University of Florida

Department of Electrical and Computer Engineering

EEL 5666

Intelligent Machines Design Laboratory

Written Report 3: “Beverly Crusher”

Jean-François A. Kamath

12/08/03

Table of Contents **Page**

Abstract

Beverly Crusher has seen great progress and several key objectives were reached. While she does not have the capacity to drive around searching for cans due to problems with the power supply and time constraints, the rest of the system performs admirably. The core of the project was the crusher system, which works extremely well, far better than I had expected. Beverly is able to detect the presence of soda cans, pick them up, crush them, and clear the crusher reliably. Overall this project was a success since the primary goal of crushing cans was accomplished.

Executive Summary

Beverly Crusher is a robot that can detect the presence of a can, pick it up, and crush it. The original design called for her to actively seek out cans in her environment, but this proved to be problematic and difficult for several reasons. The first limiter on Beverly's performance was the power supply; quite simply, the Ni-MH battery packs that were being used could not handle the current requirements of the entire system, which will be discussed shortly. The second limiter was the sensitivity of the IR sensors and their tendency to give wildly fluctuating readings. This problem was only corrected a couple of days before the final demonstration and could not be effectively incorporated into the system.

The crushing system shows excellent performance with the ability to easily and consistently pick up the cans and crush them with no effort. The cans are detected using a break sensor constructed from a CDS-cell and an LED. This seems to work quite well even in bright lighting, requiring only minor calibration depending on the environment. The crusher is a screw press capable of exerting nearly 500 lbs of force on whatever is placed into it. The U-shaped frame anchors the power screw and holds all of the pieces together. An aluminum plate serves to crush the cans and is attached to both the power screw as well as three slider rods which keep the plate lined up. A motor capable of exerting 70 oz-in of torque is run through a 30:1 gear ratio to exert the crushing force.

The collector and clearer mechanisms are made of wood and are controlled by three servos. The collector grips the cans with a moveable arm and then flips them upside down into the crusher. The clearer arm is used to push crushed cans out of the crusher to prepare it for the next cycle.

Introduction

Beverly Crusher will be a robot designed to locate discarded soda cans, crush them, and dispose of them in a receptacle that she carries. She will need to navigate a simple environment that will consist of several bounding walls as well as the soda cans. A major focus of this project will be design and construction of the retrieval mechanism and the can crusher. This paper will cover the ideas and designs in development of this robot. The basic designs for the frame and various mechanisms will be discussed followed by a description of the actuators and sensors that may be used in development. The behavior of the robot will also be covered.

Integrated System

The robot will consist of several main systems: the can retriever, the crusher, dispenser, sensor array, and drive system. The sensor array will be used for many tasks including wall avoidance, can location, and crusher maintenance. Beverly needs to avoid hitting walls, partly to prevent damage to various parts, but also to continue her search for cans. Sensors will be used to locate cans and determine when they are in a position to be picked up and crushed.

Mobile Platform

The platform must be able to navigate a somewhat restricted environment while carrying potentially heavy retrieval and crushing apparatuses. It may also be required to withstand high stresses during the crushing process, so materials other than balsa wood may become necessary. The design of the platform is still being worked on since several possibilities exist for the arrangement. However, the designs for the crusher and retriever are nearing completion and should allow the structure of the platform to be finalized soon.

Actuation

Two kinds of actuators will be used, servos for positioning of the gripper and for movement of the platform, and a motor with gearing system for the crusher. Hacked servos are being used to move the platform since high speed is of minimal concern. Three servos will be required to operate the collector and clearer: one will open and close the "hand" and one will raise the can and flip it over. The third servo will be used to move the clearer arm to prepare the crusher for the next can. The servos used for actuation of the collector and clearer are Hitec HS-311 standards. The drive servos are Hitec HS-805 ¼ scale giant servos that provide an enormous 300 oz-in of torque. These can be purchased at www.servocity.com.

To control the servos, a pulsed signal of variable length must be implemented. All of the servos require a pulse in the range of 0.9 to 2.1 ms followed by approximately 20 ms dead time. The length of the pulse determines the position the servo will move to, or in the case of the hacked drive servos, how fast and in what direction they will turn. Since implementing PWM would have been difficult considering my limited programming background, I instead wrote a function that controls all of the servos simultaneously. It could easily be modified to handle up to 64 servos, the maximum number of pins on the processor board.

The crusher will use a single motor geared down by 30:1. A worm gear is being used due to the simplicity of setting up the gearing. Using standard spur gears would have required a far more complex system and significantly higher costs, possibly as much as an extra $200. A pair of relays is used to control the motor which is on a separate circuit from the processor and sensors. This avoids the problems of power spikes and feedback in the system. The crush plate will be moved by a ½ in diameter power screw that is turned by the worm gear. Crush time will be approximately 24 sec at 20 V and the crusher will zero itself at approximately 8 in above the base plate.

Sensors

The sensory system, used by Beverly to locate and distinguish cans, consists of six IR range sensors, a break sensor, and a bump switch. These will be used to detect potential targets, home in on them, and determine if the object is acceptable for crushing. The IR sensors are Sharp GP2D12 and can be purchased at www.junun.org for $8.25 each.

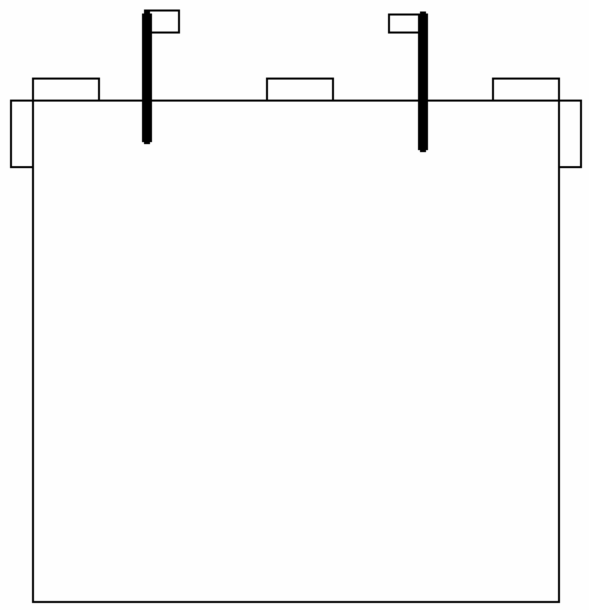


Figure S-1. Sensor layout of Beverly Crusher.

As shown in Figure S-1, the robot will utilize five IR range sensors for obstacle detection. The left and right-hand sensors will be used primarily for obstacle avoidance, such as walls and other large objects. The three, front-facing sensors are necessary for target location and obstacle avoidance. Depending on the relative readings of the sensors, Beverly will be able to distinguish between walls, concave corners, and potential cans. Table S-1 shows what different combinations of readings represent.

Relative IR Readings			Meaning
Left	Mid	Right	
M	M	M	Wall
H	M	L	Wall
L	M	H	Wall
L	L	M	Can/Table leg to the left
L	M	M	Can/Table leg to the left
M	H	H	Can/Table leg to the right
M	M	H	Can/Table leg to the right
M	H	M	Can/Table leg in front
M	L	M	Concave corner

Table S-1. Meanings of sensory reading from front IR range sensors.

If Beverly determines there might be a can to pick up, she will turn until the object is directly in front of her. She will then drive forward until one of two events occurs: the break sensor indicates that an object is in the gripper, or the front bumper indicates that Beverly has run into something. In the latter case, she will have encountered either a table leg or a wall corner, and she will proceed to back away and seek a new route. However, if the break sensor goes off, Beverly will have found a can and will proceed to pick up and crush said can. The break sensor was constructed from a CDS-cell and an LED, which were pseudo collimated using heat shrink tubing. These parts can be purchased at any electronics store.

The sixth IR sensor that is not shown in Figure S-1 is used to determine the position of the crush plate. The crush plate will always be zeroed at the same distance above the platform using the IR sensor and will be operated for an experimentally determined period of time while crushing the cans.

Behaviors

Beverly will have a relatively simple search algorithm. She will wander an environment while avoiding walls until she detects a can. She will then orient herself such that the can is within the gripper's effective area, grab it, place it in the crusher, crush it, and finally clear the crusher. Sensors to detect the presence of crushed cans in the crusher were deemed unnecessary since it is automatically cleared each time. Crimping was also eliminated from the final design for a couple of reasons. The first and most important reason was that designing the collector to crimp the cans would have been a far more complicated task than simply picking them up. The second issue was that if the can was crimped improperly, then it would not line up in the crusher and might slide out of the crusher. Ultimately, the crusher was designed to apply far more force than would be necessary to crush a non-crimped can.

Experimental Layout and Results

Several tests had to be performed throughout the course of this project, including sensors, actuators, the motor, and obstacle avoidance.

IR Sensors

Testing of the IR sensors and the break sensor were performed and showed excellent results. The IR sensors follow the sensitivity indicated in the documentation provided by the supplier. Figure Exp-1 shows the sensitivity graph.

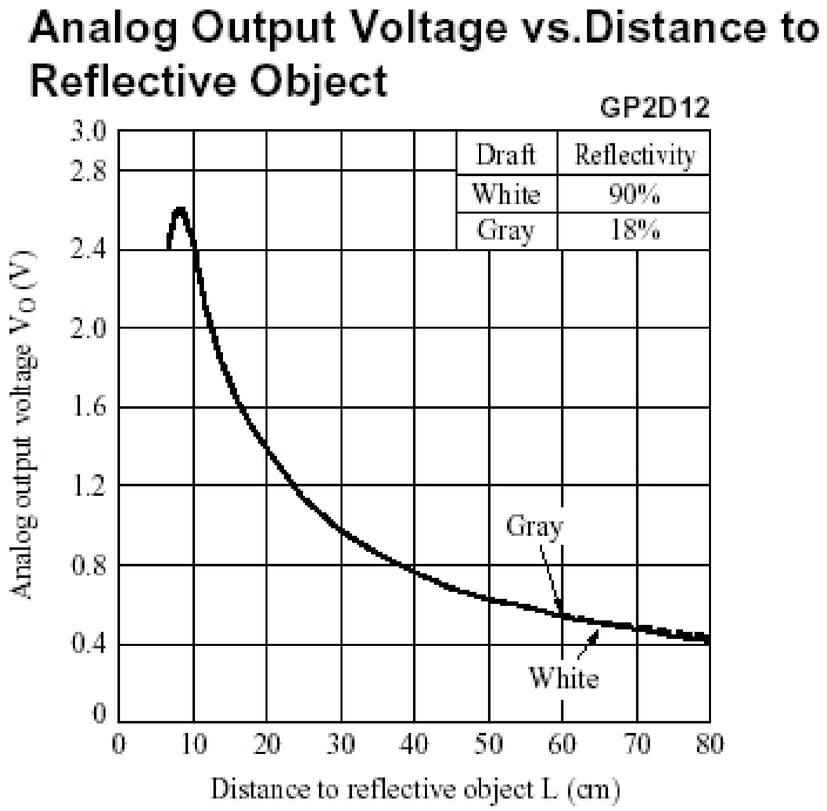


Figure Exp-1. IR range sensor readings vs Distance.

Experimentation showed that the peak reading from these sensors is in the range of 130-140 depending on exactly which sensor is being used as well as the lighting conditions. Testing has shown that in the range of 10-30 cm, the readings could be approximated as linear. This simplified the search algorithm since Beverly only needed to actively seek out objects near to her.

The actual testing procedure consisted of placing various objects in front of the sensors at ranges from 1 in to 10 in to see what readings could be obtained. The sensors performed equally regardless of the objects used. One issue that came up while working with the IR was that the readings tended to fluctuate wildly when a large number of sensors and actuators were connected to the system. At first, this caused extreme problems when trying to compare readings for obstacle avoidance and target recognition. This was corrected only a couple of days ago when a suggestion was made on how to "smooth" the signals. By averaging the readings from a sensor over a certain period of time, the noise could be reduced. Testing revealed that averaging the sensory readings over a 300 ms period completely eliminated the fluctuations in the readings and allowed me to sense differences as small as 1%. Experimentation showed that the difference in readings between the edge of a can and its vertical midline were on the order of 5%, indicating that the sensors were sensitive enough for the task.

Actuators

The servos that move the collector and the clearer behaved slightly differently from the specified performance. Rather than a pulse width of 0.9-2.1 ms for full left and right respectively, they required pulses of 0.6-2.2 ms. This may have been due to the time required to build the signal or possibly the control function that I designed. With the correct numbers input into the system, the collector and clearer performed perfectly during every test. The collector had the capacity to grip cans that were off position by as much as an inch side to side.

Crusher

The crusher required more testing than the collector and clearer due to its inherent complexity. The first experiment was designed to determine how well the parts lined up. The most critical pieces were the power screw and slider rods since any significant misalignment could lead to jamming of the system. The motor was controlled manually since it could be stopped quickly if needed. However, this proved to be unnecessary as the alignment of the system was excellent, showing virtually no resistance to motion when properly lubricated.

The second test was used to determine the minimum voltage requirements of the crusher. Three power supplies were tested: 9.6 V Ni-MH, 12 V lead acid, 19.2 V Ni-MH. In all three situations, the system crushed the cans with little effort. The only slowdown was experienced when the crush plate first made contact with the can using the 9.6 V power supply since the stall force of the system was very close to the yield point of the cans. However, within seconds the cans collapsed and were subsequently crushed. Ultimately, the 19.2 V power pack was selected since it was easiest to incorporate into the system.

The final test measured the time required to crush the cans. This simply consisted of measuring how long the system took to move the crush plate from its zero position to a distance 1 in above the base plate. For the 12 V power supply, the crush time was 36 sec, while for the 19.2 V supply, crush time was only 24 sec.

Obstacle Avoidance

Basic obstacle avoidance was demonstrated earlier in the semester and was found to work decently for avoiding large surfaces. Code was designed to allow the system to perform better obstacle avoidance as well as distinguish between cans, posts, and walls. Unfortunately, this code could not be tested fully due to multiple catastrophic system failures. The first and most important is that both drive servos burned out simultaneously, though the reasons remain unknown. Since this occurred so close to the end of the project, new servos could not be acquired to replace them. Secondly and possibly related to the servo collapses, the battery packs could not handle the power requirements of the entire system. Since I did not have the knowledge necessary to be able to turn individual sensors and servos on and off, I had to leave everything except the crusher on continuously. This effectively placed a constant drain of nearly four amps on the system.

Despite the setbacks of the loss of servos and an inadequate power supply, several tests were made to determine sensor sensitivity. At first, the real-time readings from the sensors were displayed on the LCD so that they could be compared. This led to the discovery that the readings fluctuated far too much to be used effectively. However, a new method for obtaining sensor data was suggested that virtually eliminated the problem. This technique involved “smoothing” of the data, or averaging of the readings over certain periods of time. Testing revealed that sampling data over a 300 ms period removed any variability in the readings and brought the sensitivity of the system to 1%. To detect cans, a difference of only 5% is needed, indicating that the sensor system should perform quite well.

Conclusion

Overall this project was a success since it met several of the design goals. Namely, the system is able to detect the presence of a soda can, attain the can, and crush it. Obstacle avoidance was also accomplished to a limited extent. Considering that when the project began, I had no knowledge of processor programming, servo control, relay control, or sensory input and very limited electrical experience, this project turned out quite well. It also served as an invaluable learning experience since it brought to light the real complexities involved in designing and building a system of any real complexity.

Documentation

N/A

Appendices

A.1

<u>Main Program</u>

```
/*********************************************
This program was produced by the
CodeWizardAVR V1.23.9b Standard
Automatic Program Generator
© Copyright 1998-2003 HP InfoTech s.r.l.
http://www.hpinfotech.ro
e-mail:office@hpinfotech.ro

Project :
Version :
Date    : 12/3/2003
Author  : Jean Francois Kamath
Company : Hampton, FL 32044
Comments:


Chip type           : ATmega128
Program type        : Application
Clock frequency     : 16.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 1024
*********************************************/

#include <mega128.h>
#include <delay.h>
#include <math.h>
#include <string.h>
#include "inttostr.h"
#include "servocon.h"
#include "ultradrive.h"

#define REPEAT 1;

// Alphanumeric LCD Module functions
#asm
   .equ __lcd_port=0x18
#endasm
#include <lcd.h>

#define ADC_VREF_TYPE 0x60
// Read the 8 most significant bits
// of the AD conversion result
```

```
unsigned char read_adc(unsigned char adc_input)
{
ADMUX=adc_input|ADC_VREF_TYPE;
// Start the AD conversion
ADCSRA|=0x40;
// Wait for the AD conversion to complete
while ((ADCSRA & 0x10)==0);
ADCSRA|=0x10;
return ADCH;
}

// Declare your global variables here

void main(void)
{
// Declare your local variables here
int i;     // generic counter
int nPosition[8];
int nNumCycles;
int c;

//int nSensor;
//int nBreak;
int nCrushLim;

// Input/Output Ports initialization
// Port A initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T State6=T State7=T
PORTA=0x00;
DDRA=0x00;

// Port B initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T State6=T State7=T
PORTB=0x00;
DDRB=0x00;

// Port C initialization
// Func0=Out Func1=Out Func2=Out Func3=Out Func4=Out Func5=Out Func6=Out
Func7=Out
// State0=0 State1=0 State2=0 State3=0 State4=0 State5=0 State6=0 State7=0
PORTC=0x00;
DDRC=0xFF;

// Port D initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T State6=T State7=T
PORTD=0x00;
DDRD=0xFF;

// Port E initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T State6=T State7=T
PORTE=0x00;
DDRE=0x00;

// Port F initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In Func6=In Func7=In
// State0=T State1=T State2=T State3=T State4=T State5=T State6=T State7=T
PORTF=0x00;
DDRF=0x00;

// Port G initialization
// Func0=In Func1=In Func2=In Func3=In Func4=In
// State0=T State1=T State2=T State3=T State4=T
PORTG=0x00;
DDRG=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
// Mode: Normal top=FFh
// OC0 output: Disconnected
ASSR=0x00;
TCCR0=0x00;
TCNT0=0x00;
OCR0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: Timer 1 Stopped
// Mode: Normal top=FFFFh
// OC1A output: Discon.
// OC1B output: Discon.
// OC1C output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
```

```
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
OCR1CH=0x00;
OCR1CL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer 2 Stopped
// Mode: Normal top=FFh
// OC2 output: Disconnected
TCCR2=0x00;
TCNT2=0x00;
OCR2=0x00;

// Timer/Counter 3 initialization
// Clock source: System Clock
// Clock value: Timer 3 Stopped
// Mode: Normal top=FFFFh
// OC3A output: Discon.
// OC3B output: Discon.
// OC3C output: Discon.
TCCR3A=0x00;
TCCR3B=0x00;
TCNT3H=0x00;
TCNT3L=0x00;
OCR3AH=0x00;
OCR3AL=0x00;
OCR3BH=0x00;
OCR3BL=0x00;
OCR3CH=0x00;
OCR3CL=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
// INT2: Off
// INT3: Off
// INT4: Off
// INT5: Off
// INT6: Off
// INT7: Off
EICRA=0x00;
EICRB=0x00;
EIMSK=0x00;
```

```
// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;
ETIMSK=0x00;

// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
// Analog Comparator Output: Off
ACSR=0x80;
SFIOR=0x00;

// ADC initialization
// ADC Clock frequency: 125.000 kHz
// ADC Voltage Reference: AVCC pin
// ADC High Speed Mode: Off
// Only the 8 most significant bits of
// the AD conversion result are used
ADMUX=ADC_VREF_TYPE;
ADCSRA=0x87;
SFIOR&=0xEF;

// LCD module initialization
lcd_init(16);
int_to_str(read_adc(1));
lcd_puts(gstrConvNum);
delay_ms(2000);
lcd_clear();
lcd_putsf("Initializing");

// Initialize global variables.
gnCan = 0;
gnIRLimit = 120;
gnIRSens = 5;
gnLeftIR = 0;    //
gnFLIR = 0;      //
gnFIR = 0;       //
gnFRIR = 0;      //
gnRightIR = 0;   //
gnBump = 0;      //
gnRearB = 44;// check these
gnFrontB = 226;  // check these
gnBreak = 0;     //
gnBreakSens = 230;

nNumCycles = 0;
```

```
// Sensor limits
nCrushLim = 80;

// Initialize motor
delay_ms(500);
lcd_clear();
lcd_putsf("Motor Init");
while(read_adc(7) > nCrushLim)
{
        PORTD.3 = 1;
        PORTD.4 = 1;
        PORTD.5 = 1;
}
PORTD.3 = 0;
PORTD.4 = 0;
PORTD.5 = 0;

// Initialize servos.
delay_ms(500);
lcd_clear();
lcd_putsf("Servo Init");
nPosition[0] = 14;      // Gripper Base
nPosition[1] = 15;      // Gripper Arm
nPosition[2] = 22;      // Clearer Arm
nPosition[3] = 15;      // Left Servo
nPosition[4] = 14;      // Right Servo
nPosition[5] = 14;
nPosition[6] = 14;
nPosition[7] = 14;

DriveServos(nPosition, 20);

i = 0;

delay_ms(500);
lcd_clear();
lcd_putsf("Waiting");
while(read_adc(0) != gnRearB)
        DriveServos(nPosition, 1);
lcd_clear();
lcd_putsf("Starting Program");

// Main program loop.
while (1)
    {
```

```
gnCan = 0;

// Get sensor readings.
gnBump += read_adc(0);
gnBreak += read_adc(1);
gnLeftIR += read_adc(2);
gnFLIR += read_adc(3);
gnFIR += read_adc(4);
gnFRIR += read_adc(5);
gnRightIR += read_adc(6);

nNumCycles++;

// Check if we need to average the readings and update servo speeds.
if(nNumCycles == 14)
{
     gnBump /= nNumCycles;
     gnBreak /= nNumCycles;
     gnLeftIR /= nNumCycles;
     gnFLIR /= nNumCycles;
     gnFIR /= nNumCycles;
     gnFRIR /= nNumCycles;
     gnRightIR /= nNumCycles;

     DriveControl();
  nPosition[3] = gnLeftMotor;
  nPosition[1] = gnRightMotor;
  lcd_clear();
  int_to_str(gnLeftMotor);
  lcd_puts(gstrConvNum);
  lcd_gotoxy(0,1);
  int_to_str(gnRightMotor);
  lcd_puts(gstrConvNum);

     nNumCycles = 0;

     gnBump = nNumCycles;
     gnBreak = nNumCycles;
     gnLeftIR = nNumCycles;
     gnFLIR = nNumCycles;
     gnFIR = nNumCycles;
     gnFRIR = nNumCycles;
     gnRightIR = nNumCycles;
}

// Servo control.
```

```
DriveServos(nPosition, gnDriveTime);
// If motor times are larger than 1, reset sensors.
if(gnDriveTime > 1)
{
    gnDriveTime = 1;
}

// If there is a can, pick it up and crush it.
if(gnCan == 1)
{
lcd_gotoxy(0,0);
lcd_putsf("Can Detected");
  // initialize the servo positions
  nPosition[0] = 14;
    nPosition[1] = 6;
    nPosition[2] = 22;
    nPosition[3] = 14;
    nPosition[4] = 14;
    nPosition[5] = 14;
    nPosition[6] = 14;
    nPosition[7] = 14;

    DriveServos(nPosition, 20);

    delay_ms(2000);

  /**********************/
    // Pick up and crush can.
    /**********************/
    // Flip gripper down.
    lcd_clear();
    lcd_putsf("Getting can.");
    nPosition[0] = 6;  // gripper base
    DriveServos(nPosition, 20);

    // Close gripper arm.
    nPosition[1] = 19; // gripper arm
    DriveServos(nPosition, 20);

    // Flip gripper over.
    nPosition[0] = 22;  // gripper base
    DriveServos(nPosition, 40);

    // Open arm.
    nPosition[1] = 6; // gripper arm
    DriveServos(nPosition, 40);
    // Recenter gripper.
    nPosition[0] = 6;  // gripper base
    DriveServos(nPosition, 40);

    // Lower crusher
    lcd_clear();
    lcd_putsf("Crushing can.");
    // wait until sensor says crusher has lowered enough.
    for(c = 0; c < 3600; c++)
    {
            lcd_gotoxy(0,1);
            int_to_str(read_adc(1));
            lcd_puts(gstrConvNum);
            //          DriveServos(nPosition, 1);
            PORTD.0 = 1;
            PORTD.1 = 1;
            PORTD.2 = 1;
            delay_ms(10);
}
    // shut off crusher.
    PORTD.0 = 0;
    PORTD.1 = 0;
    PORTD.2 = 0;
    delay_ms(1000);

    // Raise crusher
    lcd_clear();
    lcd_putsf("Raising Crusher");
    delay_ms(1000);
    // raise crusher for predetermined interval.
    while(read_adc(7) > nCrushLim)
    {
    PORTD.3 = 1;
 PORTD.4 = 1;
    PORTD.5 = 1;
    }
    // shut off crusher.
    PORTD.3 = 0;
    PORTD.4 = 0;
    PORTD.5 = 0;

    // clear can
    lcd_clear();
    lcd_putsf("Clearing Crusher");
    nPosition[2] = 14; // clearer arm
```

```
            DriveServos(nPosition, 40);

            // rezero clearer
          nPosition[2] = 22;
            DriveServos(nPosition, 40);

        }
       };
}
```

<u>servocon.h</u>

```
// This file defines a basic function for controlling up to 8 servos
// on the AT Mega 128 board. Essentially crude PWM.
// Note: requires delay.h

// nPos determines how far the servos will turn.  If a motor is being
// controlled, set nPos[#] to a number greater than nMaxSignal
// nCycles is the number of cycles

void DriveServos(int nPos[8], int nCycles)
{
        int t;
        int nSignal;
        int nMaxSignal;         // Determines how long each pulse lasts.

        nMaxSignal = 221;

        // Cycle through the loop until nCycles cycles have passed.
        // Used to turn large distances w/out interference from sensors.
        for(t = 0; t < nCycles; t++)
        { // start for1()
                // Send drive signal to all servos.
                PORTC.0 = 1;
                PORTC.1 = 1;
                PORTC.2 = 1;
                PORTC.3 = 1;
                PORTC.4 = 1;
                PORTC.5 = 1;
                PORTC.6 = 1;
                PORTC.7 = 1;

                // Start PWM
                for(nSignal = 0; nSignal < nMaxSignal; nSignal++)
        { // start for3()
                        // Check each servo for stop.
                        // Shut off servo signal if pulse time has passed.
```

```
				if(nSignal == nPos[0])
				{
					PORTC.0 = 0;
				}
				if(nSignal == nPos[1])
				{
					PORTC.1 = 0;
				}
				if(nSignal == nPos[2])
				{
					PORTC.2 = 0;
				}
				if(nSignal == nPos[3])
				{
					PORTC.3 = 0;
				}

				if(nSignal == nPos[4])
				{
					PORTC.4 = 0;
				}
				if(nSignal == nPos[5])
				{
					PORTC.5 = 0;
				}

				if(nSignal == nPos[6])
				{
					PORTC.6 = 0;
				}
				if(nSignal == nPos[7])
				{
					PORTC.7 = 0;
				}

				// Wait 0.1 ms
				delay_us(100);
	} // end for3()
	   } // end for1()
}
```

<u>ultradrive.h</u>

```
// This header file defines the behavior that the robot will use to
// locate and avoid obstacles.

#define M_STOP		15
```

```
#define L_FORWARD 9
#define L_HALFFOR 14
#defineL_REVERSE 21
#define L_HALFREV 16
#define R_FORWARD 21
#define R_HALFFOR 16
#define R_REVERSE 9
#define R_HALFREV 14

int gnCan;

int gnLeftIR, gnFLIR, gnFIR, gnFRIR, gnRightIR;  // Sensor readings from front IR's
int gnIRLimit;                                   // Range limit
int gnIRSens;                                    // Limit on IR sensitivity

int gnBump;                                              // Bump switches
int gnRearB, gnFrontB;                           // Values for bump switches

int gnBreak;                                     // Break sensor reading
int gnBreakSens;                                 // Limit on break
int gnLeftMotor, gnRightMotor;            // Motor speeds
int gnDriveTime;        // Amount of time to drive servos at specified speed
                                          // This time is increments of 22.1 ms


// For the demo, the robot will use 3 sensors mounted on the left,
// right and front.  The rear will have 1 bump switch to activate,
// the robot and to detect a rear obstacle.
// The other switches will not be present and their values need
// to be set to 255.
void DriveControl(void)
{
        // Check if there is a can to be picked up
        if(gnBreak >= gnBreakSens)
        {
                gnLeftMotor = M_STOP;
                gnRightMotor = M_STOP;
                gnDriveTime = 1;

                // Indicate that a can needs to be crushed.
                gnCan = 1;
        }

        // Check if the robot has backed into something
        else if(gnBump == gnRearB)
        {
```

```
		gnLeftMotor = L_FORWARD;
		gnRightMotor = R_FORWARD;
		gnDriveTime = 25;
	}

	// Ran into post
	else if(gnBump >= gnFrontB)
	{
		gnLeftMotor = L_REVERSE;
		gnRightMotor = R_HALFREV;
		gnDriveTime = 25;
	}

	// Check if IR sensors indicate obstacle
	// Front IR
	else if(gnFLIR >= gnIRLimit ||
			gnFIR  >= gnIRLimit ||
			gnFRIR >= gnIRLimit)
	{
		// Check if can is possible
		// ^-o-^Can directly in front
		if( (gnFIR - gnFLIR) >= gnIRSens &&
			(gnFIR - gnFRIR) >= gnIRSens)
		{
			// Drive forward a little to see
			// if it is really a can
			gnLeftMotor = L_FORWARD;
			gnRightMotor = R_FORWARD;
			gnDriveTime = 1;
		}

		// ^-^-oCan to right
		else if((gnFRIR - gnFLIR) >= gnIRSens &&
				(gnFRIR - gnFIR) >= gnIRSens)
		{
			gnLeftMotor = L_FORWARD;
			gnRightMotor = R_REVERSE;
			gnDriveTime = 1;
		}

		// o-^-^Can to left
		else if((gnFLIR - gnFIR) >= gnIRSens &&
				(gnFLIR - gnFRIR) >= gnIRSens)
		{
			gnLeftMotor = L_REVERSE;
			gnRightMotor = R_FORWARD;
```

```
        gnDriveTime = 1;
}

// o-o-^Can to left
else if((gnFLIR - gnFRIR) >= gnIRSens &&
                (gnFIR - gnFRIR) >= gnIRSens)
{
        gnLeftMotor = L_REVERSE;
        gnRightMotor = R_FORWARD;
        gnDriveTime = 1;
}

// ^-o-oCan to right
else if((gnFIR - gnFLIR) >= gnIRSens &&
                (gnFRIR - gnFLIR) >= gnIRSens)
{
        gnLeftMotor = L_FORWARD;
        gnRightMotor = R_REVERSE;
        gnDriveTime = 1;
}

// ^-o-oCan to right
else if((gnFIR - gnFLIR) >= gnIRSens &&
                (gnFRIR - gnFLIR) >= gnIRSens)
{
        gnLeftMotor = L_FORWARD;
        gnRightMotor = R_REVERSE;
        gnDriveTime = 1;
}

// o-^-oCorner
else if((gnFLIR - gnFIR) >= gnIRSens &&
                (gnFRIR - gnFIR) >= gnIRSens)
{
        // Check for obstacles to sides
        if(gnRightIR >= gnIRLimit)
        {
                // Turn left
                gnLeftMotor = L_REVERSE;
                gnRightMotor = R_FORWARD;
                gnDriveTime = 12;
        }

        else
        {
                // Turn right
```

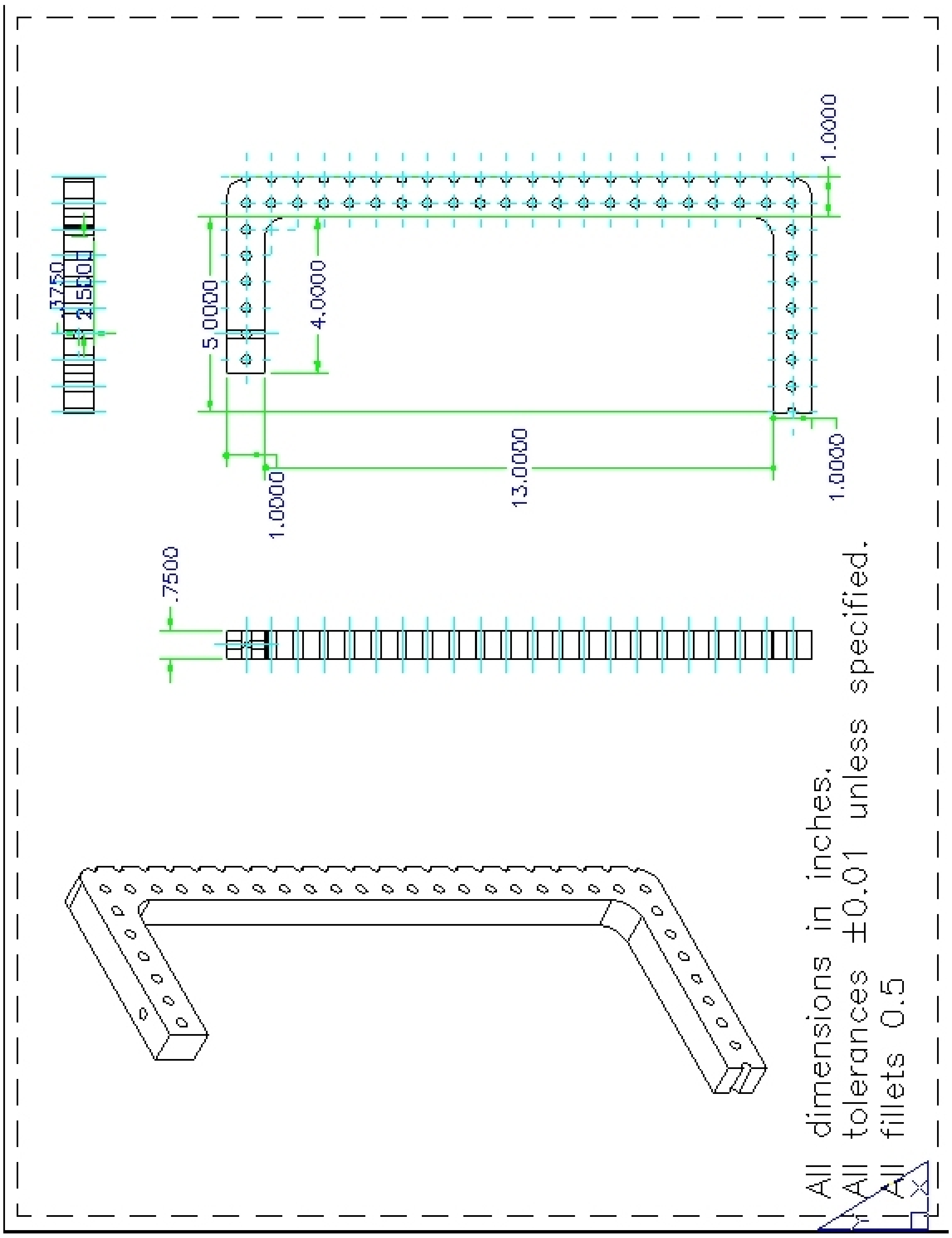
```
                                gnLeftMotor = L_FORWARD;
                                gnRightMotor = R_REVERSE;
                                gnDriveTime = 12;
                        }
                }

                // o-o-oWall
                else if(abs(gnFIR - gnFLIR) >= gnIRSens &&
                                abs(gnFIR - gnFRIR) >= gnIRSens)
                {
                        // Turn right
                        gnLeftMotor = L_FORWARD;
                        gnRightMotor = R_REVERSE;
                        gnDriveTime = 12;
                }
        }

        // Left IR
        // Wall to left
        else if(gnLeftIR >= gnIRLimit)
        {
                // Turn right a little
                gnLeftMotor = L_FORWARD;
                gnRightMotor = R_REVERSE;
                gnDriveTime = 12;
        }

        // Right IR
        // Wall to right
        else if(gnRightIR >= gnIRLimit)
        {
                // Turn left a little
                gnLeftMotor = L_REVERSE;
                gnRightMotor = R_FORWARD;
                gnDriveTime = 12;
        }

        else
        {
                // Default to straight forward
                gnLeftMotor = L_FORWARD;
                gnRightMotor = R_FORWARD;
                gnDriveTime = 1;
        }
}
```

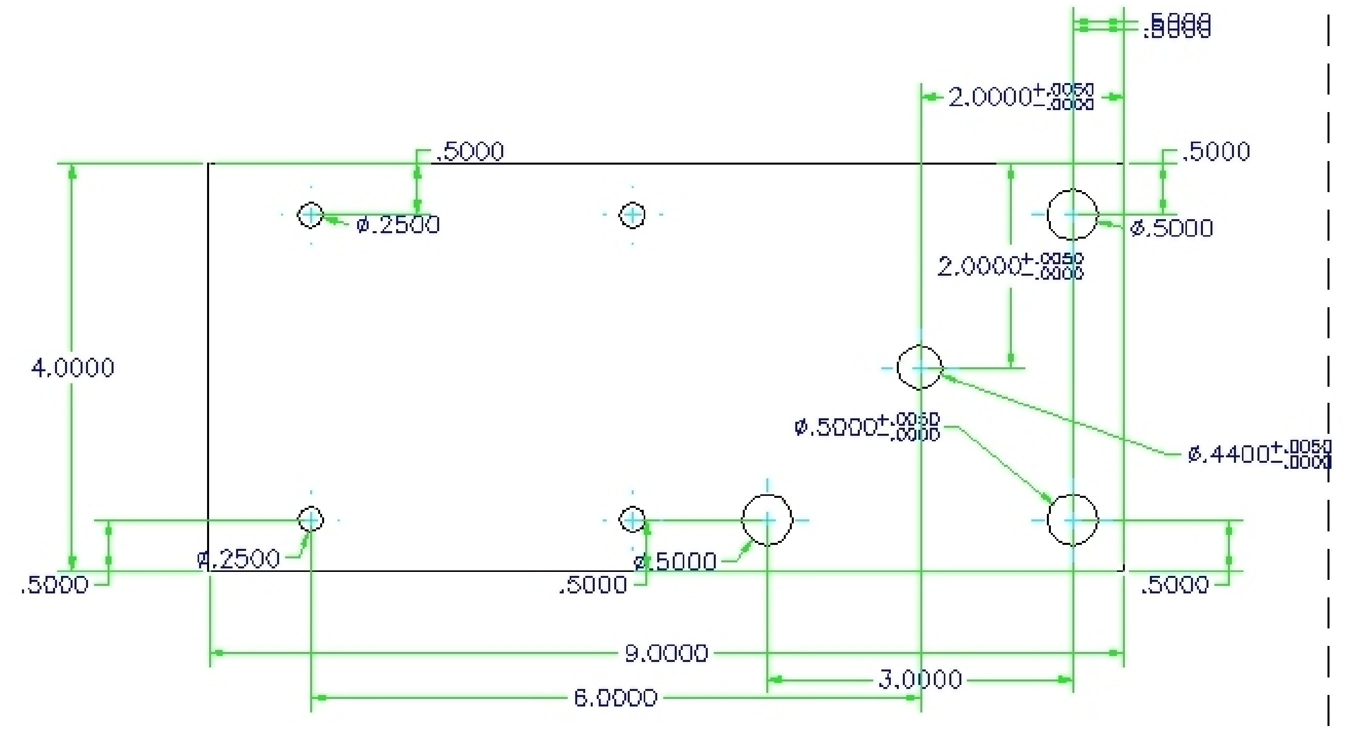
inttostr.h

```
// This function converts an integer into a string to be written to the LCD
char gstrConvNum[20];

char int_to_char(int num)
{
 char retval;
 num = abs(num);
 while(num >= 10)
 {
  num = num/10;
 }
 switch(num)
 {
  case 0: retval = '0'; break;
  case 1: retval = '1'; break;
  case 2: retval = '2'; break;
  case 3: retval = '3'; break;
  case 4: retval = '4'; break;
  case 5: retval = '5'; break;
  case 6: retval = '6'; break;
  case 7: retval = '7'; break;
  case 8: retval = '8'; break;
  case 9: retval = '9'; break;
 };

 return retval;
}

void int_to_str(int num)
{
	int power, pow, temp, i, wr_loc, pos;
	power = 0;
	wr_loc = 0;

	// Determine power
	temp = num = abs(num);
	while(temp >= 10)
	{
	 temp = temp/10;
	 power++;
	}

	// Add negative sign if needed
	pos = 0;
	if(num < 0)
```

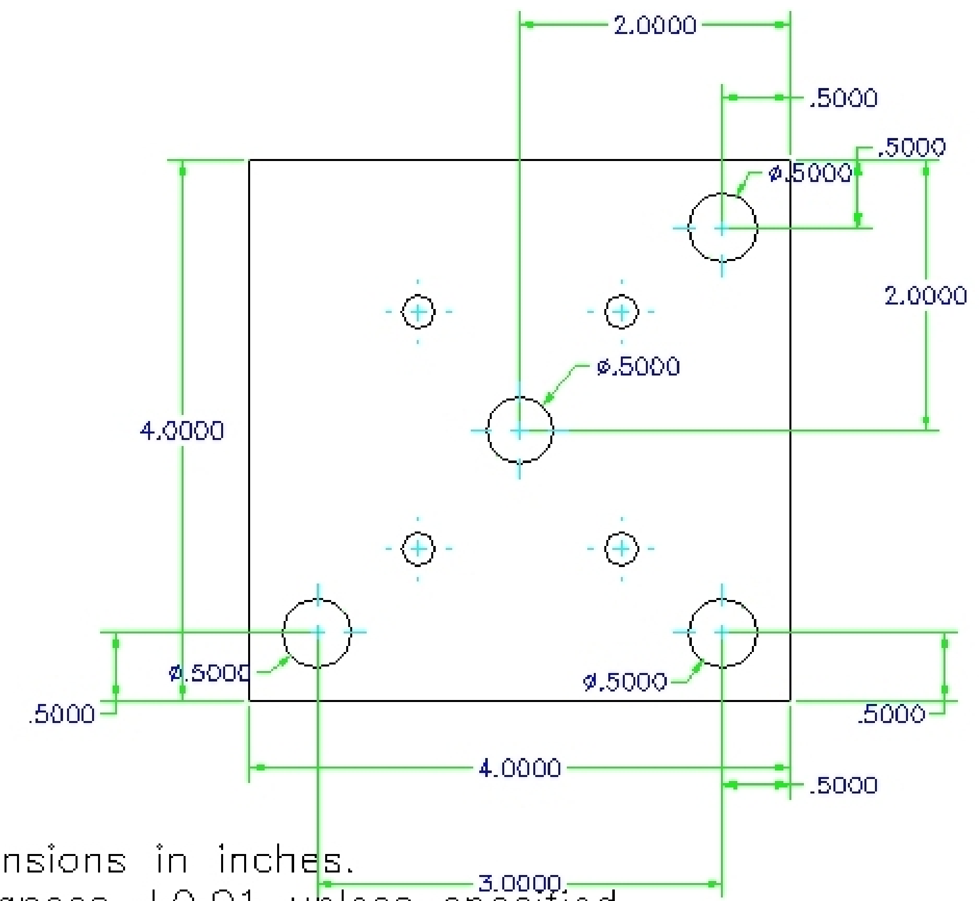
```
         {
                 num *= -1;
                 gstrConvNum[0] = '-';
                 pos = 1;
         }

         // Create string
        while(power >= 0)
        {
                 // Convert first digit
                 temp = num;
                 while(temp >= 10)
                 {
                  temp = temp/10;
                 }
                 gstrConvNum[wr_loc] = int_to_char(temp);
                 wr_loc++;

                 // Remove first digit
                 pow = 1;
                 for(i = 0 + pos; i < power + pos; i++)
                 {
                  pow = pow*10;
                 }
                 if((num - temp*pow) < pow/10)
                 {
                  gstrConvNum[wr_loc] = '0';
                  wr_loc++;
                  power--;
                 }
                 num = num - temp*pow;

                 // Decrement power
                 power--;
        } // end while2

        gstrConvNum[wr_loc] = '\0';
}
```

A.2 Schematics

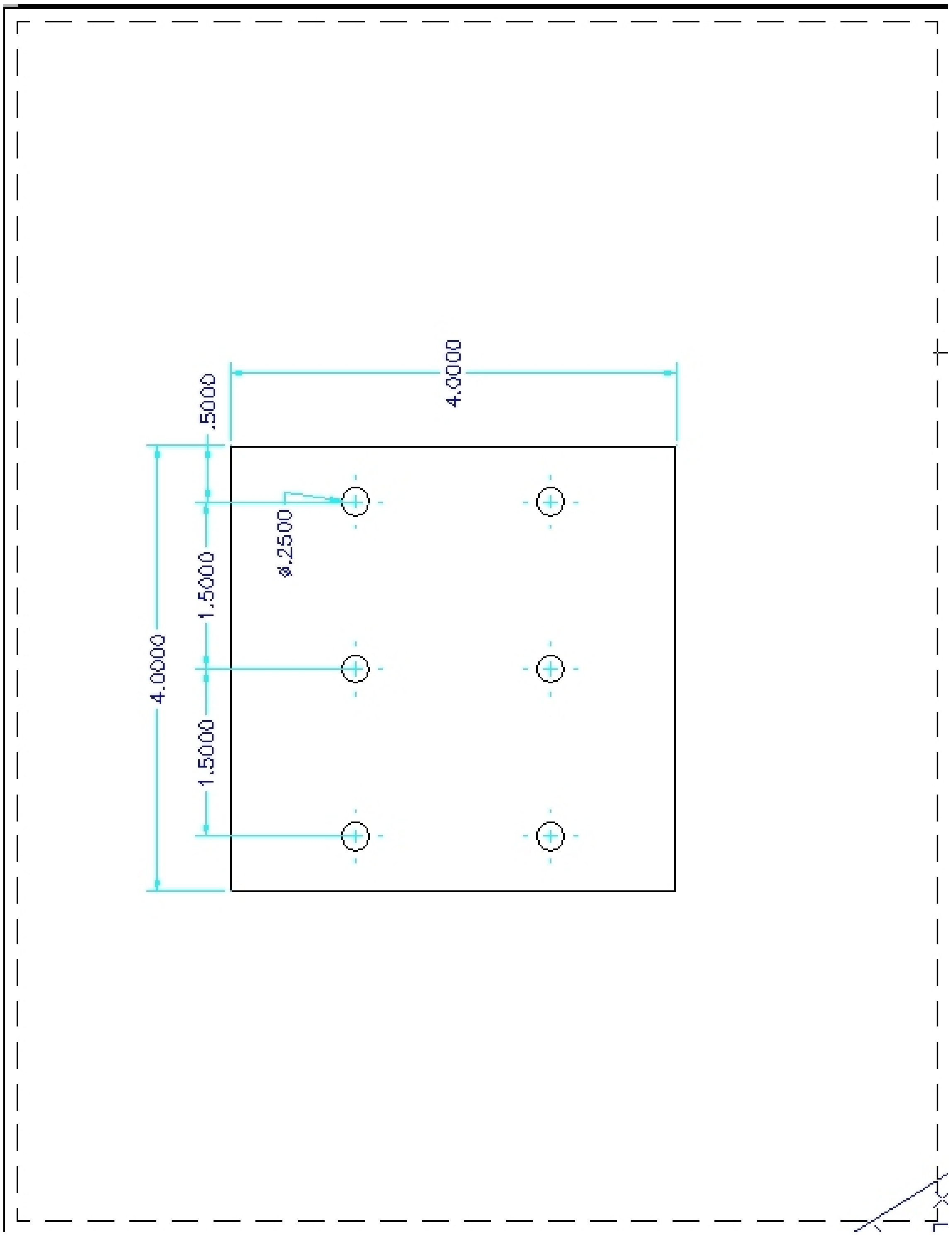


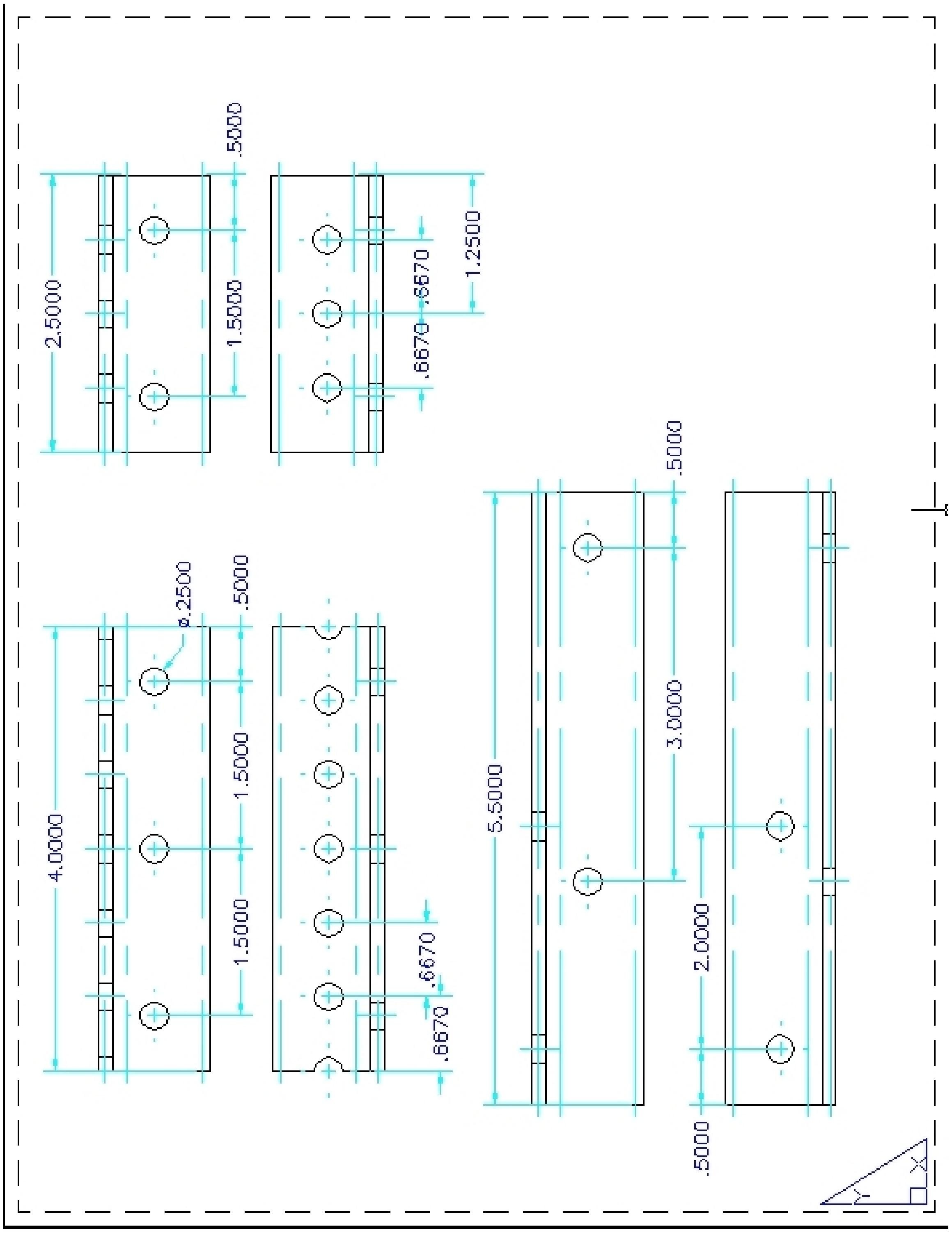


All dimensions in inches.
All tolerances ±0.01 unless specified.
Plate thickness is 0.25



All dimensions in inches.
All tolerances ±0.01 unless specified.
Plate thickness is 0.25


4.0000
.5000
4.0000
1.5000
1.5000
ø.2500



2.5000
1.5000
.5000
.6670
.6670
1.2500
4.0000
ø.2500
.5000
1.5000
1.5000
.6670
.6670
5.5000
.5000
3.0000
2.0000
.5000