned with controling the LCD
*  display.  Also includes function my_delay() which is a generic delay loop
*  and delay_ten() which delays precisely 10 milliseconds.
*
*  Last modified:          4-26-90
*  Modified by:            John C. Wilson
*/

#include <stdio.h>
#include <register.h>

/* Here are the functions in disp.c */
void my_delay(int time);  /* generic delay loop */
void delay_ten();                       /* delays precisely 10 milliseconds */
void disp_print(char *ptr);             /* prints a string to the LCD display */
void disp_entry();                      /* move cursor to 14 pos. of line one */
void line_one();                        /* move cursor to line one of the display */
void line_two();                        /* move cursor to line two of the display */
void init_disp();                       /* initialize the display */
void clear_one();                       /* clear line one of the display */
void clear_two();                       /* clear line two of the display */
void clear_disp();                      /* clear both lines of the display */

/*****************************************************************************
*  Function my_delay counts up to "time" and returns.  Its purpose is to
*  waste time.
*/
void my_delay(int time)  {

   int i;

   for (i = 0; i < time; i++);          /* delay for time */
}

/*****************************************************************************
*  function delay_ten() uses up 10ms of time.  It is exited precisely 10ms after
*  it was entered.  Its main use is for a 10ms delay when programming the
*  on-chip EEPROM.  */
void delay_ten() {

   TOC1 = TCNT + 20000;                 /* load compare reg corres. to 10ms */
   TFLG1 = 0x80;                        /* clear timer flag one */
   while ((TFLG1 & 0x80) == 0x00)       /* wait till timer flag one is set */
      ;             /* do nothing */
}
```

```
/*****************************************************************************
*  Function init_disp initializes the LCD display after power-up.
*/
void init_disp()  {

   PORTD = 0x3D;
   PORTB = 0x38;          /* set for 8-bit data transfer */
   delay_ten();           /* delay 10ms */
   PORTB = 0x38;
   my_delay(300);

   PORTB = 0x0c;          /* turn on display and cursor */
   my_delay(40);

   PORTB = 0x06;          /* set to write across display */
   my_delay(40);

   PORTB = 0x01;          /* reset and clear display */
   delay_ten();           /* delay ten milliseconds */
}

/*****************************************************************************
*  Function disp_entry moves the cursor to the 14th position on line one and
*  clears to the end of the line.  This is the position where the count of
*  pills is usually displayed.
*/
void disp_entry()  {

   char buf[4];                  /* buffer to use for clearing to end of line */

   PORTB = 0x8d;                 /* set cursor to 14th position on first line*/

   strcpy(buf,"   ");            /* store blanks for clearing to end of line */
   disp_print(buf);              /* clear to end of line */

   PORTB = 0x8d;                 /* set cursor to 14th position on first line*/
}

/*****************************************************************************
*  Function line_one moves the cursor to the home position on line one.
*/
void line_one()  {

   PORTB = 0x80;                 /* set cursor to first line*/
   my_delay(10);
}
```

```
/*****************************************************************************
*  Function line_two moves the cursor to the home position on line two.
*/
void line_two()  {

  PORTB = 0xc0;                 /* set cursor to second line*/
}

/*****************************************************************************
*  Function clear_one clears line number one and puts the cursor at the home
*  position.
*/
void clear_one()  {

  char buf[17];                 /* buffer for clearing to end of line */

  line_one();                           /* move cursor to line one */
  strcpy(buf,"                ");       /* store buffer for clearing */
  disp_print(buf);                      /* clear line one */
  line_one();                           /* return cursor to home position */
}

/*****************************************************************************
*  Function clear_two clears line number two and puts the cursor at the home
*  position (of line number two).
*/
void clear_two()  {

  char buf[17];                 /* buffer for clearing to end of line */

  line_two();                           /* move cursor to line two */
  strcpy(buf,"                ");       /* store buffer for clearing */
  disp_print(buf);                      /* clear line two */
  line_two();                           /* return cursor to home position */
}

/*****************************************************************************
*  Function clear_disp clears the display and puts the cursor at the home
*  position of line one.
*/
void clear_disp()  {

  clear_two();
  clear_one();
}
```

```
/*****************************************************************************
*  Function disp_print prints the character string pointed to by ptr, out to
*  the LCD display starting at the current cursor position.  ptr must be a
*  null terminated string.
*/
void disp_print(char * ptr)  {

  PORTD = PORTD | 0x02;       /* set RS bit */
  while (*ptr != '\0')  {
    PORTB = *ptr;             /* output character to PORT B  */
    ptr++;                    /* next char */
    my_delay(1);              /* must delay slightly or outrun display */
  }
  PORTD = PORTD & 0xFD;       /* reset RS bit */
}
```

```
        /* File PCC.H -- Program begins here */

        /* these variables had to be global since they */
        /* are all stored in EEPROM (except num_pills) */

int lg_fd_time = 17;        /* drum "fast mode" - delay between motor steps */
int lg_sd_time = 28;        /* drum "slow mode" - delay between motor steps */
int sm_fd_time = 25;        /* drum "fast mode" - for small pills */
int sm_sd_time = 36;        /* drum "slow mode" - for small pills */
int lg_vc_time = 40;        /* cone: delay between on cycles - for large pills */
int sm_vc_time = 40;        /* cone: delay between on cycles - for small pills */
int vc_onsteps = 5;         /* number of steps cone is on during vibration */
int vc_offsteps = 2;        /* number of steps cone is off during vibration */
int fd_steps = 50;          /* number of drum steps in fast mode */
int norm_rev_steps = 150;/* number of steps to reverse cone after
                                    pills have been counted */
int c_slow_time = 90;       /* time between steps when almost finished counting */
int d_slow_time = 90;       /* time between steps when almost finished counting */
int c_rev_time = 45;        /* time between cone steps when reversing */
int d_rev_time = 45;        /* time between drum steps when reversing */

near int num_pills;         /* number of pills to count */


/*****************************************************************************
* function main runs the pill counter.  Upon power-up, everything is
*  initialized, and run_ppc() is executed till power-down. */
main()  {

   init_ports();            /* initialize ports for reading and writing */
   init_disp();             /* initialize display after power-up */
   init_cntl();             /* initialize control parameters */
   clear_disp();
   while(0 == 0)  {
      run_ppc();
   }
}
```

```
        /* File PCC_CNTL.C -- controls the overall operation of the pill counter */

#include <stdio.h>
#include "pcc_cntl.h"
#include "keys.h"
#include "register.h"

        /* these variables are all integers and are stored in EEPROM */
extern far lg_fd_time;      /* drum "fast mode" - delay between motor steps */
extern far lg_sd_time;      /* drum "slow mode" - delay between motor steps */
extern far sm_fd_time;      /* drum "fast mode" - for small pills */
extern far sm_sd_time;      /* drum "slow mode" - for small pills */
extern far lg_vc_time;      /* cone: delay between on cycles - for large pills */
extern far sm_vc_time;      /* cone: delay between on cycles - for small pills */
extern far vc_onsteps;      /* number of steps cone is on during vibration */
extern far vc_offsteps;     /* number of steps cone is off during vibration */
extern far fd_steps;        /* number of drum steps in fast mode */
extern far norm_rev_steps;/* number of steps to reverse cone after
                             pills have been counted */
extern far c_slow_time;     /* time between cone steps when almost done counting */
extern far d_slow_time;     /* time between drum steps when almost done counting */
extern far c_rev_time;      /* time between cone steps when reversing */
extern far d_rev_time;      /* time between drum steps when reversing */
extern far transition_pause;/* pause between RUN mode & SLOW mode */

        /* this variable is stored in RAM (that's why it is near) */
extern near num_pills;      /* number of pills to count */

        /* these variables are LOCAL to this module */
tinyint fd_time;            /* fast drum time (local to module) */
tinyint sd_time;            /* slow drum time (local to module) */
tinyint vc_time;            /* cone delay between on cycles */
int rev_steps;                      /* number of reverse steps for clearing pills */
tinyint d_ramp[RAMP_STEPS+1]; /* array of delay values to allow drum
                                 to ramp from slow to fast mode */
unsigned char c_mtr_state[4];
unsigned char d_mtr_state[4];
tinyint c_state;                    /* cone motor drivers state */
tinyint d_state;                    /* drum motor drivers state */
tinyint c_wait;                     /* counters */
tinyint d_wait;
tinyint cone_mode;                  /* cone motor state (ON or OFF) */
tinyint off_step_count;             /* cone counter */
tinyint on_step_count;              /* cone counter */
```

```
/*****************************************************************************
*  function init_cntl initializes all variables needed after power-up.  */
void init_cntl()  {

   init_vars(c_mtr_state, d_mtr_state);
   num_pills = 0;
   c_wait = 0;
   d_wait = 0;
   cone_mode = ON;
   on_step_count = 0;

}

/*****************************************************************************
*  function step writes out the present state of software steps to the
*  hardware through port A.  It was created in order to conserve ROM space
*  (the C statement compiles into 17 assembly statements), since this
*  statement is executed quite often. */
void step()  {
   PORTA = c_mtr_state[c_state] | d_mtr_state[d_state];
}

void step_cone(int direction)  {

   if (c_wait >= vc_time)  {                    /* has vc_time elapsed? */
      if (cone_mode == ON)  {                   /* do we step this time? */
         if (on_step_count >= vc_onsteps)  {    /* have stepped vc_onsteps? */
            off_step_count = 0;
            cone_mode = OFF;                    /* don't step next time */
         }
         else  {                            /* we will step now */
               if (direction == FORWARD)
               c_state = ++c_state & 0x03;  /* compute next forward step */
               else
                  c_state = --c_state & 0x03;  /* compute next reverse step */
            step();                         /* step cone motor */
            ++on_step_count;
         }
      }
      else  {                               /* cone_mode == OFF */
         if (off_step_count >= vc_offsteps)  {
            on_step_count = 0;
            cone_mode = ON;
         }
         else
            off_step_count++;               /* artificial step */
      }
      c_wait = 0;                           /* reset cone wait variable */
   } else
      ++c_wait;                             /* wait again, don't step this time */
}
```

```
/*****************************************************************************
*   function falling controls the pill counter while the pills are falling
*   from the drum into the cone.  At this time, the drum is turning "slow"
*   and the cone is vibrating.  The procedure is exited when the drum
*   detector senses the foil.
*/
void falling()  {

  tinyint d_time;
  tinyint check_kbd = 0;

  d_time = sd_time;

  while ((PORTA & DRUM_MASK) != FOIL_PRESENT)  {
    step_cone(FORWARD);

    if (d_wait >= d_time)  {
      d_state = ++d_state & 0x03;          /* figure next step */
      step();                              /* steps drum motor */
        d_wait = 0;
    }
    else
        ++d_wait;

    if (++check_kbd >= 170)  {   /* check for user interrupt */
      check_kbd = 0;
      if (reset_hit() == TRUE)
        return;
    }
    if (update_cnt() != RUN)
      return;

  }               /* end while loop */
}                 /* end function falling() */
```

```
/*****************************************************************************
*   Function not_falling controls the pill counter while pills are not
*   falling from the drum into the cone.  At this time, the drum is turning
*   "fast"  and the cone is vibrating.  The procedure is exited after
*   the drum has taken fd_step steps.
*/
int not_falling()  {

  tinyint d_step_count = 0;
  tinyint d_time;
  tinyint i;

  d_time = d_ramp[0];              /* initialize with first value of accel ramp */


    /* drum will accelerate to a faster speed in this while loop.  */
  while (d_step_count < fd_steps)  {
    step_cone(FORWARD);

    if (d_wait >= d_time)  {
      d_state = ++d_state & 0x03;          /* figure next step */
      step();                              /* step drum motor */
      ++d_step_count;
      d_wait = 0;
      if (d_step_count < RAMP_STEPS)
        d_time = d_ramp[d_step_count];
      else
        d_time = d_ramp[RAMP_STEPS];
    }
    else
      ++d_wait;

    ++i;                    /* equilize speeds of falling and not_falling */

    if (update_cnt() != RUN)
      return(SLOW);
  }               /* end while */
  return(RUN);
}
```

```
/*****************************************************************************
*  Function backwards clears the drum and cone of pills.  It runs the drum
*  forward and the cone backwards 10*"rev_steps" times. */
void backwards()  {

  int i;                              /* counter */
  int num_rev_steps;                        /* number of reverse steps */

  num_rev_steps = rev_steps*10;  /* increase number of reverse steps */

  for (i = 0; (i < num_rev_steps) && (reset_hit() != TRUE); )  {
    step_cone(REVERSE);
    if (d_wait >= d_rev_time)  {  /* check if time to step drum */
      d_state = ++d_state & 0x03;         /* figure next drum step */
      step();                             /* step drum motor */
      d_wait = 0;                         /* reset drum counter */
      i++;                                /* add one to count */
    }
    else
      ++d_wait;
    update_cnt();
  }      /* end for loop */
}

/*****************************************************************************
*   Function finish counts out the last few pills while rotating the cone and
*   drum slowly.  Once update_cnt() returns STOP (pills have been counted),
*   function backwards is called.
*/
void finish()  {

  unsigned char enable[8];          /* enable array (used for half step) */
  unsigned char half_step_phase[8];
  int i;

  enable[0] = 0x05;
  enable[1] = 0x25;
  enable[2] = 0x05;
  enable[3] = 0x15;
  enable[4] = 0x05;
  enable[5] = 0x25;
  enable[6] = 0x05;
  enable[7] = 0x15;
  half_step_phase[0] = 0x00;
  half_step_phase[1] = 0x40;
  half_step_phase[2] = 0x40;
  half_step_phase[3] = 0x60;
  half_step_phase[4] = 0x60;
  half_step_phase[5] = 0x20;
  half_step_phase[6] = 0x20;
  half_step_phase[7] = 0x00;
```

```
                /* stop cone and drum for "transition_pause" * 10ms */
                /* in order to stop pills before slowing down cone */
  for (i = 0; i < transition_pause; i++)  {
    delay_ten();                          /* delay 10ms */
    update_cnt();
  }

        /* move cone and drum slowly till last pill comes through */
  while (update_cnt() != STOP)  {
    if (c_wait >= c_slow_time)  {                 /* check if time to step cone */
      c_state = ++c_state & 0x07;                 /* figure next cone step */
      PORTA = half_step_phase[c_state] | d_mtr_state[d_state];
      PORTD = enable[c_state];                /* step cone motor */
      c_wait = 0;                                 /* reset cone counter */
    }
    else
      ++c_wait;
    if (d_wait >= d_slow_time)  {                 /* check if time to step drum */
      d_state = ++d_state & 0x03;                 /* figure next drum step */
      PORTA = half_step_phase[c_state] | d_mtr_state[d_state];
                                                  /* step drum motor */
      d_wait = 0;                                 /* reset drum counter */
    }
    else
      ++d_wait;

    if (reset_hit() == TRUE)                  /* check to see if user interupt */
      return;

  }      /* end while */

  c_state = 0;                                /* reset cone state */
  PORTD = ENABLE;                             /* ensure that the motors are enabled */
  backwards();                                /* clear pills */
}
```

```
/*****************************************************************************
*  function set_params will set the run-time parameters depending upon which
*  mode the user has selected.  Calculates the ramp delays for drum motor.*/
void set_params(mode)
tinyint mode;  {

   tinyint i;
   unsigned int accel,denom,last;

   if ((mode == SMALL) || (mode == REPEAT_SMALL))  {          /* if small pill mode */
      fd_time = sm_fd_time;                                   /* set fast drum speed */
      sd_time = sm_sd_time;                                   /* set slow drum speed */
      vc_time = sm_vc_time;                                   /* set cone speed */
   }
   else  {                                          /* set variables for large pills */
      fd_time = lg_fd_time;                                   /* set fast drum speed */
      sd_time = lg_sd_time;                                   /* set slow drum speed */
      vc_time = lg_vc_time;                                   /* set cone speed */
   }

   accel = (unsigned int) (160*(sd_time - fd_time))/ RAMP_STEPS;
   d_ramp[0] = sd_time;
   last = sd_time;
   for (i = 1; i < RAMP_STEPS; i++)  {
      denom = 800 + (10*accel/sd_time*last/fd_time*last/(sd_time+fd_time));
      last = (800*last)/denom;
      d_ramp[i] = last;
   }
   d_ramp[RAMP_STEPS] = fd_time;

}
```

```
/*****************************************************************************
* function run_ppc starts and controls operation of the pill counter.
*/
void run_ppc()  {

  char buffer[17];                    /* character buffer */
  char num_buf[5];

  tinyint state = RUN;
  tinyint mode;
  void reset_count();                 /* declare function to be called */

  clear_one();
  strcpy(buffer,"Enter Number ");
  disp_print(buffer);
  mode = get_entry();
  if (num_pills == 987)  {
    change_params();
    return(FINISH);
  }

  clear_one();                  /* clear line one of display */
  strcpy(buffer,"Checking...");
  disp_print(buffer);           /* print "Checking" */
  check_op();           /* check out operations */

  set_params(mode);             /* set up parameters dependent upon mode */

  if ((mode == REPEAT) || (mode == REPEAT_SMALL))  {        /* if repeat mode */
    rev_steps = 0;                                          /* no reversing  */
    strcpy(buffer,"Repeat to:   ");
  } else  {
    rev_steps = norm_rev_steps;         /* repeat in nomal mode */
    strcpy(buffer,"Counting to: ");
  }

  reset_count();        /* reset counters */
  clear_one();                  /* clear line one of the display */
  itob(num_pills,num_buf,10);   /* convert integer to string (number pills) */
  strcat(buffer,num_buf); /* append number of pills to count */
  disp_print(buffer);           /* print "Counting to:" number of pills */

  strcpy(buffer,"0   Counting...");
  line_two();                   /* move cursor to head of line two */
  disp_print(buffer);           /* print "0  Counting..." */

  PORTD = ENABLE;                           /* enables motors */
```

```
        /************ RUNS HERE *********/
  while ((reset_hit() == FALSE) && (state == RUN))  {
    falling();
    state = not_falling();
  }

  if (state != RUN)
    finish();

  print_count();                    /* print # pills actually counted */

  if ((mode == START) || (mode == SMALL))       /* if not repeat mode */
    num_pills = 0;                              /* reset num_pills to 0 */

  PORTD = DISABLE;                  /* disable motors to reduce heat */

}
```

```
/* File pcc_cntl.h -- Include file to pcc_cntl.c */

#define FOIL_PRESENT 0x00    /* Value read in if drum detector sees foil */
#define DRUM_MASK 0x01       /* Masks out all bits except bit 0 */
#define CNT_MASK 0x78        /* Masks out bits 0,1,2,7 */
#define ENABLE 0x05          /* Enables motors for running */
#define DISABLE 0x3D         /* Disables motors to reduce heat */

#define ON 1                 /* cone motor to step when ON */
#define OFF 0                /* cone motor not to step when OFF */

#define STOP 0
#define RUN 1
#define SLOW 2

#define RAMP_STEPS 7

#define PASS 1
#define FAILURE 0

#define FINISH 1
#define NOT_FINISH 0

#define FORWARD 1
#define REVERSE 0

typedef unsigned char tinyint;
```

```
        /* File INIT.C -- contains functions to initialize hardware ports, variables upon
         *                each new count, and functions to work with the EEPROM */

#include "pcc_cntl.h"
#include "register.h"

        /* these variables are all integers and are stored in EEPROM */
extern far lg_fd_time;      /* drum "fast mode" - delay between motor steps */
extern far lg_sd_time;      /* drum "slow mode" - delay between motor steps */
extern far sm_fd_time;      /* drum "fast mode" - for small pills */
extern far sm_sd_time;      /* drum "slow mode" - for small pills */
extern far lg_vc_time;      /* cone delay for large pills */
extern far sm_vc_time;      /* cone delay for small pills */
extern far vc_onsteps;      /* number of steps cone is on during vibration */
extern far vc_offsteps;     /* number of steps cone is off during vibration */
extern far fd_steps;        /* number of drum steps in fast mode */
extern far norm_rev_steps;          /* number of steps to reverse cone after
                                       pills have been counted */
extern far c_slow_time;     /* time between steps when almost finished counting */
extern far d_slow_time;     /* time between steps when almost finished counting */
extern far c_rev_time;      /* time between cone steps when reversing */
extern far d_rev_time;      /* time between drum steps when reversing */

void init_ports();
void init_eeprom();
void init_vars();
void ee_erase();
void ee_store();

/*************************************************************************
*   Function init_ports initializes the 68HC11 for running the pill counter.
*   It is meant to be called once upon power-up.  Returns nothing.  */
void init_ports()  {

   tinyint *ptr;

   ptr = 0x4000;            /* these 2 lines only for Evaluation board */
   *ptr = 0x00;             /* switch RXD pin from monitor to display */

   SPCR = 0x04;             /* turn off serial port */
   SCCR2 = 0x00;            /* turn off serial comm. on port D */

;
#asm
   SEI
#endasm
   HPRIO = 0x08;            /* set input capture 1 as highest priority inter. */
;
#asm
   CLI
#endasm
```

```
  TFLG1 = 0x04;        /* clear interrupt flag */
  TCTL2 = 0x20;        /* capture on falling edges */
  TMSK1 = 0x04;        /* counting interrupt request enabled */
  PACTL = 0x40;        /* set bit 7 of port A as input to Pulse accumulator */
                       /* Set to event counter and to count on falling edges */

  PORTA = 0x00;        /* Motors starting point */
  DDRC = 0xF0;         /* 0-3 inputs, 4-7 outputs (for keyboard) */
  PORTC = 0xF0;        /* initialize keyboard outputs high */
  DDRD = 0x3F;         /* set bits 0-5 for output (bits 0-1 for display), */
                       /* bit 2 used as output to LEDs, 3-5 motor enable */
  PORTD = 0x3D;        /* LED's on and Motors disabled */

  PIOC = 0x02;         /* Select simple strobe active low for STRB */
}


/*****************************************************************************
*  function init_vars initializes the two arrays that control the operation of
*  the cone stepper motor driver and the drum stepper motor driver. Also
*  initializes the EEPROM by calling init_eeprom(). */
void init_vars(c_mtr_state, d_mtr_state)
unsigned char *c_mtr_state, *d_mtr_state; {

  c_mtr_state[0] = 0x00;
  c_mtr_state[1] = 0x40;
  c_mtr_state[2] = 0x60;
  c_mtr_state[3] = 0x20;

  d_mtr_state[0] = 0x00;
  d_mtr_state[1] = 0x10;
  d_mtr_state[2] = 0x18;
  d_mtr_state[3] = 0x08;

  init_eeprom();
}

/*****************************************************************************
* function ee_erase erases one byte of EEPROM at location "addr" */
void ee_erase(addr)
unsigned char *addr;  {

  PPROG = 0x16;              /* set to BYTE erase mode */
  *addr = 10;                /* write any data to EEPROM address */
  PPROG = 0x17;              /* set EEPGM bit -- start erasing */
  delay_ten();               /* delay 10ms */
  PPROG = 0x00;              /* turn off high voltage & set to read mode */

}
```

```
/*****************************************************************************
*  function ee_store stores an integer (2 bytes) "data" into EEPROM at
*  location "addr".  "data" is first split into its high and low bytes since
*  the EEPROM can only be programmed one byte at a time.
*  A character pointer, "byte_addr", is introduced and set equal to the
*  int pointer "addr" so when incremented, the compiler will only add one
*  to "byte_addr" (as opposed to adding two to the int pointer).  The low
*  and high bytes are erased separately, and then programmed to be equal
*  to "data". */
void ee_store(addr, data)
int *addr, data;  {

   unsigned char lsb_byte, msb_byte;      /* least and most significant byte */
   unsigned char *byte_addr;              /* pointer for byte addressing */

   lsb_byte = data;                       /* The msb of data is discarded */
   msb_byte = (data >> 8);                /* load msb of data */
   byte_addr = (char *) addr;             /* set byte_addr to point to data */

   ee_erase(byte_addr);             /* erase msb at byte_addr */
   ee_erase(byte_addr+1);           /* erase lsb at byte_addr */

   PPROG = 0x02;                    /* set EELAT bit -- ready to program */
   *byte_addr = msb_byte;           /* store MSB byte to EEPROM address */
   PPROG = 0x03;                    /* set EEPGM bit -- start programming */
   delay_ten();                     /* delay 10ms */
   PPROG = 0x02;                    /* turn off high voltage */
   *(byte_addr+1) = lsb_byte;       /* store LSB byte to EEPROM address */
   PPROG = 0x03;                    /* set EEPGM bit -- start programming */
   delay_ten();                     /* delay 10ms */
   PPROG = 0x00;                    /* turn off high voltage & set to read mode */

}
```

```
/*****************************************************************************
*  function init_eeprom checks an arbitrary variable stored in EEPROM
*  to see if it has been either erased or never programmed (its value would
*  0xFFFF).  If the EEPROM needs to be reprogrammed, then it does so,
*  otherwise the function exits */
void init_eeprom()  {

   if (lg_fd_time == 0xFFFF)  {    /* if eeprom has been erased */
      ee_store(&lg_fd_time,17);
      ee_store(&lg_sd_time,28);
      ee_store(&sm_fd_time,25);
      ee_store(&sm_sd_time,36);
      ee_store(&fd_steps,50);
      ee_store(&vc_onsteps,5);
      ee_store(&vc_offsteps,2);
      ee_store(&lg_vc_time,40);
      ee_store(&sm_vc_time,45);
      ee_store(&c_slow_time,90);
      ee_store(&d_slow_time,150);
      ee_store(&norm_rev_steps,150);
      ee_store(&c_rev_time,45);
      ee_store(&d_rev_time,35);
   }
}
```

```
        /* File pcc_user.c -- Contains code to read input from the user */

#include "keys.h"

extern far lg_fd_time;
extern far lg_sd_time;
extern far sm_fd_time;
extern far sm_sd_time;
extern far vc_onsteps;
extern far vc_offsteps;
extern far lg_vc_time;
extern far sm_vc_time;
extern far fd_steps;
extern far norm_rev_steps;
extern far c_slow_time;
extern far d_slow_time;
extern far c_rev_time;
extern far d_rev_time;
extern num_pills;

void cc_store();

/*****************************************************************************
*  function update interprets the user input for changing a variable and
*  returns the appropriate value.  If the user enters a new value, then
*  update returns that value.  If the user does not enter a value by
*  just pressing the enter key, then update returns the old value which
*  is variable "old_var"
*/
int update(int old_var) {

   int new_var;

   num_pills = old_var;
   get_entry();
   if (num_pills == old_var) {   /* if nothing was entered */
      new_var = old_var;
   }
   else                          /* else something was entered */
      new_var = num_pills;
   return(new_var);
}
```

```
/*****************************************************************************
*  function change_params prints a brief description of each variable
*  and its current value.  change_params calls update to allow the user
*  to optionally change the value of the variable.
*/
void change_params(){

   char buf1[17];

   clear_disp();
   strcpy(buf1,"Changing params");
   line_two();
   disp_print(buf1);
   strcpy(buf1,"lg_fd_time");
   clear_one();
   disp_print(buf1);
   ee_store(&lg_fd_time, update(lg_fd_time));

   strcpy(buf1,"lg_sd_time");
   clear_one();
   disp_print(buf1);
   ee_store(&lg_sd_time, update(lg_sd_time));

   strcpy(buf1,"sm_fd_time");
   clear_one();
   disp_print(buf1);
   ee_store(&sm_fd_time, update(sm_fd_time));

   strcpy(buf1,"sm_sd_time ");
   clear_one();
   disp_print(buf1);
   ee_store(&sm_sd_time, update(sm_sd_time));

   strcpy(buf1,"fd_steps ");
   clear_one();
   disp_print(buf1);
   ee_store(&fd_steps, update(fd_steps));

   strcpy(buf1,"onsteps ");
   clear_one();
   disp_print(buf1);
   ee_store(&vc_onsteps, update(vc_onsteps));

   strcpy(buf1,"offsteps ");
   clear_one();
   disp_print(buf1);
   ee_store(&vc_offsteps, update(vc_offsteps));
```

```
  strcpy(buf1,"lg_vc_time ");
  clear_one();
  disp_print(buf1);
  ee_store(&lg_vc_time, update(lg_vc_time));

  strcpy(buf1,"sm_vc_time ");
  clear_one();
  disp_print(buf1);
  ee_store(&sm_vc_time, update(sm_vc_time));

  strcpy(buf1,"rev_steps/10 ");
  clear_one();
  disp_print(buf1);
  ee_store(&norm_rev_steps, update(norm_rev_steps));

  strcpy(buf1,"c_rev_time ");
  clear_one();
  disp_print(buf1);
  ee_store(&c_rev_time, update(c_rev_time));

  strcpy(buf1,"d_rev_time ");
  clear_one();
  disp_print(buf1);
  ee_store(&d_rev_time, update(d_rev_time));

  strcpy(buf1,"c_slow_time ");
  clear_one();
  disp_print(buf1);
  ee_store(&c_slow_time, update(c_slow_time));

  strcpy(buf1,"d_slow_time ");
  clear_one();
  disp_print(buf1);
  ee_store(&d_slow_time, update(d_slow_time));

}
```

```
        /* File register.h -- Contains all memory mapped ports */
/*****************************************************************/
/* "Non PAGE 0" INTERNAL REGISTER BLOCK - BYTE and WORD EXTERNS  */
/*****************************************************************/

extern far unsigned char

/*****  0:8      1:9      2:A      3:B      4:C      5:D      6:E   7:E **/

        PORTA,                  PIOC,   PORTC,  PORTB,  PORTCL,       DDRC,
        PORTD,          DDRD,           PORTE,  CFORC,  OC1M,   OC1D,
        TCTL1,  TCTL2,  TMSK1,  TFLG1,  TMSK2,  TFLG2,  PACTL,  PACNT,
        SPCR,   SPSR,   SPDR,   BAUD,   SCCR1,  SCCR2,  SCSR,   SCDR,
        ADCTL,          ADR1,   ADR2,   ADR3,   ADR4,
        OPTION, COPRST, PPROG,  HPRIO,  INIT,   TEST1,  CONFIG;

extern far unsigned int
                                                        TCNT,
            TIC1,           TIC2,           TIC3,       TOC1,
            TOC2,           TOC3,           TOC4,       TOC5;

/*****************************************************************/
/*              INTERNAL REGISTER BIT ADDRESS DEFINES            */
/*****************************************************************/
extern far struct {
        unsigned int _7 : 1;
        unsigned int _6 : 1;
        unsigned int _5 : 1;
        unsigned int _4 : 1;
        unsigned int _3 : 1;
        unsigned int _2 : 1;
        unsigned int _1 : 1;
        unsigned int _0 : 1;
        } _INIT;

/* INIT (RAM and I/O Mapping Register) bits */
#define RAM3    _INIT._0    /* Si03D.7    RAM3 - RAM0 = RAM map position */
#define RAM2    _INIT._1    /* Si03D.6    */
#define RAM1    _INIT._2    /* Si03D.5    */
#define RAM0    _INIT._3    /* Si03D.4    */
#define REG3    _INIT._4    /* Si03D.3    REG3 - REG0 = 64-byte register */
#define REG2    _INIT._5    /* Si03D.2                  block position   */
#define REG1    _INIT._6    /* Si03D.1    */
#define REG0    _INIT._7    /* Si03D.0    */
```

```
        /* File stdio.h */

/*      STDIO.H         Include file for C88 input/output.  */

/*      a 'FILE' is simply an integer in this implementation    */
typedef int  FILE;

/*      Standard input, standard output and standard error.  */

#define stdin     0
#define stdout    1
#define stderr    2

#define NULL    0
#define TRUE    1
#define FALSE   0
#define EOF     -1
#define ERR     -1
```

APPENDIX B

Operational Description

Once the compiled program "pcc.h" has been down-loaded into the 68HC11EVB, the BUFFALO command "G C000" is entered from the personal computer. All commands to the pill counter are now given from the keypad. The LCD display now reads:

Enter Number 0

showing that the pill counter is waiting for a value. A greater number of pills is deposited into the drum. The user enters the quantity of pills desired and then presses a mode select key for counting to begin. These keys along with their actions are described in the following table.

Key	Action
START	Counts to the entered quantity and then deposits all extra pills into the overflow bin.
REPEAT	Counts to the entered quantity and stops.
LARGE PILL START	Same as START but the cone and drum move faster for more expedient counting.
LARGE PILL REPEAT	Same as REPEAT but the cone and drum move faster for more expedient counting.

Once one of these buttons is pressed, the system performs a self-test by checking the photoelectric counters. If everything is satisfactory, the drum and cone go into motion. The drum rotates smoothly allowing the pills to fall onto the cone approximately one at

a time. To increase the separation of the pills, the cone vibrates while rotating. The pills flow around the edge of the cone and into the chute where they are counted. The display shows the up-to-date quantity:

Counting to: 75
28 Counting...

or

Repeat to: 75
28 Counting...

in the repeat mode of operation.

Once the counted quantity reaches to within two of the entered quantity, the drum and cone slow down to count the final few pills. If the user had selected repeat mode, the drum and cone will stop. In a normal mode, the drum rotates quickly and the cone rotates in reverse so that extra pills are deposited into the overflow bin. The display now shows that the pill counter is ready to operate again:

Enter Number: 0
Counted 75

or

Enter Number: 75
Counted 75

following the repeat mode.

VITA

John Clement Wilson was born August 17, 1967 in Lexington, Kentucky to Clement and Charlotte Wilson. When John was five, his family moved to Boulder, Colorado which quickly became his claimed home town. There, he attended elementary, junior high, and high schools, graduating in the top 15% of his class at Boulder high. John's proudest accomplishment as a youth was the attainment of his Eagle Scout rank at the age of 16.

John entered the University of Colorado at Boulder in September 1985 to pursue a B.S. degree in electrical and computer engineering. Throughout his undergraduate education, he was employed part-time at companies such as Data Display, IBM, and Storage Technology. Scouting remained a large part of John's life with active involvement in Troop 70 as an Assistant Scoutmaster. John was awarded the B.S. degree in Electrical and Computer Engineering in December, 1989.