NASA-CR-193783

THE CATHOLIC UNIVERSITY OF AMERICA
DEPARTMENT OF ELECTRICAL ENGINEERING

Semiannual Report

A CAPACIFLECTOR-BASED ROBOTIC SYSTEM

Grant Number NAG 5-780

Dr. Charles C. Nguyen, D. Sc.
Principal Investigator and Professor

and

Paulo F. Uribe
Graduate Research Assistant

submitted to
Dr. Charles E. Campbell, Jr.
Mr. John Vranish
Code 714
Goddard Space Flight Center (NASA)
Greenbelt, Maryland

August 1993

SUMMARY OF THE REPORT

This report presents preliminary results obtained from a research grant entitled "Capaciflector-based Control and Imaging," with Grant Number NAG 5-780, for the period between February 1, 1993 and August 1, 1993.

This report deals with the development of a robotic system which is used to evaluate the feasibility of servicing and repairing the Hubble Space Telescope (HST) using robots. In particular, the task of opening the HST's toolbox is considered in this report. First the main components of the robotic system will be introduced. Then each component will be described in detail from low-level to high-level. Finally tasks that have been accomplished during the reporting period for the development of the robotic system will be presented. Listings of source codes for the accomplished tasks are given at the end of the report.

1. THE ROBOTIC SYSTEM

The task of opening the Hubble Space Telescope's Tool Box using the RRC K2 series robotic arm requires the use of a complex system made of computers, sensors, and controllers. Figure 1 shows a block diagram of the system currently in use to achieve this task.

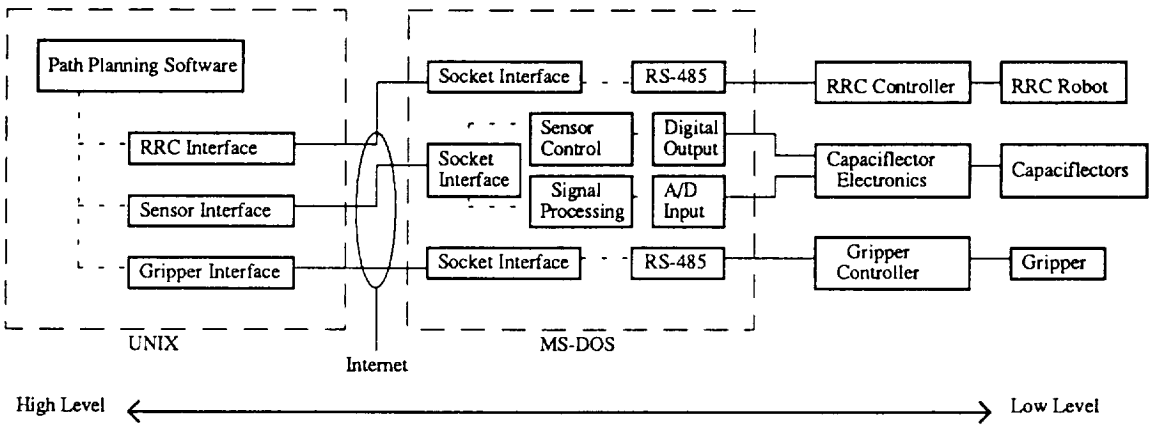

Figure 1. Block diagram of robotics system.

As can be seen in Figure 1, the software development is done on both UNIX and MS-DOS platforms. The software in both platforms is written in C/C++ and the communication between systems is accomplished with the use of the Simple Sockets Library (SSL) [1]. The UNIX software can run on any UNIX machine connected to the Goddard Network. The MS-DOS software runs on a 80386 machine equipped with two RS-485 ports and an OMEGA analog-to-digital (A/D) and digital-to-analog (D/A) board. Accomplished tasks for the development of the above system are presented in the following sections.

2. CAPACIFLECTOR

The capaciflector [2,3] is a capacitive sensor which has an increased sensitivity due to a shield that is placed between the sensor and ground. This shield prevents field lines from the sensor to go directly to ground. To accomplish this, a voltage follower is connected from the sensor to the shield, thereby keeping the shield at the same potential as the sensor. The capaciflector sensors and electronics that were in use were very susceptible to electrical noise emitted by the robot. The typical sensor setup is shown in Figure 2a. As seen in this figure, the capaciflector's shield is mounted over the robot ground. This setup works well as long as the robot is not powered. But when it is powered, the noise generated by the robot is induced into the shield which keeps it from being at exactly the same

potential as the sensor. This problem is solved by separating the sensor's circuit ground from the robot ground and "shielding" the shield with the sensor ground (see Figure 2b). The above separation resulted in a signal with substantially less noise than previously obtained.

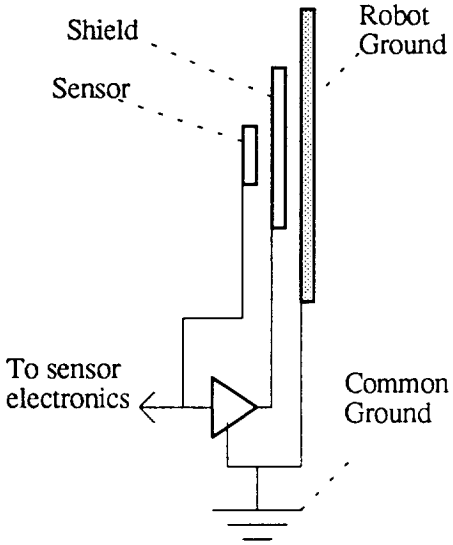

Figure 2a. Previous sensor setup.

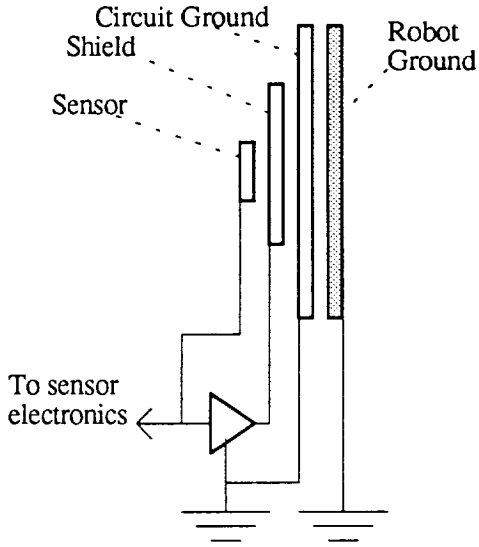

Figure 2b. Improved sensor setup.

3. SENSOR SOFTWARE (MS-DOS)

In order to reduce the amount of space and cables required to use eight capaciflector sensors, a circuit that multiplexes eight sensors into two analog to digital converters was developed. This circuit requires that the multiplex control signals be generated externally. This is done by a 80386 computer with an OMEGA board that has digital output and analog to digital (A/D) input channels. Figure 3 shows a block diagram of the sensor interface to the 80386. It was

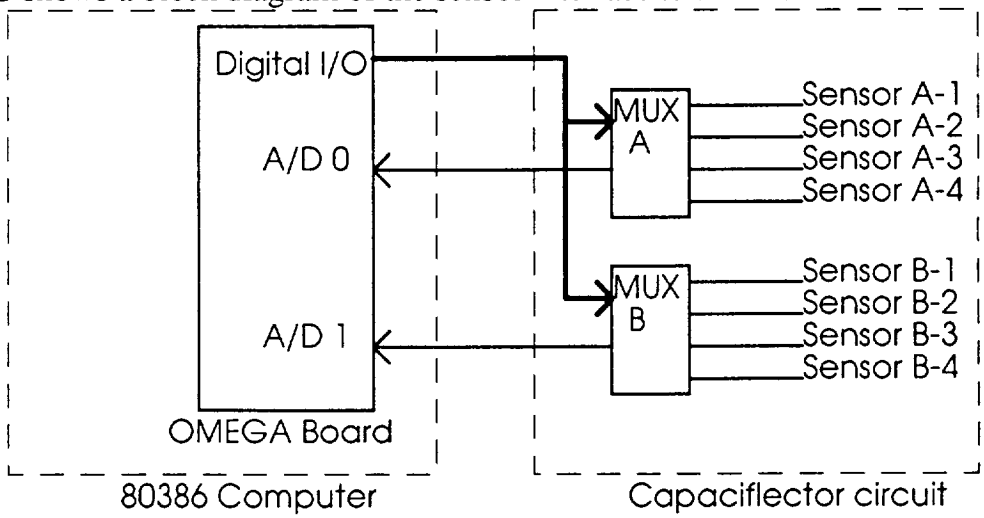

Figure 3. Block diagram of the sensor connection to a 80386 computer.

found through experimentation that the sensor circuit does not respond reliably to a fast multiplexing control signal due to the settling time of an integrated circuit not being obeyed. Consequently a complex method using a clock generated computer interrupt was developed to read all eight sensors at the fastest reliable rate. This algorithm schedules the set-up, A/D conversion, and the read operations for each sensor in the following manner:

Clock	n-1	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
	sumb4	suma1	sumb1	suma2	sumb2	suma3	sumb3	suma4	sumb4	suma1
Action	adca4	adcb4	adca1	adcb1	adca2	adcb2	adca3	adcb3	adca4	adcb4
	rdb3	rda4	rdb4	rda1	rdb1	rda2	rdb2	rda3	rdb3	rda4

Where sumax = set-up multiplexer A, sensor x; sumbx = set-up multiplexer B, sensor x; adcax = start the A/D conversion on multiplexer a, sensor x; adcbx = start the A/D conversion on multiplexer b, sensor x; rdax = read A/D value from multiplexer a, sensor x; rdbx = read A/D value from multiplexer b, sensor x.

The fastest clock fequency that the sensor circuit can work reliably is 20Hz. Therefore with the interrupt scheduled algorithm there is a 50mS settling time between the set-up and start of the A/D conversion and 50mS between the start of conversion and 50mS between the start of conversion and the read of the A/D value. At this rate each sensor is read at 2.5 Hz. This slow rate limits the velocity at which the robot can move since at a fast velocity the robot can hit an object before it is detected by the sensor. To take care of this problem a second mode (static mode) of operation was added to the software. In this new mode one sensor is selected from multiplexer A and one from multiplexer B. The two sensors are then read continuously. Since no multiplexing is involved, there is no setling time that has to be obeyed and each sensor can be read as fast as the 80386 computer and the OMEGA board can read them. The current rate for the static mode is 25Hz per sensor. At this new rate the robot can operate at faster speeds.

After the analog signals from the sensors have been converted to digital values they go through several processes. Since each sensor outputs a different voltage when it is far from an object, the sensors have to be normalized so that each sensor reads zero at that distance. Then they are scaled so that each sensor reads 10 when it is touching an object. The equation used to normalize and scale is the following:

$$V=10\frac{S_n - Si_n}{10 - Si_n} \qquad (1)$$

where S_n is the value from sensor n and Si_n is the intial value (far from any object) from sensor n. It has been found through experimentation that any two sensors which have the same surface area will exibit the same responce after their

signals have been normalized and scaled using Equation 1. If one sensor is bigger than the other, the bigger sensor will always exhibit a higher reading than the smaller sensor. However the two readings will converge to 10 as the sensors get close to an object, therefore even if the sensors have a different size or shape their values can be used after being processed by equation 1 for such tasks as leveling or centering as long as the task is performed close enough to the object.

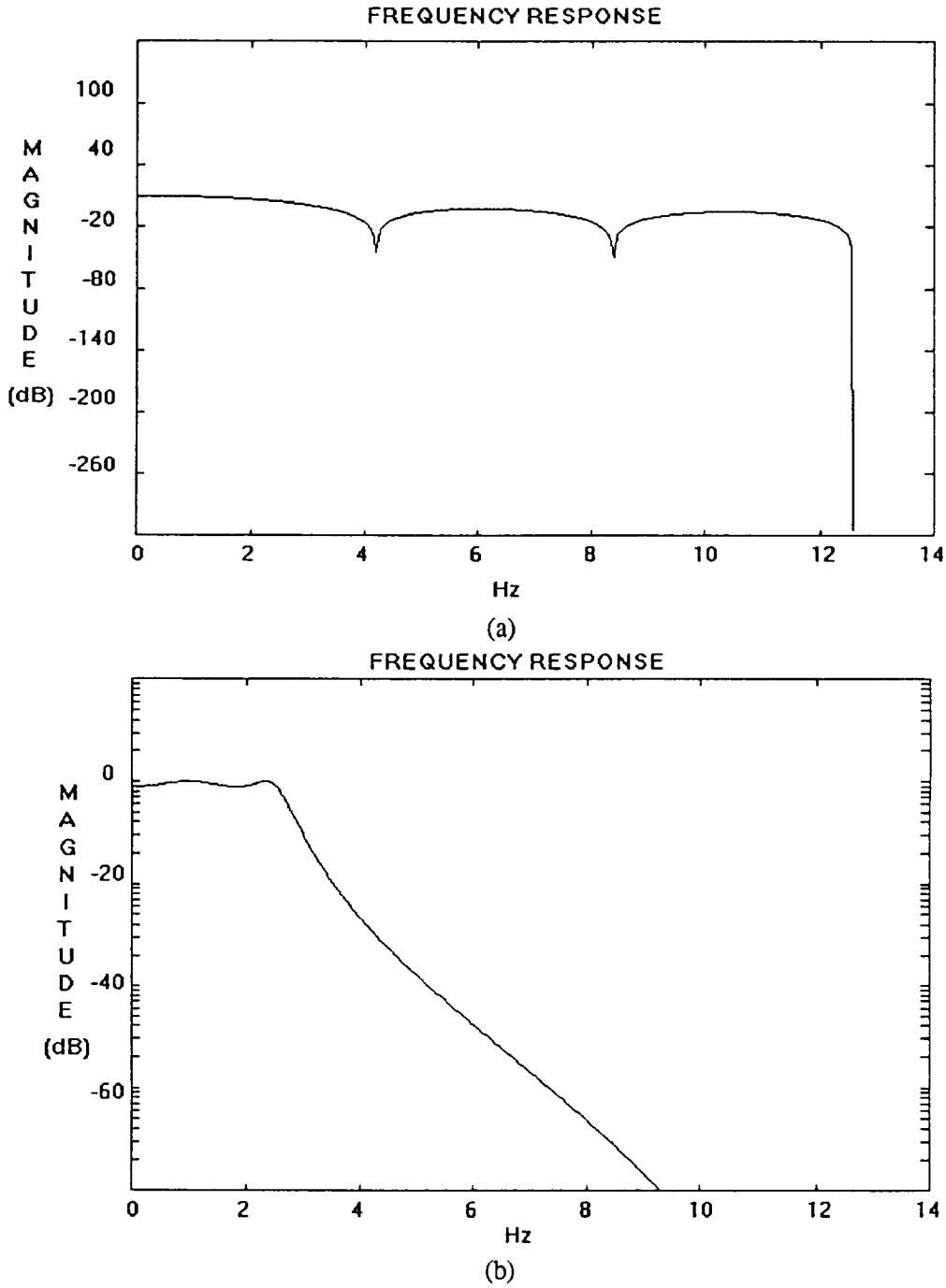

Figure 4. Frequency response of (a) run-length-average filter and (b) Chebyshev filter.

After the sensor readings have been normalized and scaled they are filtered using two algorithms: a run-length-average of the last 5 samples, or a chebyshev fourth order filter. As seen from Figure 4, the chebyshev filter has the best frequency response. However since the nyquist criterion is not obeyed in the sampling process (there is no low pass filter in front of the A/D converter) the run-length-average filter, which is less sensitive to aliasing, is being used until a low pass filter is installed. The sensor control software is shown in Appendix 1 and the graphical display routines are shown in Appendix 2.

4. SENSOR SOFTWARE (UNIX)

In order to interface the MS-DOS computer controlling the sensor hardware to the path planning software running on UNIX, a server using the Simple Sockets Library (SSL) was developed. This server can read the sensors on the MS-DOS PC and send them asynchronously to up to five different clients running in any machine connected to the Internet network. One of the clients must be the path planning software and the other four can be graphical display programs. The software is shown in Appendix 3.

5. GRIPPER SOFTWARE (MS-DOS & UNIX)

The gripper software is an interface between the path planning program and the gripper controller. The UNIX portion of the software is a set of subroutines that when called, send the type of action required from the gripper and a parameter specifying the amount of that action to the MS-DOS computer through the Internet network. The MS-DOS computer in turn sends those commands to the gripper controller through a RS-485 port.

Since the gripper controller can not receive a new command until the previous one has been completed, the MS-DOS software has to monitor the gripper while also controlling the sensor electronics and performing the signal processing described above. After a gripper command has been completed, a simple handshake signals the UNIX portion of the software. Appendix 4 shows both the UNIX and MS-DOS portions of the software.

6. PATH PLANNING SOFTWARE

The path planning software is a program that sequentially performs robot tasks. Each task is not performed until the previous one has been completed to a certain degree of success. Opening of the tool box requires three basic operations: pulling a pip-pin, unlocking a latch mechanism, and opening the tool-box door. Since the robot's end effector starts at an arbitrary position, there are three

rotational and three linear displacement unknowns at each point of interest in the tool-box. By leveling the end-effector against the surface of the tool-box, two of the three rotational unknowns are removed. By leveling against the handlebar of the tool-box, the third rotational unknown is removed.

The leveling task is performed by reading two sensors, s1 and s2, that are located at the extremities of the fingers and rotating the gripper in order to minimize the error equation e = |s1 - s2|.

The algorithm that performs the leveling takes into account the size and distance between the sensors used to level. It also accounts for numerical errors produced in the calculation of the inverse-Jacobian which in turn cause translation errors of the end effector. The basic equation for leveling is given by

$$ROT = \frac{(S1K1 - S2K2)}{\max(S1,S2)} Kr \tag{2}$$

where ROT is the rotation to be performed in radians, K_1 and K_2 are proportionality constants used when the sensors are not of the same size, and K_r is a conversion factor which takes into account the distance between the fingers. The term in the denominator attenuates the rotation when the fingers get close the leveling surface. The displacement of the end effector while leveling isexpressed as

$$d = [V_L - \max(S1,S2)]K_d - ROT \cdot K_j \tag{3}$$

where V_L is the voltage at which the leveling is performed, K_d is the displacement gain, and the term ROT K_j compensates the displacement error introduced by the inverse-Jacobian when a rotation is performed.

7. CONCLUSION

In this report we have considered the development of a robotic system used to evaluate the feasibility of servicing and repairing the Hubble Space Telescope (HST) using robots. All the components of the system were described from low-level to high-level and recommendations were made on how to increase the sensitivity and performance of the Capaciflector. The task of opening the HST's toolbox was presented with an emphasis on how the three rotational unknowns are removed. The source code for all the accomplished tasks was presented in the appendices. Future research can be directed to studying new control algorithms such as a six degree of freedom virtual force control to implement a class of impedance control using Capaciflectors.

8. REFERENCES

[1] **Campbell, Jr., Charles E.**, "The Simple Socket Library," Goddard Space Flight Center, Internal Distribution, October, 1992.

[2] **Vranish, J.M., R.L. McConnell, and Mahalingam, S.**, "Capaciflector Collision Avoidance Sensors for Robots," *Computers and Electrical Engineering: An International Journal*, Vol. 17, Number 3, pp. 173-180, 1991.

[3] **Nguyen, C.C.**, "Capaciflector-Based Virtual Force Control and Centering," Semianual Report, Grant NAG 5-780, *The Catholic University of America,* Goddard Space Flight Center, February, 1993.

Appendix 1. Sensor control software.

```
/*
 *      FACILITY:
 *              NASA Goddard Robotics Lab
 *              Engineering Test Bed
 *
 *                      ABSTRACT:
 *              Subroutines to read the new sensors.
 *
 *      DESIGN DECISIONS:
 *              In order for the new sensors to work properly, an interrupt handler
 *              is used to multiplex and read the sensor values at a low rate, instead
 *              of the fast rate used for the old sensors (in the microseconds range).
 *
 *      MODIFICATION HISTORY:
 *              15 April 1993    Paulo Uribe, CUA
 *                      Created
 *              24 June 1993    Paulo Uribe, CUA
 *                      Added a 4th order Chebyshev filter with cutoff frequency at .2pi
 */

#include "newsens.h"

/* filter variables */
int cap_average[9][AVERAGE_N], cap_pntr[9], dc_offset[9];
double cap_x[9][AVERAGE_N];
double cap_y[9][AVERAGE_N];
double cap_z[9][AVERAGE_N];
double hp[2];
double a[]={0.0, 1.4996, -0.8482};
double aa[]={0.0, 1.5548, -0.6493};
double b[]={.001836, 2.0, 1.0};
double gain[9];

extern int raw_skin_data[20];

/* Interrupt routine variables */
int interrupt_enable=0, index, sens_num, cap_val[9];
int read_s[18]={ 0,4,1,5,2,6,3,7,8,0,4,1,5,2,6,3,7,8 } ;
int set_up[18]={ 1,5,2,6,3,7,8,0,4,1,5,2,6,3,7,8,0,4 } ;
int a_to_d[18]={ 1,0,1,0,1,0,1,2,0,1,0,1,0,1,0,1,2,0 } ;

/* Sensor number to mux address conversion array */
char dbit[9]={ 1+16, 2+16, 4+16, 8+16, 1+32, 2+32, 4+32, 8+32, 0 };

int mux1_channel = MUX1_CHANNEL;
int mux2_channel = MUX2_CHANNEL;

void mux(int enable)
/* Enables/disables multiplexing of the sensors.
        input paramter:
                enable = 1, enables muliplexing
```

```
                        = 0, disables multiplexing

        Note: When multiplexing is disabled the sensors set by the set_mux_channel()
                function are read continuously.
*/
{
        if(enable) interrupt_enable = 1;
        else
        {
                interrupt_enable = 0;
                outp(AD_ADDRESS+9 , 0); /* Reset the Omega board */
        }
}

int set_mux_channel(unsigned int mux1, unsigned int mux2)
/* Sets the channels to read from each mux when the function mux(0)
        has been called.
        input parameters:
                mux1 = integer from 0 to 3
                mux2 = integer from 4 to 7

        ouput parameters:
                0 = mux1 or mux2 out of range, values not set
                1 = values set
*/
{
        if(mux1 < 4 && (mux2 > 3 && mux2 < 8) )
        {
                mux1_channel = mux1;
                mux2_channel = mux2;
                ADC_Board_In(mux1,0,0);
                ADC_Board_In(mux2,0,0);
                ADC_Board_In(mux1,0,0);
                ADC_Board_In(mux2,0,0);
                ADC_Board_In(mux1,0,0);
                ADC_Board_In(mux2,0,0);
                return 1; /* ok */
        }
        else
        {
                return 0; /* error */
        }
}

int ADC_Board_In( int sensor_number, int normalize, int filter)
/* Read Analog Value from the Boards
        global paramaters:
                interrupt_enable = 0 reads the channels defined by MUX1_CHANNEL and
                                        MUX2_CHANNEL and returns 0 for any other sensor number.
                                 = 1 returns the sensor values read by the interrupt handler.
        input parameters:
                sensor_number = sensor to read ( integer from 0 to 7 )
                normalize = integer 0 for no normalization of the sensor
                          = integer 1 to normalize the sensor
```

```
                filter = integer 0 for no filtering of the sensor data
                       = integer 1 to filter the sensor data return value:
                integer from -2048 to 2048 representing the sensor value from -10V to 10V.
*/
{
        register i;
        int value,omega_channel,n;
        long sum;

        double dsum1,dsum2;

        if(sensor_number==8)
        {
                /* Limit cycling detection */
                dsum1=(double)(raw_skin_data[0]-2048)-0.617*hp[1]+1.111*hp[0];
                value=abs((int)(0.617*dsum1-1.234*hp[0]+0.617*hp[1]));
                hp[1]=hp[0];
                hp[0]=dsum1;
                return value;
        }
  else
        {
                if ( interrupt_enable )
                {
                        value = cap_val [sensor_number];
                }
                else if (sensor_number==mux1_channel || sensor_number==mux2_channel )
                {
                        if(sensor_number < 8)
                        {
                                omega_channel=1;
                                if(sensor_number < 4) omega_channel=0;
                                outp(AD_ADDRESS + 0x0B, dbit[sensor_number] & 0x0F );
                                outp(AD_ADDRESS + 0x0B, dbit[sensor_number] ); /* Select Analog
input */
                                outp(AD_ADDRESS + 0x0B, dbit[sensor_number] & 0x0F );
                        }
                        outp(AD_ADDRESS+1 , omega_channel ); /* Select Omega Board input
channel */

                        outp(AD_ADDRESS+2 , 0); /* Start conversion */

                        while( (inp(AD_ADDRESS) & 0xC0) !=0x40); /* Wait for conversion to end
*/

                        value = inpw(AD_ADDRESS+3);
                }
                else return 0; /* return 0 if sensor_number is out of range */

                /* remove any dc offsets and normalize the gains */
                if(normalize)
                {
                        value -= dc_offset[sensor_number];
                        value = (int) ((double)value * gain[sensor_number]);
```

```
                if(value>2048) value=2048;
                else if(value<-2048) value=-2048;
        }
}
/* average last AVERAGE_N points */
if(filter==1)
{
        cap_pntr[sensor_number]++;
        if(cap_pntr[sensor_number]>AVERAGE_N-1) cap_pntr[sensor_number]=0;
        sum=0;
        cap_average[sensor_number][cap_pntr[sensor_number]]=value;
        for(i=0;i<AVERAGE_N;i++)
        {
                sum+=(long)cap_average[sensor_number][i];
        }
        value=(int)sum/AVERAGE_N;
}
else if(filter==2)
{
        n=sensor_number;
        for(i=2;i;i--)
        {
                cap_x[n][i]=cap_x[n][i-1];
                cap_y[n][i]=cap_y[n][i-1];
        }
        cap_x[n][0]=(double)value+a[1]*cap_x[n][1]+a[2]*cap_x[n][2];
        dsum1=cap_x[n][0]+2*cap_x[n][1]+cap_x[n][2];
        cap_y[n][0]=dsum1+aa[1]*cap_y[n][1]+aa[2]*cap_y[n][2];
        dsum2=cap_y[n][0]+2*cap_y[n][1]+cap_y[n][2];
        value=(int)(dsum2*b[0]);

}
else if(filter==3)
{
        n=sensor_number;
        for(i=2;i;i--)
        {
                cap_x[n][i]=cap_x[n][i-1];
                cap_y[n][i]=cap_y[n][i-1];
                cap_z[n][i]=cap_z[n][i-1];
        }
        /* First stage (2nd order filter) */
        cap_x[n][0]=(double)value+1.2686*cap_x[n][1]-0.7051*cap_x[n][2];
        dsum1=cap_x[n][0]+2*cap_x[n][1]+cap_x[n][2];

        /* Second stage (2nd order filter) */
        cap_y[n][0]=dsum1+1.0106*cap_y[n][1]-0.3583*cap_y[n][2];
        dsum1=cap_y[n][0]+2*cap_y[n][1]+cap_y[n][2];

        /* Third stage (2nd order filter) */
        cap_z[n][0]=dsum1+0.9044*cap_z[n][1]-0.2155*cap_z[n][2];
        dsum1=cap_z[n][0]+2*cap_z[n][1]+cap_z[n][2];

        value=(int)(dsum1*0.0007378);
```

```
        }
        if(sensor_number==8)
        {
                /* Limit cycling detection */
                if(value>3072) printf("\n*** TOUCHING PIPPIN HANDLE ***");
                else if(value>2119) printf("\n*** LIMIT CYCLE          ***");
        }


        return value;
}


void interrupt far sensor_interrupt_handler(void)
/* This interrupt handler reads the sensors following the pattern described
        by the read_s[], set_up[], and a_to_d[] arrays. The index to these arrays
        is      incremented in each interrupt cycle. When it reaches MAX_INDEX the
        index   is reset to 0. During cycle X the sensor to be read is held in
        read_s[X], the omega board channel of the sensor that will be read in the
        next cycle is held in a_to_d[X], and the number of the sensor that will
        be read to cycles ahead is held in set_up[X]. This procedure mimizes the
        settling time required by the rms-dc converters in the sensor electronics.

        In order to increase the sampling rate of the sensors, a different word has
        to be loaded into the timer/counter chip's divide register of the PC. This
        is done using the set_timer() function.
*/
{
        if(interrupt_enable)
        {
                index++;
                if( index > MAX_INDEX ) index=0;

                cap_val[ read_s[index] ] = inpw(AD_ADDRESS+3);

                sens_num = set_up[index];

                if(sens_num < 8)
                {
                        outp(AD_ADDRESS + 0x0B, dbit[sens_num] & 0x0F );
                        outp(AD_ADDRESS + 0x0B, dbit[sens_num] ); /* Select Analog input */
                        outp(AD_ADDRESS + 0x0B, dbit[sens_num] ); /* Select Analog input */
                        outp(AD_ADDRESS + 0x0B, dbit[sens_num] & 0x0F );
                }

                outp(AD_ADDRESS+1 , a_to_d[index] );
                outp(AD_ADDRESS+2 , 0);

        } /* End Interrupt Enable */

        /* hand control to other timer interrupt service routines */
        (*sensor_interrupt_oldhandler)();
}
```

```
void initialize_sensors(void)
/*      Removes any dc offsets and sets the gains of each sensor in order for them
        to have the same response.
*/
{
        unsigned int i,j,k;
        int cap_value_a[9],cap_value_b[9],cap,omega_mux,first;
        long sum;
        char string[30];

        interrupt_enable = 0;

        outp(AD_ADDRESS+9 , 0); /* Reset the Omega board */

        for(i=0;i<SENSNUM;i++)
        {
                _settextcolor(10);
                _settextposition(4,1);
                sprintf(string,"Initializing offsets and gains for sensor: %d",i);
                _outtext(string);

                if(i<8)
                {
                        outp(AD_ADDRESS + 0x0B, dbit[i] & 0x0F );
                        outp(AD_ADDRESS + 0x0B, dbit[i] ); /* Select Analog input */
                        outp(AD_ADDRESS + 0x0B, dbit[i] & 0x0F );
                }
                omega_mux=2;
                if(i<8) omega_mux=1;
                if (i<4) omega_mux=0;

                for(j=0;j<60000;j++);

                sum=0;
                first=0;

                for(j=0;j<101;j++)
                {
                        outp(AD_ADDRESS+1 , omega_mux );
                        outp(AD_ADDRESS+2 , 0);

                        while( (inp(AD_ADDRESS) & 0xC0) !=0x40);

                        cap = inpw(AD_ADDRESS+3);
                        if( !first ) first=1; /* Throw out first value */
                        else sum += (long)cap;
                        for(k=0;k<1000;k++);
                }
                sum /= 100;
                cap_value_a[i] = (int)sum;
                cap_value_b[i] = 2048;
                dc_offset[i] = cap_value_a[i];
                if(cap_value_b[i] - dc_offset[i])
```

```
                gain[i] = 2048.0 / (double)(cap_value_b[i] - dc_offset[i]);
            else gain[i]=10; /* Error, set gain to 10 */
            if(gain[i] > 1.50)
            {
                _settextcolor(12);
                _settextposition(5,1);
                sprintf(string,"WARNING: Gain greater than 1.5 for sensor(s) ");
                _outtext(string);
                _settextposition(5,44+i*2);
                sprintf(string,"%d ",i);
                if(gain[i]==10) _settextcolor(14);
                _outtext(string);
            }

        }
}

void set_timer(char high_byte, char low_byte)
/* Sets the divide register of the timer/counter chip in the PC.
        The value loaded by the BIOS is 0xD86E. This value represents
        an 18.2Hz interrupt. A lower value speeds up the timer.
*/
{
        _disable() ;
        outp(0x43,0x34) ;
        outp(0x40,low_byte) ;       /* low byte */
        outp(0x40,high_byte) ;      /* high byte */
        /*make sure pic can handle interrupt*/
        outp(EOIADDR,0x20);
        _enable();
}

void install_sensor_interrupt(void)
/* installs the new sensor interrupt */
{
 /* Get existing timer vector */
 _disable();
 sensor_interrupt_oldhandler = _dos_getvect( 0x1c );
 _enable();
 _disable();
 _dos_setvect( 0x1c, sensor_interrupt_handler );
 _enable();
}

void uninstall_sensor_interrupt(void)
/* restores the timer interupt to the old vector */
{
 _disable();
 _dos_setvect(0x1c, sensor_interrupt_oldhandler);
 _enable();
}
void print_gain(void)
/* prints the dc offsets and gains for each sensor */
{
```

```
        char string[60];
        int i;

        _settextcolor(12);
        _settextposition(6,10);
        _outtext("=========================================");

        _settextcolor(15);
        for(i=0;i<9;i++)
        {
                _settextposition(7+i,10);
                sprintf(string,"Sensor #%2d: DC offset: %4d   Gain:%3.3f",i,dc_offset[i],gain[i]);
                _outtext(string);
        }
        _settextposition(16,10);
        _settextcolor(12);
        _outtext("====== Press any key to continue =========");
        getch();
}
```

Appendix 2. Graphical display routines.

```
void display(int graphon,int norm,int filter, int raw_skin_data[])
/*  input parameters:  graphon = 0  Display text only
                               = 1  Display graphics

                       norm = 0  normalizing and scaling is off
                            = 1  normalizing and scaling is on

                       filter = 0  No filtering
                              = 1 Run length average filter
                              = 2 Chevyshev filter
                              = 3 Butterworth filter
                       raw_skin_data[]  data to be plotted
*/
{
 register i,j;
 char string[60];
 int temp;
 double tempf;
 static char buf[BUFSIZE];

 _settextcolor(15);
 _settextposition(1,1);
 _outtext("Normalizing: O");
 if(aver)
 {
  _settextposition(1,15);
```

```
 _outtext("N ");
}
else
{
 _settextposition(1,15);
 _outtext("FF");
}
_settextposition(2,1);
_outtext("Filtering : ");
if(!filter)
{
 _settextposition(2,14);
 _outtext("OFF ");
}
else if(filter==1)
{
 _settextposition(2,14);
 _outtext("FIR "); /* Finite Impulse Response filter */
}
else if(filter==2)
{
 _settextposition(2,14);
 _outtext("cIIR"); /* Chebyshev IIR filter */
}
else if(filter==3)
{
 _settextposition(2,14);
 _outtext("bIIR"); /* Butterworth IIR filter */
}
_settextposition(1,65);
_outtext("Cap-socket: ON ");
if(rrc_socket)
{
 _settextcolor(15);
 _settextposition(2,65);
 _outtext("RRC-socket: ON ");
}
else
{
 _settextcolor(15);
 _settextposition(2,65);
 _outtext("RRC-socket: OFF");
}

for(i=1;i<10;i++)
{
 _settextposition(1,16+i*5);
 _settextcolor(16-i);
 sprintf(string,"%d",i);
 _outtext(string);
}
_settextcolor(15);
_settextposition(2,19);
sprintf(string,"%4d %4d %4d %4d %4d %4d %4d %4d %4d" ,
```

```
raw_skin_data[0],raw_skin_data[1],raw_skin_data[2],
raw_skin_data[3],raw_skin_data[4],raw_skin_data[5],
raw_skin_data[6],raw_skin_data[7],data);
/* Display numerically the values of the sensors */
 _outtext(string);

/* Display graphically the values of the sensors */
if(graphon)
{
 _setcolor(0);
 _moveto(xpos,80);
 _lineto(xpos,480);
 for(j=0;j<SENSEN;j++)
 {
  if( raw_skin_data[j] > 2000)
  {
   _settextcolor(15-j);
   _settextposition(1,21+j*5);
   sprintf(string,"%d",j+1);
   _outtext(string);
   _setcolor(15-j);
   if(sensor_num==j)
   {
    tempf = (double)((raw_skin_data[j])-2024)*constant;
    tempf+= 24;
   }
   else tempf = (double)(raw_skin_data[j]-2000);
   tempf = 170.0*log10(fabs(tempf)+1);
   temp = 680 - (int)tempf;
  _setpixel(xpos,temp);
  }
  else
  {
   _settextcolor(7);
   _settextposition(1,21+j*5);
   sprintf(string,"%d",j+1);
   _outtext(string);
  }
 }
 if(++xpos>600) xpos=15;
 if(xpos==15 || xpos==315) time1=time(&times);
 else if(xpos==265 || xpos==565)
 {
  time2=time(&times);
  sprintf(buf,"Sampling Frequency=%2.1fHz",250.0/difftime(time2,time1));
  _settextposition(5,1);
  _settextcolor(15);
  _outtext(buf);
 }
}
_settextposition(4,1);
printf("\n");
}
```

Appendix 3. Sensor server (UNIX).

```
/*
 *      FACILITY:
 *              NASA Goddard Robotics Lab
 *              Engineering Test Bed
 *
 *                      ABSTRACT:
 *              Capaciflector sensor server.  Accepts 1 sensor client, 1 control (path planning software)
 *              client and up to 4 sensor-read clients.  Software written using client-server architecture.
 *
 *      MODIFICATION HISTORY:
 *              15 May 1993     Paulo Uribe, CUA
 *                      Created
 *              20 July 1993    Paulo Uribe, CUA
 *                      Added capabilities for up to 4 sensor-read clients to be accepted by server.
 */


#include <unistd.h>
#include <stdio.h>
#include <math.h>
#include <sys/times.h>
extern "C"{
#include "xmath.h"
#include "sockets.h"
#include "xtdio.h"
}
#define MAX_BUF 128
#define DATABAD 30
#define MAXSRCLIENTS 2

 Socket *server,*client;       // server socket pctraj
 Socket *control;     // accept pc connection
 Socket *sensor;       // client to sensor program
 Socket *sread[MAXSRCLIENTS];

 void Open_server();  // open server pctraj
 void Accept_client(); // accept pc connection
 void Connect_sensor(); // open client to sensor program
 void Close_sensor(); // close sensor connection
 void get_control_cmd();
 void get_sr_cmd(int);
 void get_cap_val();

 int yn();

 char buf[MAX_BUF],cmd[MAX_BUF];

 char ServName[]="Capaciflector";
 int cap_value[10];
 float fcap[10];
```

```
 int run=1,control_socket,sensor_socket;
 int muxflag=0,capflag=0,sendflag=0,toutcount=0,sensortout=1000;
 int mux1,mux2,mux;
 int sensnum,const_flag;
 double con;
 int shift,sr_socket,srsflag[MAXSRCLIENTS];
 int ss_hs=0;
 char s_hs[50];

void main()
{
 register i;
 int flag=1,databad=DATABAD;

 fprintf(stderr,"\nCapaciflector server. V1.2\n");
 fprintf(stderr,"\nOpening server...");
 Open_server();
 fprintf(stderr,"Server open\n");
 Smaskset(server);
 Smasktime(0,10);

 while(run)
 {
  i=0;
  shift=0;
  while(i<MAXSRCLIENTS)
  {
        if(!sread[i])
        {
          Accept_client();
          i=MAXSRCLIENTS;
        }
        else
        {
     sr_socket=1;
     if(Stest(sread[i])<0)
     {
       fprintf(stderr,"\nError in sensor-read socket.");
       sr_socket=0;
     }
     else if(Stest(sread[i])) get_sr_cmd(i);
     if(!sr_socket)
     {
        fprintf(stderr,"\nClosing sensor-read client\n");
        Sclose(sread[i]);
        sread[i]=NULL;
            shift = 1;
     }
        }
        i++;
  }// End MAXCLIENTS
  i=0;
  while(shift)
  {
```

```
shift=0;
while(i<MAXSRCLIENTS)
{
  if(!sread[i] && i<MAXSRCLIENTS-1)
  {
       if(sread[i+1]) shift=1;
       sread[i]=sread[i+1];
       sread[i+1]=NULL;
  }
      i++;
}
}// End While(Shift)
if(control==NULL) Accept_client();
else
{
       if(ss_hs&&!databad)
       {
        Sprintf(control,s_hs);
        if(sensor) Sprintf(sensor,"sk");
        ss_hs=0;
       }
   control_socket=1;
   if(Stest(control)<0)
   {
        fprintf(stderr,"\nError in control socket.");
        control_socket=0;
       }
   else if(Stest(control)) get_control_cmd();
   if(!control_socket)
   {
      fprintf(stderr,"\nClosing control client\n");
      Sclose(control);
          control=NULL;
   }
}
if(capflag)
{
  if (!databad)
  {
   for(i=0;i<8;i++)
    fcap[i]=10*( cap_value[i] - 2048.0 )/2048.0;
        fcap[8]=(int)cap_value[8];
   sprintf(buf,"qc %2.4f %2.4f %2.4f %2.4f %2.4f %2.4f %2.4f %2.4f %2.4f" ,
    fcap[0],fcap[1],
    fcap[2],fcap[3],
    fcap[4],fcap[5],
    fcap[6],fcap[7],fcap[8]);
   if(sendflag && control)
        {
        Sputs(buf,control);
    sendflag=0;
        capflag=0;
        }
   else
```

```
    {
        i=0;
        while( sread[i] && i<MAXSRCLIENTS)
        {
         if(srsflag[i] && sread[i])
       {
        Sputs(buf,sread[i]);
        srsflag[i]=0;
        capflag=0;
           }
           i++;
     }
    }
  }// End if(!databad)
  else
  {
        /* The first several sensor readings after a change in a mux channel are bad. Therefore when a
        change in any mux channel is detected, databad is set to DATABAD and while databad>0
        capflag is set to zero to stop any transmition of the data to any of the sensor-read and
        control clients. */

        fprintf(stderr,"\r[bad data %2d]", databad--);
        fprintf(stderr,"\r              \r");
        capflag=0;
  }
}// End if(capflag)
if(muxflag && sensor)
{
  databad=DATABAD;
  muxflag=0;
  sprintf(buf,"ss %d %d %d",mux1,mux2,mux);
  Sputs(buf,sensor);
}

 if(sensor==NULL) Accept_client();
 else
 {
    sensor_socket=1;
        if(const_flag)
        {
         Sprintf(sensor,"sg %d %f",sensnum,con);
         const_flag=0;
        }
        if(flag)
        {
         sensortout=500;
         flag=0;
         Sprintf(sensor,"se");
        }
    else if(!sensortout)
    {
         fprintf(stderr,"\r     Sensor timed out. Trying again(%d)...\r",toutcount);
         toutcount++;
         if(toutcount>100) sensor_socket=0;
```

```
            flag=1;
           }
           if(Stest(sensor)<0)
           {
            fprintf(stderr,"\nError in sensor socket.");
            sensor_socket=0;
           }
           else if(Stest(sensor))
           {
            get_cap_val();
            flag=1;
           }
           else sensortout--;
           if(!sensor_socket)
           {
                    fprintf(stderr,"\nClosing sensor client\n");
                    Sclose(sensor);
                    sensor=NULL;
           }
  } // End else
 } // End while(run)
 fprintf(stderr,"\nExiting");
 i=0;
 while(sread[i] && i<MAXSRCLIENTS)
 {
  if(sread[i])
  {
   fprintf(stderr,"\nClosing sensor-read client %d",i);
   Sprintf(sread[i],"sq");
   Sclose(sread[i]);
  }
  i++;
 }
 if(control)
 {
  fprintf(stderr,"\nClosing control client");
  Sprintf(control,"sq");
  Sclose(control);
 }
 if(sensor)
 {
  fprintf(stderr,"\nClosing sensor client");
  Sprintf(sensor,"sq");
  Sclose(sensor);
 }
 fprintf(stderr,"\nClosing server\n");
 Smaskunset(server);
 Sclose(server);
}

void Open_server()
{
 int tr=2;
```

```
 while(tr)
 {
 server = Sopen (ServName,"c");
 if(server)
 {
  Sprintf(server,"cp");
  if(tr==2) fprintf(stderr,
        "\nWARNING: <%s> Server already open.  Will try to close it...",
        ServName);
  Sprintf(server,"cQ");
  Sclose(server);
  sleep(1);
  tr--;
 }
 else
 {
  break;
 }
 }
 if(!tr)
 {
  fprintf(stderr,
   "\nCould not close it.  Press [RETURN] to start a new server");
  getchar(stderr);
 }
 else if(tr==1)
  fprintf(stderr,"\nPrevious server closed.  Opening new server...");

 server = Sopen (ServName, "s");

 if (!server)
 {
  // If the Server already exists remove it
  Srmsrvr (ServName);
  server = Sopen (ServName, "s");
  if (!server)
  {
   fprintf(stderr,"\nUnable to open <SensorServer> as server..\n");
   exit(0);
  }
 }
}
void Accept_client()
{ // Accept Pc Connection
 register i;
 if( Smaskwait() ) client = Saccept( server );
 else
 {
  return;
 }
 if(client!=NULL)
 {
  i=40;
  while(!Stest(client) && i--) fprintf(stderr,".");
```

```
        /* Wait for client to identify itself */
if(Stest(client))
{
 Sgets ( buf, MAX_BUF, client );
 if (buf[0]=='s' && buf[1]=='n')
 {
  if (sensor==NULL)
  {
   sensor=client;
   fprintf(stderr,"\nPC connected\n");
  }
  else
  {
   fprintf(stderr,"\nERROR: A second sensor client has been connected");
       fprintf(stderr,"\n      Closing it...");
       Sprintf(client,"sQ");
       Sclose(client);
       fprintf(stderr,"Closed.");
      }
 }
 else if (buf[0]=='s' && buf[1]=='r')
 {
  i=0;
  while( sread[i] && i<MAXSRCLIENTS ) i++;
      if ( i==MAXSRCLIENTS )
      {
   fprintf(stderr,
        "\nERROR: A fifth sensor read client has been connected");
   fprintf(stderr,"\n      Closing it...");
   Sprintf(client,"sQ");
   Sclose(client);
   fprintf(stderr,"Closed.\n");
  }
      else
      {
       sread[i]=client;
       fprintf(stderr,"\nSensor-read client connected\n");
      }
 }
 else if (buf[0]=='c' && buf[1]=='p')
 {
       if (control==NULL)
       {
        control=client;
    fprintf(stderr,"\nControl program connected\n");
   }
      else
   {
   fprintf(stderr,
    "\nERROR: A second control program client has been connected");
   fprintf(stderr,"\n      Closing it...");
   Sprintf(client,"sQ");
   Sclose(client);
   fprintf(stderr,"Closed.\n");
```

```
            }
        }
        else
        {
            fprintf(stderr,
          "\nERROR: Unknown client has been connected<%s>",buf);
         fprintf(stderr,"\n      Closing it...");
         Sprintf(client,"sQ");
         Sclose(client);
         fprintf(stderr,"Closed.\n");
        }
      } // End if(Stest(client))
      else
      {
       fprintf(stderr,
         "\nERROR: Unknown client has been connected");
       fprintf(stderr,"\n      Closing it...");
       Sprintf(client,"sQ");
       Sclose(client);
       fprintf(stderr,"Closed.\n");
      }
    } // End if(client)
    else fprintf(stderr,"\nError accepting client\n");

    fprintf(stderr,"\n\n\nClients:");
    if(sensor) fprintf(stderr,"\nPC sensor controller");
    if(control) fprintf(stderr,"\nControl software");
    for(i=0;i<MAXSRCLIENTS&&sread[i];i++);
    if(i) fprintf(stderr,"\n%d Sensor-read clients");
    fprintf(stderr,"\n");
}

int yn()
{
    char c;
    c=getc(stdin);
    if(c=='y'||c=='Y')
    {
            fprintf(stderr,"Y");
            return 1;
    }
    fprintf(stderr,"N");
    return 0;
}

void get_cap_val()
{
    char cmd[MAX_BUF];
    int i,junk,mux1,mux2,mux;
    int dcap_value[10];

    cmd[0]=cmd[1]=' ';
    buf[0]=buf[1]=' ';
    Sgets(buf,MAX_BUF,sensor);
```

```
 sscanf(buf,"%s",cmd);
 switch(cmd[0])
 {
        case 's':
        toutcount=0;
        switch(cmd[1])
        {
          case 'k':  /* Read the sensor values from the PC client */
                sscanf(buf,"%s %d %d %d %d %d %d %d %d %d",
                        cmd,&junk,&dcap_value[0],&dcap_value[1],
                        &dcap_value[2],&dcap_value[3],
                        &dcap_value[4],&dcap_value[5],
                        &dcap_value[6],&dcap_value[7],&dcap_value[8]);
                for(i=0;i<9;i++) cap_value[i]=dcap_value[i];
                capflag=1;
                break;

          case 'Q': /* Shut down */
                run=0;
          case 'q': /* Close client */
                sensor_socket=0;
                break;

          default:
                break;
        }
        break;

    case 'r':
        switch(cmd[1])
     {
          case 's': /* Mux change handshake */
            sscanf(buf,"%s %d %d %d",cmd,&mux1,&mux2,&mux);
            ss_hs=1;
            sprintf(s_hs,"rs %d %d %d",mux1,mux2,mux);
            break;
          default:
            break;
         }
      break;

        default:
        break;
 }

}

void get_control_cmd()
{

 cmd[0]=cmd[1]=' ';
 buf[0]=buf[1]=' ';
 Sgets(buf,MAX_BUF,control);
 sscanf(buf,"%s",cmd);
```

```
 switch(cmd[0])
 {
     case 'c':
      switch(cmd[1])
      {
           case 'c': /* Clear limit cycling flag on PC */
                     if(sensor) Sprintf(sensor,"sk");
                     break;
        case 'e':  /* Request sensor data */
                     sendflag=1;
                      break;

        case 's':  /* Change mux channel command */
                     sscanf(buf,"%s%d%d%d",cmd,&mux1,&mux2,&mux);
                     muxflag=1;
                     break;
        case 'g': /* Send ( to PC ) gain used in control program */
                     sscanf(buf,"%s%d%lf",cmd,&sensnum,&con);
                     const_flag=1;
                     break;

        case 'Q': /* Shut down */
                     run=0;

        case 'q': /* Close client */
                     control_socket=0;
                     break;

        default:
                     break;
      }
      break;

      default:
        fprintf(stderr,"[%s]",buf);
        break;
 }
}
void get_sr_cmd(int num)
{
 cmd[0]=cmd[1]=' ';
 buf[0]=buf[1]=' ';
 Sgets(buf,MAX_BUF,sread[num]);
 sscanf(buf,"%s",cmd);
 switch(cmd[0])
 {
     case 'c':
      switch(cmd[1])
      {
       case 'e':
          srsflag[num]=1;
          break;

       case 'Q':
```

```
            fprintf(stderr,
              "\nSensor read client %d tryed to terminate me!",num);
           case 'q':
             sr_socket=0;
             break;

           default:
             break;
          }
          break;

        default:
          fprintf(stderr,"[%s]",buf);
          break;
    }
    while(Stest(sread[num])>0) Sgets(buf,MAX_BUF,sread[num]);
}
```

Appendix 4. Gripper software.

```
/*
 *      FACILITY:
 *              NASA Goddard Robotics Lab
 *              Engineering Test Bed
 *
 *                      ABSTRACT:
 *              Subroutines to send commands to the toolbox gripper.
 *
 *      DESIGN DECISIONS:
 *   Subroutines transmit_byte2(), get_byte2_ready_status(), get_byte2(),
 *   and init_port_2() were copied from rrc_comm.c.
 *
 *      MODIFICATION HISTORY:
 *              1 June 1993             Paulo Uribe, CUA
 *                      Created
 */

#include "rrc_grpr.h"

extern Socket *rrc_socket;
extern int gripcmd;

int initialize_gripper(void)
{
        int position,force,speed;
        init_port_2();
        return !send_command("MP3000",&position, &force, &speed);
}
void read_port(void)
{
        int timeout=5000;
        printf("\rWaiting for gripper...Press any key if suspected crash in the little giant");
        while(get_byte2_ready_status()!=DATA_READY && !kbhit());
```

```
        if(kbhit()) getch();
        printf("\r                                                    ");
        while(get_byte2_ready_status()==DATA_READY)
        {
                get_byte2();
                timeout=1000;
        printf("\rWaiting for next char...Press any key if suspected crash in the little giant");
                while(get_byte2_ready_status()!=DATA_READY && timeout &&!kbhit()) timeout--;
                if(kbhit()) getch();
        printf("\r                                                    ");
        }
        printf("\rTimeout=%d",timeout);
  return;
}
int send_command(char *c,int * position, int * force, int * speed)
{
        register i;
        int error,n,timeout=0;
        char r[15];

        gripcmd=0;
        for(i=0;i<6 && c[i];i++);
        if(i!=6 && c[i]) { printf("Gripper command error <%s> ",c); return 255; }
        for(i=0;i<6;i++) transmit_byte2(c[i]);
        Sprintf(rrc_socket, "OK");
        printf("GRP[%d]",i);
        gripcmd=1;
        i=0;
        while(i<2 && timeout<700)
        {
                if (get_byte2_ready_status()==DATA_READY)
                {
                  r[i]=get_byte2();
                i++;
                        timeout=0;
                }
          timeout++;
        }
         if(timeout==700) { printf("Gripper time out "); return 255; }
        n=2;
        if(r[0]=='G') n=10;
        else if(r[1]=='P') n=6;
        while(n && timeout<500)
        {
                if (get_byte2_ready_status()==DATA_READY)
                {
                  r[i]=get_byte2();
                i++;
                        n--;
                        timeout=0;
                }
          timeout++;
        }
        r[i]=0;
```

```
        if(c[0]==r[0] && c[1]==r[1])
        {
                switch(r[0])
                {
                        case 'F':
                                return r[3];
                        case 'G':
                                error=atoi(r+10);
                          r[10]=0;
                                *position=atoi(r+6);
                                r[6]=0;
                                *force=atoi(r+2);
                                return error;
                         case 'M':
                                switch(r[1])
                                {
                                        case 'E':
                                                error=atoi(r+2);
                                                return error;
                                        case 'P':
                                                error=atoi(r+6);
                                                r[6]=0;
                                                *position=atoi(r+2);
                                                return error;
                                }
                        case 'S':
                                error=atoi(r+6);
                                r[6]=0;
                                *speed=atoi(r+2);
                                return error;
                        default:
                                return 11;
                }
        }
        else
        {
                return 12;
        }
}


/*****************************************************************
*
* Procedure transmit_byte2()
*
* This procedure will transmit the data byte passed.
*
*
*****************************************************************/
int transmit_byte2(char data)
{
long timeout = 0L ;
```

```
        while (!(inp(LSR_Addr_Port_2) & 0x20))
                if (timeout++ == 100000)
                        return(XMIT_TIMEOUT);

  outp(TXD_Addr_Port_2, data) ;
  printf("[%c]",data);
        return(XMIT_OK) ;
}

/*****************************************************************
*
* Procedure get_byte2_ready_status()
*
* This procedure will receive a data byte from the input com port
* if one is available.  If not available, the return status will
* reflect it
*
*
*****************************************************************/
int get_byte2_ready_status(void)
{

        if (!(inp(RST_Addr_Port_2) & 0x01))
                return (NOT_READY) ;
        else
          return(DATA_READY) ;
}

/*****************************************************************
*
* Procedure get_byte2()
*
* This procedure will receive a data byte from the input com port
* if one is available.  If not available, the return status will
* reflect it
*
*
*****************************************************************/
unsigned char get_byte2(void)
{
        return((unsigned char) inp(RXD_Addr_Port_2)) ;
}

/*********************************************************
*
* Procedure init_port_2()
*
* Initilize Port 1 82510 USART to 8 bits no parity
*
* Divisor = 18432000/2/16/baud
* BAL = 0x0a for 56.7 Kbaud
*     0x3c for 9.6  Kbaud
*     0xf0 for 2.4  Kbaud
*********************************************************/
```

```
void init_port_2(void)
{
  outp(GIR_Addr_Port_2,Bank_One) ;
  outp(ICM_Addr_Port_2,0x10) ;
  outp(GIR_Addr_Port_2,Bank_Zero) ;
  outp(LCR_Addr_Port_2,0x80) ;
  outp(BAH_Addr_Port_2,0) ;
  outp(BAL_Addr_Port_2,0xf0) ;        /* set baud rate, see above */
  outp(LCR_Addr_Port_2,0x07) ;
  outp(GIR_Addr_Port_2,Bank_Two) ;
  outp(TMD_Addr_Port_2,0xc1) ;
  outp(IMD_Addr_Port_2,0x0c) ;
  outp(RIE_Addr_Port_2,0x00) ;
  outp(RMD_Addr_Port_2,0x30) ;
  outp(FMD_Addr_Port_2,0x00) ;
  outp(GIR_Addr_Port_2,Bank_Three) ;
  outp(BACF_Addr_Port_2,0x04) ;
  outp(MIE_Addr_Port_2,0x00) ;
  outp(TMIE_Addr_Port_2,0x00) ;
  outp(CLCF_Addr_Port_2,0x50) ;
  outp(GIR_Addr_Port_2,Bank_One) ;
  outp(TCM_Addr_Port_2,0x02) ;
  outp(RCM_Addr_Port_2,0xb4) ;
  outp(GIR_Addr_Port_2,Bank_Zero) ;
  outp(LCR_Addr_Port_2,0x07) ;
  outp(GER_Addr_Port_2,0x01) ;
}
```