01 May 1988

Computer control of a PBX washout plant

Scott Cameron Sharp

Chung You Ho
Missouri University of Science and Technology

COMPUTER CONTROL OF A PBX WASHOUT PLANT

S. C. Sharp* and C. Y. Ho

CSc-88-4

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401 (314)341-4491

*This report is substantially the M.S. thesis of the first author, completed May, 1988.

ABSTRACT

A fully automated, computer controlled plant has been designed specifically for safe removal of plastic bonded explosives (PBX) from obsolete military munitions. This PBX washout plant consists of a two stage delivery system and robotically operated high pressure waterjet lance. The assigned task was to develop control packages for each component.

The first stage of the delivery system is a battery operated overhead trolley. Its control package consist of a dedicated computer, DC motor and custom positioning subprograms. The dedicated computer communicates through an infrared link to the operator's computer. This link was developed due to requirements of a hazardous environment. Probable software solutions for the communication are presented given current system configurations.

The trolley positions the munition directly above a hydraulically operated table - the second stage of the delivery system. This stage's control package incorporates three closed loop, first order circuits and software. The table positions the munition for explosive removal by the waterjet.

The robotic waterjet lance is hydraulically controlled through four closed loop, first order circuits. The concepts of its controlling software is presented to better understand the currently developed software.

ACKNOWLEDGEMENTS

I would like to thank Dr. David Summers for the opportunity to assist Mr. John Tyler and Mr. Jen Sriwattanathamma in both the new conceptual phase and ongoing technical development phase of the PBX Washout Plant.

I am grateful for the patience and constant support of my thesis advisor, Dr. Peter C. Y. Ho, for without his efforts this thesis would not have been completed. I am also indebted to Dr. Arlan DeKock and Dr. Kelvin Erickson for their comments on initial drafts, and for their technical insight while serving as thesis committee members.

In addition, I would like to extend my warmest thanks to the faculty and staff of the Rock Mechanics and Explosive Research Center for the financial support, and encouragement they have so consistently offered to me. Finally, a special thanks to my family for their unending support and understanding throughout my educational career.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I. INTRODUCTION

The Rock Mechanics and Explosive Research Center (RMERC) of the University of Missouri-Rolla (UMR) has been conducting research for the Naval Weapons Support Center of Crane, Indiana. The problem deals with the disposal of a stockpile of obsolete military munitions. The munitions must be disposed of in an environmentally safe fashion which rules out detonation or burial. RMERC has previously determined that a high pressure water jet can remove the explosive material from inside a munition shell. The explosive materials are held in the shells with plastic binders. The removed explosive material can be separated from the plastic binders through chemical separation. The explosive material can then be reused. The research is being continued to the point of designing a pilot plant that will automate the process of transporting a variety of different sizes and types of shells to a robot that will wash out the explosive material inside a shell. The pilot plant is known as the Plastic Binded Explosive (PBX) Washout Plant[1,2]. This thesis presents software solutions for controlling the components of the PBX washout plant.

It is assumed that the reader has a knowledge of general computer hardware concepts. A knowledge of Borland's Turbo Pascal[3] and the BASIC language is also assumed. The BASIC language used in this thesis is Analog Device's MacBASIC[4,5,6]. This version of BASIC has some extended capabilities that will be presented as needed.

A. PBX WASHOUT PLANT

The PBX washout plant is centered around a computer controlled hydraulic robot that manipulates a high pressure water jet. This robot is known as the Waterjet Ordnance and Munition Blastcleaner with Automated Tellurometry (WOMBAT) [1,2]. The computer sends position commands to electronic circuits that control hydraulic valves used to move the WOMBAT. The water jet itself is not computer controlled.

The WOMBAT system has been previously developed and has been used but without any type of automated delivery process. With this system, the plant operator manually connects a shell to the WOMBAT's faceplate for each washout. This loading and unloading process is time consuming and more importantly, dangerous. The process of automating the PBX washout plant includes developing a delivery system that will connect any of the shells specified in the PBX washout plant contracts to the WOMBAT. Each type of shell has a faceplate adapter.

The delivery system consists of two components. The first is a Monorail system[1,2] and the second is the Table system[1,2]. The monorail system consists of an overhead monorail track and a carrier that moves on the track with a computer controlled electric motor. The carrier is known as the monorail carrier. Figure 1 shows the configuration of the delivery system components and the WOMBAT. This and other drawings of the components of the PBX washout plant are availible at RMERC.

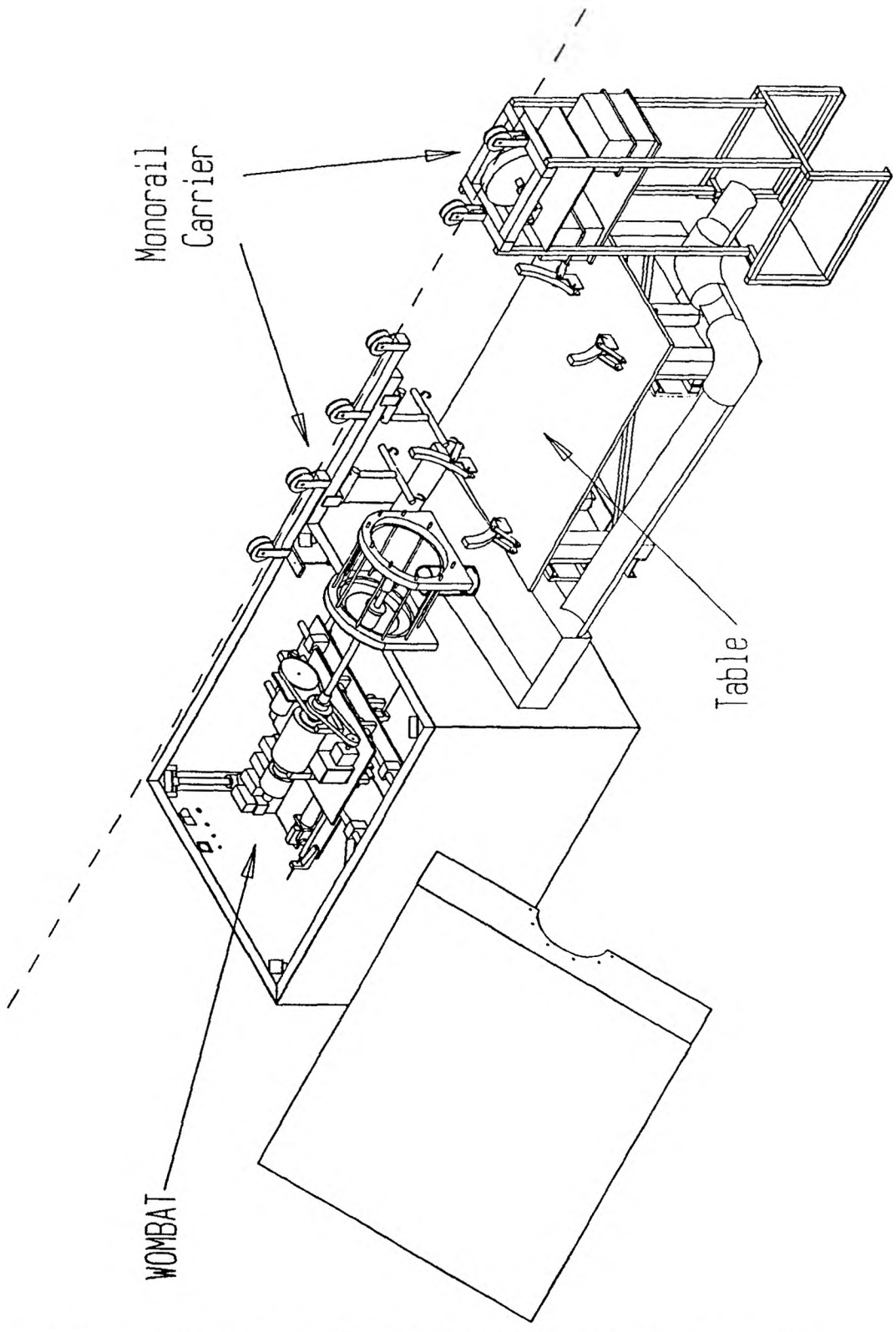

Figure 1. Delivery System Components and the WOMBAT.

The washout process is performed inside one of the UMR Experimental Mines. The monorail track leads to the WOMBAT. A shell is loaded on the monorail carrier at the end of the track furthest from the WOMBAT. This area is known as the loading station. The monorail carrier delivers the shell to the area in front of the robot which is known as the washout area. Figure 2 contains an overhead diagram of the PBX washout plant.

The second component, the Table system, consists of a hydraulic table with a grasping mechanism to hold a shell. It is controlled in a manner similar to the WOMBAT's control. After a shell has been delivered to the washout area by the monorail system it is connected to the WOMBAT by the Table system and is ready for the washout process. The delivery system also removes the previously washed out shell. The washed out material is collected in a container on the monorail carrier during the washout. The container and the washout material also return to the loading station.

The final step of automating the PBX washout plant will be an explosive reclaiming system for the washed out material. This system is known as Remotely Operated Binder Extraction Technique (ROBET)[1,2]. It chemically separates the explosive material from the plastic binders. A small scale version of ROBET has been developed at RMERC. It has proven to be efficient. In the current stages of the PBX washout plant project, the washed out materials and cleaned shells are returned to the loading station where the washout

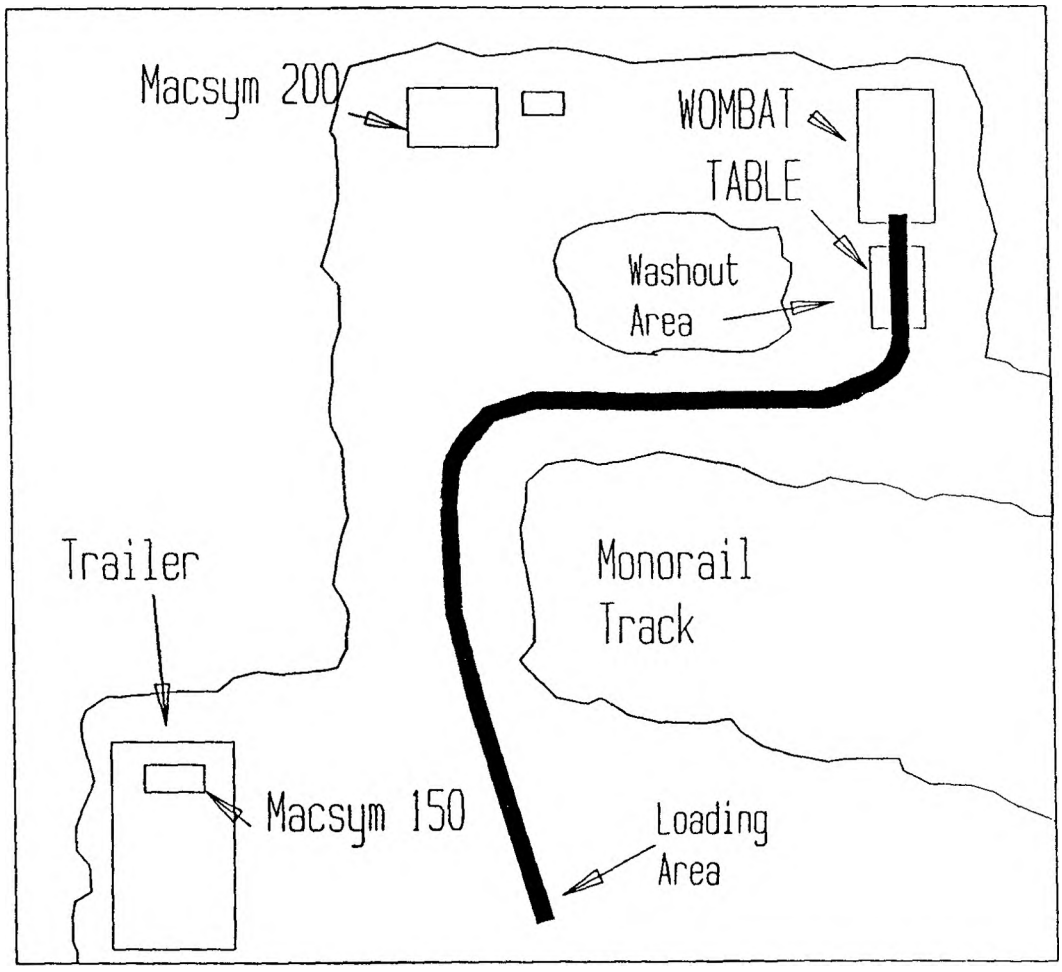

Figure 2. Overhead View of PBX Washout Plant.

material is saved and analyzed later. In future stages of the PBX washout plant, the washed out material is to be taken to a large scale ROBET system. It will be located inside the mine along an extension of the monorail track. The control of the ROBET system is not covered in this thesis.

B. COMPUTER SYSTEMS

Two computer systems are used on the automated PBX washout plant. The first computer is the plant computer. It controls the WOMBAT, the Table and the Monorail systems. It will eventually control the ROBET system in the final stage of automation of the PBX pilot plant. The second computer, the monorail computer, aids the plant computer in the control of the monorail system. It is dedicated to receiving motor commands and data from the plant computer and sending position data back.

The plant computer is an Analog Devices Macsym 350 system. It consists of two body parts: the Macsym 150[7] and the Macsym 200[8]. The Macsym 150 is the main computer component of the system. It contains slots in which RS-232 cards[9] can be placed. The Macsym 200 is an external card cage component that is connected to the Macsysm 150.

The Macsym 200 can contain up to sixteen Analog/Digital Input/Output (ADIO) cards. The Macsym 200 used at the PBX washout plant contains a four channel, twelve bit digital to analog (D/A) output card[10] and a 32 channel analog to digital (A/D) input card[11].

The plant operator interacts with the Macsym 150 unit which is located outside of the mine in a trailer. The Macsym 200 is located in the mine near the WOMBAT and Table systems where it controls their hydraulic systems. The plant operator controls the PBX washout plant with the assistance of several video monitors showing different views of the plant.

The Macsym system is programmed in an extended version of BASIC known as MacBASIC. MacBASIC contains special instructions for the extended capabilities of the Macsym 200's ADIO cards. It is a line number based language; the sequence of execution is oriented by the next greatest line number. It has graphics capabilities that may be used to plot positions from various feedback channels.

The second computer is the monorail computer. It must be small in size because it rides on the monorail carrier. An IBM XT compatible computer[12-17] is desirable because of the availablity of hardware[18,19,20] to control the monorail carrier's motor. A Micro PC system was purchased from Faraday Electronics Incorporated, Sunnyvale, California[16]. Its physical design is different from that of an IBM XT and most IBM XT compatibles. Many of the integrated circuits from an IBM XT have been combined into a single large scale integrated circuit chip. With this chip, the entire motherboard is replaced by a short card that fits into a six slot backplane. The backplane contains this computer card plus add-on cards normally found in an IBM XT computer.

The monorail computer is programmed in Borland's Turbo Pascal Version 3.0. This is a procedural language that has extended capabilities that allow access to hardware ports. The programs can be compiled into machine code that is executable from the Disk Operating System (DOS)[3,21].

The monorail computer controls the electric motor with the use of a Hewlett Packard (HP) HCTL-1000 motor control processor[20,22]. The HP HCTL-1000 is contained on a board external to the monorail computer. The board is connected to the monorail computer through an external bus system. A second external board is also connected to the monorail computer through the external bus system. This board is an A/D input system that aids the Table system by measuring the load on the monorail carrier.

The two computer systems communicate over an infrared communication link[1,2]. This link actually replaces a RS-232 cable. Each computer's RS-232 port is connected to its own infrared transmitter and receiver. The communication between the transmitters and the receivers occurs through the open air without any physical connections.

C. PLANT ALGORITHM

A shell that is to be washed out at the PBX washout plant is first strapped to the monorail carrier at the loading station. Once the shell is strapped to the carrier and a faceplate adapter is mounted to the shell, it is ready to be transported to the washout area. Values which represent positions along the monorail track are sent to the

HP HCTL-1000. The monorail carrier will perform the move when instructed by the plant operator. Video monitors help ver-ify the final position of the shell when the washout area is reached. The shell can now be connected to the WOMBAT.

The Table system is used to connect the shell to the WOMBAT. The table will rise and grasp the shell. It will then move the shell until the shell is firmly pressed against the faceplate of the WOMBAT. This completes the delivery and connection steps of the washout process. The table will keep the shell pressed firmly to the faceplate until the washout has been completed.

The shell is now ready to be washed out by the WOMBAT. The washout is performed by moving the water jet in circular paths at increasing depths into the shell. If, at any time during the washout, the operator believes there might be a problem, an abort button can be pressed which immediately halts the WOMBAT's movement. This feature is important because of dangerous problems that may occur during the washout process.

After the shell is washed out, it can be disconnected from the WOMBAT. The front of the table lowers, tipping the shell to drain any excess material. The shell is then released to the monorail carrier. The monorail carrier returns the shell to the loading area along with the removed material which was collected during the washout.

II. WOMBAT CONTROL

The WOMBAT is a hydraulically powered 3-axis robot that is computer controlled. Its purpose is to manipulate a shaft with a high pressure water jet nozzle on its end. The shaft and the nozzle are known as the lance. The WOMBAT is enclosed in a large metal case with the lance protruding perpendicularly from the case and pointing along the path of the monorail track. The shaft of the lance will rotate during the circular paths of a washout. This allows the spray of the water jet from the lance, which is slightly angled, to reach a broader area.

A. COORDINATE SYSTEM.

The coordinate system of the WOMBAT[2] is defined as a right hand coordinate system with respect to the lance. The Z coordinate of the WOMBAT is defined by the direction of spray from the lance. With the middle finger in the Z direction, the X coordinate, defined by the thumb, is parallel to the ground. The Y coordinate, defined by the index finger, is perpendicular to the ground.

The reach of the WOMBAT's lance is the three dimensional range that contains all possible positions of the tip of the lance. The volume of the three dimensional range is a box with all sides either parallel or perpendicular to the case of the WOMBAT. The range in the Z direction is negative to positive eight inches. The X and Y coordinates have a range of negative to positive four inches. The home position, (0,0,0), is at the center of the box. The lance can

be moved in any combination of moves in the three coordinate directions.

The movements of the WOMBAT are performed by hydraulic rams. A ram moves in one dimension. There are three sets of rams on the WOMBAT which are positioned to control the movement of the lance in the three coordinate directions. The flow of hydraulic fluid causes a movement at the end of a ram. Hydraulic valves are used to control the flow of fluid in a hydraulic ram. The hydraulic valves used on the WOMBAT system are controlled by a voltage signal.

B. POSITION CONTROL.

A four channel, twelve bit D/A output card is used to control the movement of the WOMBAT's rams. The output voltage signals are connected to the circuits that control the hydraulic system of the WOMBAT. This card's voltage range is positive to negative ten volts. The range is divided into 4096 subintervals giving a resolution of approximately 0.005 volts.

The output voltages are used as a position command. A positive (negative) ten volt output instructs a move to approximately positive (negative) eight inches along the Z direction and approximately positive (negative) four inches in the X and Y directions. Intermediate positions are obtained with appropriately scaled voltage commands.

The special MacBASIC keyword AOT(4,5,6) controls the four channels of voltage on a D/A card. This keyword receives a value in an assignment statement which is the

voltage at a channel. It requires two parameters to define a channel. The first parameter is the slot within the Macsym 200 that contains the D/A card. This parameter should remain constant in programs that use the D/A card because the card is generally not moved to a new slot. An integer variable, W_OUT', is assigned the value for the slot containing the WOMBAT's D/A card in example programs presented later.

The second parameter of the AOT keyword is the channel on the selected card. Four integer variables are used in the example programs to give symbolic names to each channel's use on the WOMBAT system. These variables are Xaxis', Yaxis', Zaxis' and ROT'. MacBASIC uses the apostrophe character (') after a variable name to indicate an integer variable. The first three variables represent the channels that control the respective axes of the WOMBAT. ROT' represents the channel that controls the rotation of the lance. For example, the statement

```
AOT (W_OUT',Xaxis') = 0
```

is used to set the lance to the Home position in the X direction.

It is often desirable to specify a movement of the lance in inches. Conversion factors that convert inches to the voltage representing the position are used in the examples presented later. The real variables CONV_X, CONV_Y, CONV_Z and CONV_RPM are set to the values for the

conversion factors of the three axes and the rotation respectively. For example, the statement

AOT (W_OUT',Yaxis') = 4 * CONV_Y

is used to set the Y direction of the lance to four inches. The values of the conversion factors are not the values 2.5 (10 volts / 4 in) for the X and Y channels and 1.25 (10 volts / 8 in) for the Z channel because of physical offsets in the WOMBAT. The actual values for the conversion factors were derived from physical measurements of WOMBAT positions at different position commands.

C. POSITION MONITORING.

The circuits controlling the valves receive a second signal from a Linear Variable Differential Transducer (LVDT) which is used to measure the position of a ram. The voltage read from the circuits will be within the negative to positive ten volt range like the D/A card. The LVDT actually have an AC signal. The control circuits convert the LVDT's AC signal to the negative to positive ten volt range before it is read by the A/D card. This allows the command position output range and actual position input range to be identical.

The MacBASIC keyword AIN(4,5,6) is used to read a voltage from a LVDT. This keyword returns a value to a real variable in an assignment statement. The value returned represents the voltage read at one of the A/D card's channels. This keyword requires four parameters. The first two parameters are the same as the AOT keyword; the slot and

channel of a A/D card. The integer variable W_IN', which holds the number of the slot of the WOMBAT's A/D card, is used in the examples presented later. The same integer variables Xaxis', Yaxis', Zaxis' and ROT' are used for the channels of the three axes and the rotation respectively.

The last two parameters of the AIN keyword should always be zero when used on the WOMBAT system. As an example of the use of the AIN command, the statement

```
Z_FBACK = AIN (W_IN',Zaxis',0,0)
```

is used to read the voltage representing the position of the lance in the Z direction. The conversion factors, CONV_X, CONV_Y, CONV_Z and CONV_RPM, still apply but their reciprocals must be used. For example, the statement

```
ROT_RPMs = AIN (W_IN',ROT',0,0) / CONV_RPM
```

is used to read the rotation in revolutions per minute.

The rotation feedback does not come from a LVDT. Instead, a circuit will convert the signal from an encoder to the negative to positive ten voltage range. The physical range of the motor performing the rotation is approximately 200 revolutions per minute (RPM) in either direction. A positive (negative) ten volt signal either sent to the D/A output channel or received from the A/D input channel represents approximately positive (negative) 200 RPM.

Given the command position signal from the D/A card and the actual position signal from the LVDTs, the circuits control the hydraulic valves such that the ram will move to

the correct position and remain there. These circuits use closed loop, first order control theory.

D. SAMPLE PROGRAMS.

This section presents examples of software that can control the WOMBAT. The actual programs[23] are much more extensive than the examples presented here. The purpose of this section is to aid in the understanding of the control of the WOMBAT. The first example program performs the basic movement of the lance that occurs during a washout. It is expanded upon in the second example program to include the abort feature.

The typical path of the lance during a washout involves several circular paths in a plane perpendicular to the Z axis at increasing depths into the shell. Given the radius, r, of the circular path, and a position in the Z direction, the lance is moved in a circular path in the X and Y directions. This is performed by dividing the path of the circle into m equal distance points on the circle. This defines a path that approximates the circle. The number of points, m, defining the circle should be large enough to define a smooth path without surpassing the resolution of the D/A card.

The lance must be fully retracted in the Z direction before a shell is mounted to the WOMBAT. This should be done by the delivery programs. Both example programs require this to have been previously performed before they

are executed. Figure 3 contains the code to perform a basic washout.

```
10 @@@  preset constants   @@@
20 ROT' = 3   CONV_RPM = 0.1
30 W_OUT' = 1  Xaxis' = 0 Yaxis' = 1 Zaxis' = 2
40 CONV_X = 2.098  CONV_Y = 2.098  CONV_Z = 1.208

50 @ Z position starts fully retracted
60 AOT (W_OUT',Zaxis') = -10

70 INPUT "RADIUS OF SHELL" r
80 INPUT "NUMBER OF SUBMOVES DURING CIRCULAR PATH " m
90 INPUT "STARTING DEPTH INTO SHELL " Z_START
100 INPUT "FINAL DEPTH INTO SHELL " Z_FINAL
110 INPUT "NUMBER OF STEPS INTO SHELL " n
120 INPUT "Rev / Min " rpm

130 SUB_Deg = 2*3.1416/m  @ 2Pi/m
140 Z_STEP = (Z_FINAL - Z_START) / n

150 @ MOVE TO (r,0,-8)
160 AOT (W_OUT',X) = r * CONV_X
170 AOT (W_OUT',Y) = 0
180 @ start rotation and move to (r,0,Z_start) @
190 AOT (W_OUT',ROT') = rpm * CONV_RPM
200 AOT (W_OUT',Z) = Z_START * CONV_Z

210 FOR I = 1 TO n      @ LOOP FOR Z STEPS
220    FOR J = 1 TO m   @ LOOP FOR CIRCULAR STEPS
230       Theta = J * SUB_Deg  @ DETERMINE ANGLE OF NEXT SUBSTEP
240       AOT (W_OUT',Xaxis') = r*COS(Theta)*CONV_X  @ COMPUTE X FROM THETA
250       AOT (W_OUT',Yaxis') = r*SIN(Theta)*CONV_Y  @ COMPUTE Y FROM THETA
260       NEXT J
270    AOT (W_OUT',Z) = Z_START + I*Z_STEP
280    NEXT I
290 AOT (W_OUT',ROT') = 0
300 AOT (W_OUT',Z) = -8 * CONV_Z
310 END
```

Figure 3. Basic WOMBAT Program.

With a shell mounted to the faceplate of the WOMBAT, the lance is first moved in the X direction by the distance of the radius, r, and remains at zero in the Y direction. This defines the starting position of the circular path.

The washout starts by spinning the lance at a specified RPM, moving the lance to the starting Z distance and performing the first circle path in that Z plane. At the end of the circular path, the lance has returned to the starting point of the circular path. The lance then moves in the Z direction, which is into the shell, by the Z increment and another circular move is performed. This process of circular paths at increasing depths continues until the circular path at the final Z distance has been completed. After the final circular path, the lance fully retracts in the Z direction and the rotation stops. Assistant plant operators control the high pressure water jet system during the washout.

A washout program should include an abort feature that will stop the movement of the lance immediately. This involves detecting an abort signal and leaving the loop that is performing the movement of the lance. Figure 4 contains the second example program which shows the basic structure of the abort feature used on the actual WOMBAT program. It allows the lance to back up a specified number of substeps along the circular path when an abort signal is detected. The washout can continue only after the lance has completed the backup movement.

The abort service code allows the washout program to stop executing without interrupting the program at the keyboard. A method of program termination is required because a Control-B, which interrupts the execution from the

keyboard, will cause the Macsym 200 to reset all boards in it. This will set all D/A outputs to zero which causes the lance to jump to the home position. Serious problems can occur to certain types of shells which have a solid pipe through their center or shells that still have explosive material in the negative Z direction. The lance can slam into the pipe when it jumps to the home position or into explosive material. A Control-B should only be performed when the WOMBAT's hydraulic pump is turned off.

```
10 @@@  preset constants   @@@
20 W_IN' = 0  W_OUT' = 1
30 Akey' = 4   ROT' = 3
40 Xaxis' = 0 Yaxis' = 1 Zaxis' = 2
50 CONV_X = 2.098  CONV_Y = 2.098
60 CONV_Z = 1.208  CONV_RPM = 0.1

70 INPUT "RADIUS OF SHELL" r
80 INPUT "SUBMOVES PER CIRCULAR PATH " m
90 INPUT "STARTING DEPTH INTO SHELL " Z_START
100 INPUT "FINAL DEPTH INTO SHELL " Z_FINAL
110 INPUT "NUMBER OF STEPS INTO SHELL " n
120 INPUT "Rev / Min " rpm

130 BegDeg = 0  EndDeg = 2*3.1416
140 SUB_Deg = EndDeg/m
150 Z_STEP = (Z_FINAL - Z_START) / n
160 PasStart = Z_START   PassEnd = Z_FINAL

170 AOT (W_OUT',X) = r * CONV_X
180 AOT (W_OUT',Y) = 0
190 AOT (W_OUT',ROT') = rpm * CONV_RPM
200 AOT (W_OUT',Z) = Z_START * CONV_Z

210 @ @ @  start/continue washout  @ @ @
220 GOSUB 260 @ CIRCULAR PATHS
230 AOT (W_OUT',Z) = -8*CONV_Z
240 AOT (W_OUT',ROT') = 0
250 END

260 @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
270 @ @ @   CIRCULAR PATHS    @ @ @
280 @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
290 FOR Pass = PasStart TO PassEnd  STEP Z_STEP
300    FOR Theta = BegDeg TO EndDeg STEP SUB_deg
310       AOT (W_OUT',Xaxis') = r*COS(Theta)*CONV_X
320       AOT (W_OUT',Yaxis') = r*SIN(Theta)*CONV_Y
330       ABORTkey = AIN (W_IN',Akey',0,0)
340       if abs(ABORTkey) > 2 then
350          PasStart = PASS BegDeg = THETA
360          RESET FOR  GOTO 410 @ABORT_SERVICE
370       NEXT Theta
380    AOT (W_OUT',Z) = Pass * CONV_Z
390    NEXT Pass
400 RETURN

410 @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
420 @ @ @   ABORT_SERVICE   @ @ @
430 @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @
440 INPUT "NUMBER OF SUBSTEPS TO BACKUP " Bstep
450 ! "*** RESET ABORT KEY BEFORE BACKING UP "
460 INPUT "Enter Y to backup, N to stop Washout" X$
470 IF X$ <> "Y" THEN END

480 PassEnd = PasStart  SUB_deg = -abs(SUB_deg)
490 EndDeg = BegDeg-(Bstep*SUB_Deg)
500    GOSUB  260 @ CIRCULAR PATHS
510 PassEnd = Z_FINAL  SUB_deg = abs(SUB_deg)
520 BegDeg = EndDeg  EndDeg = 2*3.1416
530 INPUT "Enter Y to continue washout " X$
540 IF X$ = "Y" then GOTO 210 @ continue washout
550 GOTO  410 @ ABORT SERVICE
```

Figure 4. WOMBAT Program with Abort/Backup/Continue.

The abort button used at the pilot plant is connected to a channel of the A/D input card. The integer variable Akey' represents this channel. The slot containing the A/D card is still represented with W_IN'. As long as the button is not pressed, the voltage read from the abort channel will be near zero. When it is pressed, the voltage will be near ten volts (possibly negative ten volts if the button's connection is reversed). By checking the value read at this channel before the statements performing the substep moves, the abort condition can be detected when the absolute value of the voltage at this channel rises above a threshold slightly above zero. The threshold has been set at a value representing two volts in the example.

The segment of code containing the loops to perform the lance movement is a subroutine. This allows the abort service code to change the parameters of the loops and reexecute the subroutine to move the lance backwards along the circular path. The parameters are then restored so the remainder of the washout path will finish.

III. MONORAIL SYSTEM CONTROL

The Monorail system is part of the shell delivery system of the pilot washout plant. It consists of an overhead rail and a carrier. The carrier is powered by an electric motor. It receives power from two batteries located on the carrier. Precise control of the motor is necessary because the carrier is moving explosive material. The monorail computer is used to control the motor. It is located on the carrier and has the ability to communicate with the plant computer.

A. MOTOR CONTROL.

The external bus system that connects the monorail computer to the HP HCTL-1000 motor control processor is known as the Local Application Bus (LAB) 40 System [18,19,20]. The HP HCTL-1000 is on an external function module. A ribbon cable is used to connect the HP HCTL-1000 function module to the plant computer. A second external function module is also connected to the plant computer on the ribbon cable. This function module is an A/D input system. It aids the table system by measuring the load of the monorail carrier. It will be discussed in the Table System Control section.

The HP HCTL-1000 performs position-to-position motor moves through trapezoidal velocity profiling. A motor move starts by accelerating at a specified acceleration rate until it reaches a specified maximum velocity. The motor will remain at the maximum velocity until the HP HCTL-1000

determines that the motor must decelerate so that the motor will stop at the final position. If the velocity of a trapezoidal profile motor move is plotted over time, its profile should be a trapezoid or possibly a triangle. The velocity profile will be a triangle if the motor does not accelerate to the maximum velocity before the HP HCTL-1000 starts decelerating the motor.

The motor is equipped with an optic encoder. The HP HCTL-1000 counts the number of position signals from the optic encoder. The optic encoder sends 2640 position signals per revolution to the HP HCTL-1000. The actual position of the motor depends on the number of encoder positions counted from the starting position. For example, an actual position of 264,000 (2640 * 1000) means the motor has gone through 1000 complete revolutions from a starting position. The direction of the motor revolutions determines whether the position signals add or subtract from the actual position.

The HP HCTL-1000's trapezoidal velocity profile generator determines a command position according to a 2 MHz clock frequency. This is the position the motor should be at as opposed to the actual position of the motor which may not be the same value. The difference between the command position and the actual position of the motor is used to determine the compensation needed to correct the actual position. The motor compensation is performed by varying the sign and magnitude of the motor's voltage.

With the use of constant clock periods, the velocity of the motor is easily computed by the HP HCTL-1000 from the number of position signals per clock period. This is the unit of velocity expected by the HP HCTL-1000. The acceleration is also easily determined from the rate of change of the velocity.

Moving the motor from one position to another involves sending a final position value to the HP HCTL-1000 and instructing the HP HCTL-1000 to start the move. The other parameters of the HP HCTL-1000 that affect a motor move should be set during program initialization and their values should be changed only if different values are required. The position of the motor can be read by the monorail computer from the HP HCTL-1000 during a move. The position information should be returned to the plant computer over the communication link. This information can be used to graph the progress of the move.

The two computer systems communicate over an infrared communication link. This link actually replaces a RS-232 cable. The RS-232 port of each computer is connected to its own infrared transmitter and receiver circuit. Batteries on the carrier along with the infrared communication link allow the carrier to be a self contained unit; computer cables and power lines are not connected to the carrier.

B. MONORAIL PROGRAMS REQUIREMENTS.

Given the configuration of the two computer systems and their purpose, several requirements are placed on the

software for the monorail system. The plant operator should not interact with the monorail computer directly. The monorail computer should receive commands and data from the plant computer and should send information back over the communication link. When the monorail computer is turned on, it should automatically be ready to receive commands from the plant computer. The monorail computer should remain idle until a command is received.

Programs written for the monorail computer require the use of special Turbo Pascal keywords that many programmers do not normally use. These extended capabilities allow access to the hardware ports needed to control the connection of a shell to the WOMBAT and perform the basic means of communication with the plant computer. Subprograms that perform the required hardware manipulation are presented in this section. They allow programs that will control the monorail carrier to be written without understanding the details of the computer hardware. These subprograms will be known as extended keywords (ExKey). The subprograms for the monorail computer are written in Turbo Pascal and the subprograms for the plant computer are written in MacBASIC.

Any program written in Turbo Pascal on the monorail computer for the monorail system will be referred to as a slave program. Any program written in MacBASIC on the plant computer for the monorail system will be referred to as a master program. A slave ExKey is a version for the monorail computer and a master ExKey is a version for the plant

computer. This convention helps clarify which program or version of an ExKey is being discussed.

Two types of ExKeys are presented: communication and motor control. Many of the motor control ExKeys require some data to be sent between the two computers. The ExKeys are presented in pairs; a slave version for the monorail computer, and the counterpart, master version for the plant computer.

The programs for both computers must be coordinated so that either one of the programs will not wait for an event that will never occur. This is known as deadlock. When both computers execute their respective version of an ExKey in a coordinated manner, the deadlock possibilities are greatly reduced.

To use the slave ExKeys, their code must be included in programs on the monorail computer. The code is stored on a disk file. The code can be added to a program with Turbo Pascal's include-file compiler directory, or the file may be read into a program with the read-from-file editor command. There are other subprograms in this file that are called by the ExKeys.

The slave ExKeys are user written procedures. They are executed in a program like other user written procedures. The programmer only needs to know the types of parameters that might be involved. Parameters that are passed by reference will be referred to as variable parameters.

The master ExKeys are also contained on a disk file. The line number structure of MacBASIC makes the task of including their code in programs on the plant computer difficult. To use the master ExKeys, renumber the main program allow space in the line numbering sequence for the ExKeys' code. This program is then saved on disk and the memory cleared. The code for the ExKeys is loaded into memory and renumbered so they will fit into the space left in the main program. The main code is entered back into memory. This will not overwrite any code if everything is numbered correctly.

The master ExKeys are not easily implemented because of the line numbering nature of MacBASIC. The ExKeys are implemented with subroutine calls to a line number. In this section, because the line numbers for the ExKeys are not known, the name, instead of the line number, will be used in a GOSUB command. The starting line numbers for the subroutines will have to be referred to from a listing of the renumber ExKey code for use in any GOSUB statements calling an ExKey. To actually implement an example of code, the line number must be determined and then substituted in place of the name in the GOSUB statement.

Subroutines in MacBASIC can also use parameters. The subroutine code must start with a DECLARE statement followed by the parameter list. The GOSUB statement must refer to the line number of the DECLARE statement. All parameters used in MacBASIC are pass-by-reference.

C. COMMUNICATION EXTENDED KEYWORDS

1. SetUpCom1. The SetUpCom1 ExKey is used to set up the hardware involved with communication. It should only be executed once and before any other communication ExKeys are executed. Parameters are not used by either version of this ExKey.

2. ReadCom1. The ReadCom1 ExKey is used to read a byte of data sent from the other computer. Both versions return the read byte in a variable integer parameter. This ExKey will not return from execution until a byte is received.

3. SendCom1. The SendCom1 ExKey is used to send a byte value to the other computer which should be waiting with a ReadCom1. This ExKey will not return from execution until the byte is read by the receiving computer. Both versions use an integer parameter which must be in the range of zero to 255.

ReadCom1 is the counterpart ExKey for SendCom1. The receiving computer should execute ReadCom1 while the sending computer executes its SendCom1. There is one byte of buffering on both computers which allows some relaxation on the timing of the execution of both ExKeys. The buffering allows a SendCom1 to be performed before a ReadCom1 is performed. Both ExKeys will remain executing until the communication has completed.

A communication scheme or protocol is used by these two ExKeys to insure that a transfer completes correctly or an

error message will appear. The protocol removes the details of possible communication errors that can occur.

The SendCom1 and ReadCom1 ExKeys present a basic means of communication between the two computers. They send and read only one byte at a time. By developing other procedures and functions that use these two ExKeys, more advanced operations can be easily accomplished such as transferring strings of characters or large integer numbers.

D. MOTOR CONTROL EXTENDED KEYWORDS

The motor control ExKeys are presented in this section. The monorail computer should control the motor as instructed by commands it receives from the plant computer. Unless noted otherwise, both versions of an ExKey must be executed because data is transmitted between the two computers. All of these ExKeys do not use parameters. Sample programs will be presented later that should clarify their use.

1. Init Lab40. The Init_Lab40 ExKey is used to initialize the LAB 40 system and the HP HCTL-1000. The slave version has to be performed ber any motor control ExKeys are executed and should be executed only once. The master version is used to set variables that may be used by a master program. Its use is optional. The meaning of these variables will be explained later. The variables set are: PperR, TIME', VEL, ACCEL, GAIN', POLE', ZERO0'. The variable PperR represents the pulses per revolutionofptic encoder, TIME' represents the sampling time periods, VEL represents

the maximum velocity and ACCEL represents the acceleration rate. The variables GAIN', POLE' and ZERO0' represent digital filter parameters.

2. Soft Reset. The Soft_Reset ExKey is used to reset all the position registers in the HP HCTL-1000. The position registers include the Command, Actual and Final positions. The master version will display a message describing the functions performed by this ExKey. The use of the master version is optional.

3. Set Filter Vars. The Set_Filter_Vars ExKey is used to set the HP HCTL-1000's digital filter registers. These values determine the performance and behavior of the motor. The master version prompts the operator for the gain, pole and zero values. Default values can be entered by pressing only the return key for each prompt. The default values were set by Init_Lab40; therefore this ExKey does not require execution. The default values should always be used unless instructed to use other values by someone familiar with the control theory used by the HP HCTL-1000. The variables GAIN', POLE' and ZERO0' are set to the new values on the master version. These variables are discussed in the Motor Control Hardware section.

4. Enter Final Position. The Enter_Final_Position ExKey is used to enter the next position for a motor move. It must be performed before a motor move is executed. The values for position must be in the range of -8,388,608 to

8,388,607. This is the range of a 24 bit signed integer. The units for this position is in position signals from the motor's optic encoder.

The master version prompts for a position and sends it to the monorail computer if it is in the correct range. When a position is entered that is not in range, the operator will be prompted for a new value. The code for master version can easily be changed, with the use of conversion factors, to allow inputs of inches or revolutions to be converted to the position signal counts.

5. Set Accel. The acceleration rate used in a trapezoidal profile motor move is changed with the Set_Accel ExKey. The HP HCTL-1000 was initialized by Init_Lab40 to accelerate at the slowest possible value that is measurable. The master version prompts the operator for the acceleration in revolutions per microsecond squared. The possible range of values is displayed at the prompt. The variable ACCEL is set to the new value by the master version.

6. Set Vel. The value for maximum velocity is changed with the Set_Vel ExKey. The HP HCTL-1000 was initialized by Init_Lab40 to allow a motor move to accelerate until the largest measurable value of velocity is reached. The motor remains at that velocity until the HP HCTL-1000 starts decelerating the motor to its final position. The master version prompts the operator for the velocity in revolutions per second. The possible range of values are displayed at

the prompt. The variable VEL is set to the new value by the master version.

7. Set Time. The value for sampling time periods of the HP HCTL-1000 can be changed with the Set_Time ExKey. The HP HCTL-1000 was initialized by Init_Lab40 with the longest allowable time periods. The time periods determine the allowable ranges of acceleration and velocity. This value should only be changed by someone familiar with the control theory of the HP HCTL-1000. The master version prompts for a value between 15 and 255 with 255 the default value representing the largest possible time period. When a valid value is entered, the master version will display the number of microseconds that the value represents. The variable TIME' is set to the new value by the master version.

8. Display Position. The Display_position ExKey is used to read the actual position of the motor from the HP HCTL-1000 and send it to a master program which displays the number. If a monitor is connected to the monorail computer, the slave version will also display the command position of the motor on that screen. The command position is not sent to the plant computer.

9. Execute Move. The Execute_move ExKey is used to start a motor move. The actual position of the motor is repeatedly sent to the master version while the move is being performed. Display_position is used by this ExKey to display the actual position on the plant computer. A message

stating that the move has completed will be displayed on the plant computer when the motor move has completed and the final position is sent.

10. Quit. The Quit ExKey is used when the monorail system programs are to terminate execution. It deselects the Lab 40 system on the monorail computer. There is not a master version because there is not any hardware that has to be deselected on the plant computer. The only requirement of a master program is that it must be developed so that it instructs the corresponding slave program to execute the Quit ExKey.

E. SAMPLE PROGRAMS.

The ExKeys present a means of controlling the electric motor of the monorail carrier as instructed from the plant computer without understanding the hardware involved. Two sample program pairs are presented in this section that show the use of the ExKeys. The first pair of programs inputs a final position on the plant computer. The move then automatically executes. The second pair of programs is more flexible than the first pair. It uses a menu which allows any of the motor control ExKeys to be executed. The program listings for the first slave program, Slave1, and the first master program, Master1, are presented in Figure 5.

Both programs execute in parallel; Slave1 on the monorail computer and Master1 on the plant computer. The Slave1 program will first initialize the communication hardware of

the monorail computer and set up the LAB 40 system. It will then execute Enter_Final_Position. This ExKey will wait until a position is received from Master1. When the position is received, Execute_Move will be executed. Execute_Move s move and repeatedly sends the actual position to Master1 until the move has completed.

```
program slave1;
   {$I keyword.pas}
begin
SetUpCOM1;
Init_LAB40;
while true do begin
  Enter_Final_Position;
  Execute_Move;
  end;
end.
```

```
5 @  MASTER1 PROGRAM
10 GOSUB SetUpCOM1
20   GOSUB Enter_Final_Position
30   GOSUB Execute_Move
40   GOTO 20
50@ the KEYWORD.BAS file
60@ should follow
```

Figure 5. Slave1 and Master1 Programs.

Meanwhile, Master1 first initialized the plant computer's communication hardware. The Init_LAB40 ExKey does not need to be executed with this simple pair of programs because the variables it sets are not used. Enter_Final_Position prompts the operator for a final position which will be sent to Slave1. When the position is entered, Execute_Move will execute. Execute_Move receives actual position data sent by Slave1 and displays the position values on the screen. A message stating that the move has been completed appears after the last position is received. Figure 6 contains an example of what would be displayed on the plant computers display for a motor move to position 9999.

```
ENTER FINAL POSITION  <-8388608,83886076>  ?9999

0
68
962
2568
4926
7270
8842
9728
9911
9999

  ---  MOVE HAS BEEN COMPLETED  ---

ENTER FINAL POSITION  <-8388608,83886076>  ?
```

Figure 6. Sample Master1 Execution.

This pair of programs allows only the default values for the digital filter, sampling time period, acceleration and velocity to be used. The position registers cannot be cleared because the Soft_Reset ExKey cannot be executed.

The second pair of programs allows all of the capabilities of the ExKeys to be executed. These programs are called Slave2 and Master2. Figure 7 contains the code for the Slave2 and Master2 programs. Master2 presents a menu for the operator to select any of the motor control functions. The motor does not automatically start a move after a final position is entered. The operator must instruct the start of the motor move.

Portions of the ExKeys' code can be modified without understanding the hardware that is involved. For example, many of the subroutines on the plant computer input values from the keyboard but a programmer familiar with MacBASIC can change the input statements to read values from a disk

file. This could allow a sequence of moves to be stored on disk and the sequence could be repeated without reentering the values. The rest of this section presents a description of the hardware involved with the monorail system computers.

```
program slave2;
var  x:  char;  y:  integer;
{$I keyword.pas}
begin
 SetUpCOM1;
 Init_LAB40;
 while true do begin
   ReadCOM1(y);      x := chr(y);
   case x of
     'P' : Enter_Final_Position;
     'E' : Execute_Move;
     'D' : Display_position;
     'V' : Set_Velocity;
     'A' : Set_Acceleration;
     'F' : Set_Filter_variables;
     'T' : Set_Time;
     'S' : Soft_Reset;
     'Q' : Quit;
    else begin end; {INVALID LETTERS IGNORED}
     end;  { CASE }
   end;  { WHILE }
end.
```

```
10  @@@ MASTER2 @@@
20 GOSUB SetUpCOM1  GOSUB Init_LAB40
30    @@@  DISPLAY MENU  @@@
40  ! " P-enter final P)osition"
50  ! " E-E)xecute Move"
60  ! " D-D)isplay Position"
70  ! " V-set V)elocity"
80  ! " A-set A)cceleration"
90  ! " F-set F)ilter variables"
100 ! " T-set T)ime"
110 ! " S-S)oft reset"
120 ! " Q-Q)uit"
130@ ENTER/SEND COMMAND TO Slave2
140 INP A'
150 GOSUB SendCOM1 (A')
160 A$ = CHR$(A')
170 IF A$ = "P" THEN
        GOSUB Enter_Final_Position
        GOTO 30  @ DISPLAY MENU
180 IF A$ = "E" THEN
        GOSUB Execute_Move
        GOTO 30  @ DISPLAY MENU
190 IF A$ = "D" THEN
        GOSUB Display_Position
        GOTO 30  @ DISPLAY MENU
200 IF A$ = "V" THEN
        GOSUB Set_Velocity
        GOTO 30  @ DISPLAY MENU
210 IF A$ = "A" THEN
        GOSUB Set_Acceleration
        GOTO 30  @ DISPLAY MENU
220 IF A$ = "F" THEN
        GOSUB Set_Filter_Variables
        GOTO 30  @ DISPLAY MENU
230 IF A$ = "T" THEN
        GOSUB Set_Time
        GOTO 30  @ DISPLAY MENU
240 IF A$ = "S" THEN
        GOSUB Soft_Reset
        GOTO 30  @ DISPLAY MENU
250 IF A$ = "Q" THEN   END
   @ INVALID CHAR
260 PNT BELL'  GOTO 30  @ DISPLAY MENU
```

Figure 7. Slave2 and Master2 Programs.

F. MONORAIL SYSTEM COMMUNICATION

The development of the subprograms used to communicate between the two computers used on the monorail system is presented in this section. The RS-232 Serial Communication ports of both computers are connected to an infrared communication system. The monorail computer's transmitter operates at the same frequency as the plant computer's receiver. The monorail computer's receiver and the plant computer's transmitter communicate at a different frequency which allows full duplex communication. Full duplex communication means that data can be sent and received by both computers at the same time.

Data is transmitted in byte segments bythe RS-232 ports. This is the lowest level of communication that can take place over the communication link. communication capabilities of the ExKeys are performed in several layers of subprogram executions. For example, some motor control ExKeys transfer large integer values that are in a three byte range. They will execute lower level subprograms that accept the large integer values and break them into three bytes. The three bytes are sent to the other computer with three executions of SendCOM1 and ReadCOM1. A communication protocol is used by SendCOM1 and ReadCOM1 in which the lowest level communication statements are called several times for each byte sent. The protocol will be described later. Figure 8 illustrates the communication levels concept.

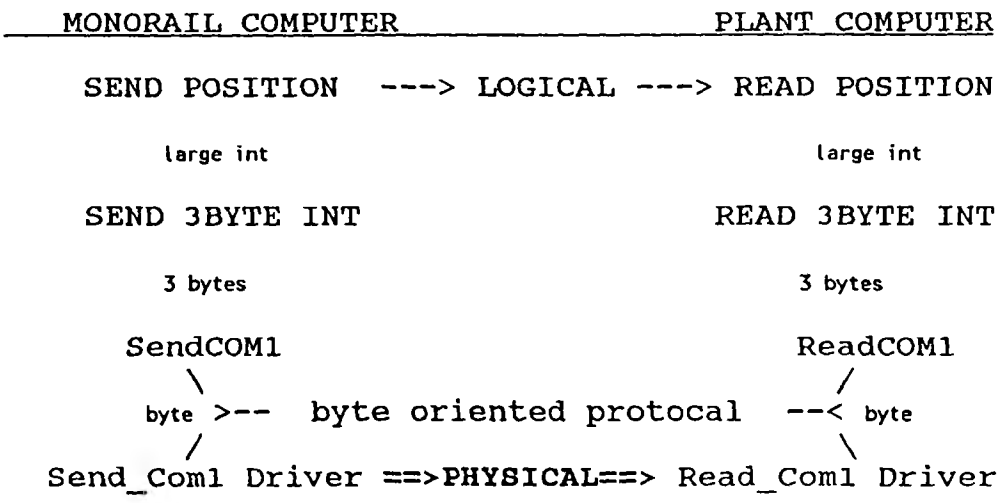

Figure 8. Communication Levels.

Only the data transmit and the data receive output lines[15,17,24,25] from the RS-232 ports are connected to the infrared communication system. The modem control lines normally associated with serial communication[15,17,24,25] are not connected to the infrared circuits. This creates some difficulties on the monorail computer.

The Turbo Pascal communication statements expect the modem control lines to indicate the communication link is operational prior to performing their task. They also do not handle communication errors gracefully so these communication statements are not used. Instead, the slave's communication ExKeys use custom communication drivers that overcome the shortcomings of the built-in capabilities. These custom communication drivers are subprograms that are included in the slave ExKey file. The Turbo Pascal language has statements that allow access to the hardware of the monorail computer which allow the communication drivers to be written in this language.

The built-in communication statements of MacBASIC do not use the modem control lines under the mode of communication used on the monorail system. Therefore these statements can be used and custom communication drivers could not have been written in this language. This is fortunate because this language does not have the capabilities to access the hardware ports which would allow custom drivers to be written.

1. Monorail Computer Communication. A RS-232 Add-on card[25] placed in a slot of the monorail computer's backplane performs the communication used on the monorail system. It is considered the primary asynchronous communication port by DOS and is referred to as COM1. This computer card is based on the National Semiconductor INS8250 Asynchronous Communication Element (INS8250)[24]. It appears as eight consecutive address locations in the computer's memory space. Ten internal registers can be accessed through: various combinations of addresses, the read/write line from the computer, and the Divisor Latch Access Bit (DLAB). The access of the INS8250's internal registers and the DLAB will be discussed later.

a. Hardware Port Access. Turbo Pascal has a predefined array that is used to access a data port in the Input Output (I/O) address space. It is named Port and the array type is byte. Its index is the location of the port. When this array is on the left side of an assignment statement, a

value can be sent to a port. When used on the right side of an assignment statement, a port can be read. For example, the value of the Line Status register at hexidecimal (Hex) location 3FD can be read into the byte type variable x with the statement:

```
x := Port[$3FD];.
```

The byte type variable x can be sent to the Transmitter Holding register at Hex location 3F8 with the statement:

```
Port[$3F8] := x;.
```

This array is used throughout the slave's subprograms so predefined constants are used to associate a readable name with each INS8250 register location. These names appear in the slave ExKey listing in Appendix A.

b. SetUpCom1 Procedure. Before any communication can occur over the RS-232 ports, the ports must be initialized. This is performed with SetUpCom1. COM1 is initialized by software. This involves three of the INS8250's registers. The SetUpCom1 ExKey automatically sets these registers to the correct values. It uses a software interrupt[3,12,13,16,17] to call the Basic Input/Output System (BIOS)[12,16,17] to perform the required register initializations. The explanation of the BIOS and the software interrupts is beyond the scope of this thesis.

The SetUpCom1 procedure can be changed easily to allow COM1 to be initialized differently, even if the user does not understand software interrupts. Variables that represent the values for the initialization attributes can be set

to different values. These attributes are the baud rate, the type of parity, the number of stop bits and the number of data bits[12,15,16,17]. The variable BAUD is set to the value for the baud rate. The variable PARITY represents the type of parity with the possiblity of no parity. The variables STOP and DATA are set to the values for the number of stop bits and the number of data bits respectively. The meaning of the initialization attributes is not covered in this thesis. The following is the portion of the SetUpCom1 procedure that allows these variables to be changed:

```
{*** CHANGE THESE TO SET UP DIFFERENTLY}
BAUD := 1200; {choices>>> 300,1200,4800,9600 }
PARITY := 0;  {choices>>> 0:none, 1:odd, 2:even}
STOP := 2;    {choices>>> 1 or 2 }
DATA := 8;    {choices>>> 7 or 8 }.
```

The SetUpCom1 procedure performs three additional functions. It clears a bit which controls which register is addressed at $3F8. This bit was mentioned earlier when the locations of the INS8250's registers were discussed. It is the Divisor Access Latch Bit (DLAB) and is the most significant bit in the Line Control register. It must be reset so that the address $3F8 and $3F9 will access the correct registers.

The DLAB bit of the Line Control register is reset by reading that register and logically "AND"ing it with the hexidecimal value $7F. This is done so that the rest of the bit values are not changed. The result is stored back into the Line Control register.

The SetUpCom1 procedure also disables the interrupt system of COM1. This is accomplished by sending all zeros to the Interrupt Enable register. The final function of this procedure is to initialize four constants that are used by the protocol of SendCOM1 and ReadCOM1.

c. Read Line Status Reg Procedure. A general purpose procedure used to check the status of the bit positions of the Line Status register is available in the slave ExKey file. This procedure is named Read_Line-_Status_Reg It has seven variable boolean parameters. Each parameter represents a bit value for each position of the Line Status register. An example that uses this procedure is presented later in this section. Figure 9 contains the code for this procedure.

```
procedure Read_Line_Status_Reg
    (VAR  R_full, overrun, parity, framing,
          break, empty_TB, empty_TR:  boolean);
var  x:  byte;
begin
   x := port[Line_Status_Reg];
   R_full := false;
        if (x AND $01) = $01 then  R_full := true;
   overrun := false;
        if (x AND $02) = $02 then  overrun := true;
   parity := false;
        if (x AND $04) = $04 then  parity := true;
   framing := false;
        if (x AND $08) = $08 then  framing := true;
   break := false;
        if (x AND $10) = $10 then  break := true;
   empty_TB := false;
        if (x AND $20) = $20 then  empty_TB := true;
   empty_TR := false;
        if (x AND $40) = $40 then  empty_TR := true;
  end;
```

Figure 9. Read_Line_Status Procedure.

d. Send Driver Procedure. The communication driver that transmits a byte on COM1 is the procedure Send_Driver. It uses an integer parameter instead of a byte because integer types are more convenient to work with than byte types. Figure 10 contains the code for this procedure.

```
procedure Send_Driver (x: integer);
   var  B1,B2,B3,B4,B5, Empty_TB, B7:   boolean;
begin
   Empty_TB := false;
   while (NOT Empty_TB) do
     Read_Line_Status_Reg(B1,
        B2,B3,B4,B5,Empty_TB,B7);
  port[Tx_buffer] := Lo(x);
  end;  { proc Send_Driver  }
```

Figure 10. Send_Driver Procedure.

Two INS8250's registers are involved when a byte is transmitted over COM1. These are the Line Status register and the Transmitter Holding register. The Transmitter Holding register accepts a byte from the computer and sends it over COM1. The Line Status register uses seven of its eight bit positions to indicate the status of data communication and the status of INS8250. When it is set (1), the sixth bit indicates that the Transmitter Holding register is empty. The Line Status register should be read and this bit tested until it is set before a byte is written to the Transmitter Holding register.

Seven boolean variables are defined in this procedure. They are used as parameters with the Read_Line_Status_Reg procedure to represent the seven bits of the Line Status Register. Only one of these variables is used in this

procedure but all must be defined. The sixth parameter indicates if the Transmitter Holding register is empty. It is named Empty_TB for empty Transmitter Buffer. It will remain unset (0) until the Transmitter Buffer register is cleared. This automatically occurs when the previous old contents in the register are sent.

The first three lines of this procedure cycle until the Transmitter Holding register is empty. Then the low order byte of the integer value is written into the Transmitter Holding register where it will be sent over the transmit line of COM1. The monorail computer can write bytes to this register faster than the INS8250 can transmit them over COM1. This is the reason the code has to loop until this register is cleared. The constant Tx_buffer is the address of the Transmitter Holding register.

e. Read_Driver Procedure. The communication driver that receives a byte over COM1 is the procedure Read_Driver. It has two parameters. The first is a variable integer. It returns the value of the byte received over COM1. The second parameter is a boolean variable that is set if a communication error occurs. The communication errors will be discussed later in this section. Figure 11 contains the code for this procedure.

The Read_Line_Status_Reg procedure is also used by this communication driver. All but two of the seven bit positions are used by the Read_Driver communication driver. The first boolean parameter represents the bit of the Line

Status Register that is set when a byte has been received into the Receiver Buffer register. The name R_full, for receiver full, is used for this parameter.

```
procedure Read_Driver (VAR i: integer;
                       VAR Read_errors: boolean);
var  x :  byte;
     R_full, E1,E2,E3,E4, B1,B2,  dataRDY:   boolean;
begin
  Read_errors := false;  dataRDY := false;

  while NOT(Read_problems) AND NOT(dataRDY) do begin
     read_Line_Status_Reg (R_full, E1,E2,E3,E4, B1,B2);
     if  E1 OR E2 OR E3 OR E4  then
         Read_errors := true
       else
         if R_full then
            dataRDY := true;
     end;  (while)

    x := port[Rx_buffer];
    if (Not read_errors) then
       i := ord(x);

    end;  ( proc Read_Driver  )
```

Figure 11. Read_Driver Procedure.

The next four parameters of the Read_Line_Status_Reg procedure represent four error conditions that may occur when a byte is received over COM1. The individual meaning of these four bits is not presented in this thesis. A logical "OR" combination of these four bits checks for any one of these errors.

When a byte is expected to be received over COM1, the Read_Driver procedure should be executed. This procedure monitors the R_full bit and the four error bits. If any of the four error bits are set, an error has occurred and any data in the Receiver Buffer register will be invalid. This

procedure sets the second boolean parameter to true so that it indicates that an error occurred. The value returned in the first parameter is not changed. The Receiver Buffer register still needs to be read because doing so will reset the R_full bit. All four error bits are cleared when the Line Status register is read.

If the R_full bit is set and none of the error bits are set, the Receiver Buffer register contains a valid value. It can be read with the Port array and the value is changed to an integer. The integer value is returned in the first parameter. The second boolean parameter is initialized to indicate that no errors have occurred and will not change values unless an error occurs.

2. Plant Computer Communication. The plant computer has two RS-232 ports in the Macsym 150 but only one is used for the communication between the two PBX pilot plant computers. Each port operates independently. They are contained on a single card that fits into one of the Macsym 150's slots. The built-in statements of MacBASIC can be used to perform all of the communication functions required by the monorail system.

a. SetUpCom1 Subroutine. An open statement must be performed before using one of the communication ports. The OPENR statement[4,5,6] is used to open a logical channel for input and another logical channel for output. It is used to assign an integer number to the input channel. It

automatically assigns the next greater integer to the output channel. It requires two parameters. The first parameter is the input channel number. The channel numbers are used by MacBASIC I/O statements to route data to the RS-232 port. The two variables, RS232OUT' and RS232IN', are assigned two consecutive values and used with I/O statements in the example programs presented later.

The second parameter of the OPENR statement is a symbolic name for the actual port. This value must be a character string (enclosed in quotes). The system name for the first RS-232 port is "$QTI:0". The first port is labeled "A" on the RS_232 connector. The second port uses the symbolic name "$QTI:1" and is labeled "B" on the connector.

The attributes of the plant computer's RS-232 ports are not initialized like the RS-232 port of the monorail computer. The monorail computer's communication attributes are set by software while the plant computer's communication attributes are set when the plant computer is turned on. Dual Inline Package (DIP) switches on the RS-232 card are used to set the different communication attributes[9].

The attributes of the RS-232 ports of both computers have to be identical. This is the reason why the slave's SetUpCOM1 ExKey does not prompt the operator for the values of the attributes. These attribute values are program constants that match the attributes of the plant computer. If the DIP switches on the plant computer's RS-232 card are

changed, the code for the slave's SetUPCOM1 ExKey will have to be changed.

b. Communication Statements. The MacBASIC I/O statements that correspond to the Read_Driver and Send_Driver ExKeys of the monorail computer are PNT:RS232OUT' and INP:RS232IN' respectively. Both statements require a parameter that represents the value to be sent or the value to be read.

Figure 12 contains a pair of programs that demonstrate the use of the MacBASIC communication statements and custom communication drivers on the monorail computer. These programs send random characters to each other and also print them on their own screens. The plant computer first generates a capital letter, sends the letter to the monorail computer, and prints the letter on its own screen. It waits until a letter is received from the monorail computer.

```
10 RS232IN' = 5
15 RS232OUT' = 6
20 OPENR:RS232IN' "$QT1:0"
30   CHAR1' = 65 + RND*26
40   PNT:RS232OUT' CHAR1'
50   PNT CHAR1'
60   INP:RS232IN' CHAR2'
70   PNT CHAR2'
80   GOTO 30
```

```
Var char1,char2: byte;
{$I keyword.pas}
begin
  SetUpCOM1;
  while true do begin
    Read_Driver (char1);
    write(char1);
    char2 := lo(32 +
             random(27));
    Send_Driver (char2);
    write(char2);
    end;
  end.
```

Figure 12. Lower Level Communication Example.

Meanwhile the monorail computer has been waiting for a letter from the plant computer. When it receives one, the letter is printed on its screen. A new random capital

letter is generated which is sent to the plant computer. The letter is printed on the screen. The monorail computer then waits for a letter to be sent from the plant computer.

The sequence of one computer sending a letter to a receiving computer, then waiting for a new letter from the other computer before repeating the process, is a type of communication protocol. It involves handshaking which prevents one computer from overrunning the other computer with letters. It does this by switching the role of which computer is sending and which computer is receiving.

3. Byte Oriented Communication Protocol. The difference between the communication ExKeys presented earlier and the communication drivers just presented involves the level at which communication is performed. The ExKeys implement a protocol that uses the communication drivers. There are several reasons for the use of a protocol to develop a higher level of communication above the level of the communication drivers. The burden of handling errors that can occur when a byte is received at a communication port is removed from a programmer who is using the communication ExKeys. It is also important to trust that the communication is taking place and that it is occurring correctly. These are the main requirements of the protocol for the communication between the two computer systems used on the monorail system.

The amount and frequency of data transferred is another factor affecting the protocol. On the Monorail system, the

communication is limited to occasional transmissions of single bytes and packets of three bytes. This is a small amount of data compared to the typical use of communication links. Most communication protocols[26-29] handle much larger packages of data.

The characteristics of the infrared communication link also affect the protocol. The infrared link cannot be considered extremely accurate and dependable because of the nature of infrared signals. In general, it will be assumed that a byte transmitted over the link will not be lost but it might not be received correctly. Sometimes an error can occur during transmission that may not be detected by the hardware of the communication port. This problem has to be corrected by the protocol.

This section attacks the characteristics and requirements conflict of the Monorail communication system. The protocol, which is named the Byte Oriented Communication Protocol (BOCP), is presented in two stages. The basic scheme of the protocol is presented first. The basic scheme does not handle errors occurring at the driver level. The basic scheme will be expanded upon to consider those errors in the second stage. After the presentation of the protocol, a discussion about the problem of lost data will follow.

The basic protocol scheme assumes that a transmission error will not occur and that the transmission will not be lost. It insures that a byte is transmitted correctly. It

does this by returning the byte read by the receiving computer, back to the sending computer. At this point, the sending computer knows whether the byte was received by the other computer. The sending computer can compare the returned byte to the byte that was sent to check for correctness. The sending computer is satisfied that the communication has occurred correctly if the two bytes are equal but the receiving computer does not know the result of the test. Another handshaking process is used so that the sending computer can notify the receiving computer if the communication occurred correctly. Figure 13 contains a diagram of the basic protocol scheme.

If the sending computer receives the same byte that it sent, it will send an acknowledge (ACK) byte back to the receiving computer. The receiving computer returns the byte received and then checks it to determine if it is an ACK. This is done because the computers need an even number of transmissions to perform correct handshaking.

If the sending computer does not receive the same byte it sent during the first handshaking sequence, then the sending computer will send a not-acknowledge (NAK) byte to the receiving computer. The NAK signals that the sending computer has determined that the transmission did not occur correctly. The transmission process is repeated until the byte is sent accurately.

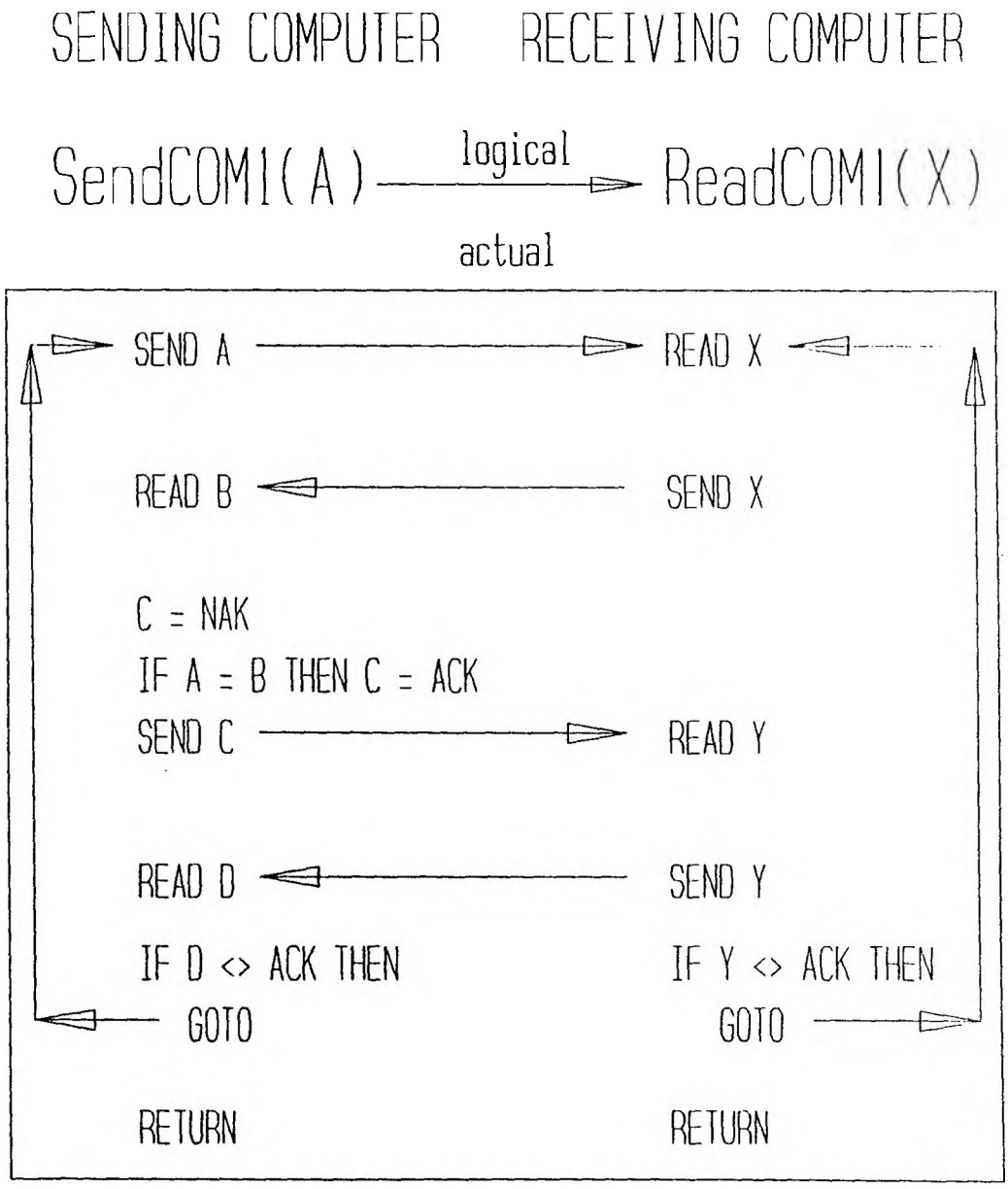

Figure 13. **Basic BOCP Scheme.**

The second stage of the presentation of the BOCP handles the hardware errors that are detected by the read drivers. Figure 14 shows the additions to the basic scheme that handle returned errors from the read drivers. In general, if an error is detected, a NAK is returned. MacBASIC has an ON ERROR statement that allows a segment of code to be executed when an error occurs. The use of this statement and the error numbers corrosponding to the type of errors is beyond the scope of this thesis.

A serious problem occurs if the last transmission of the protocol is received with an error. The sending computer does not know if the receiving computer returned a ACK and left the protocol scheme or if it sent a NAK and is repeating the protocol. It is obvious that the protocol is not foolproof but in most cases it will recover from errors.

The infrared communication circuits might later be developed to include a signal line that would indicate if the communication link is not reliable at a particular point in time. This status line should be read just before sending a byte over the link. If the status does not indicate a possible problem, the byte should be sent. Otherwise the transmission should wait until the line changes to a safe status indicated. This may cause problems if the line does not change quickly from a bad indication to a good one.

A time out scheme could also be added to the protocol at a later time if the current protocol becomes less reliable. A time out scheme interrupts the wait at a read

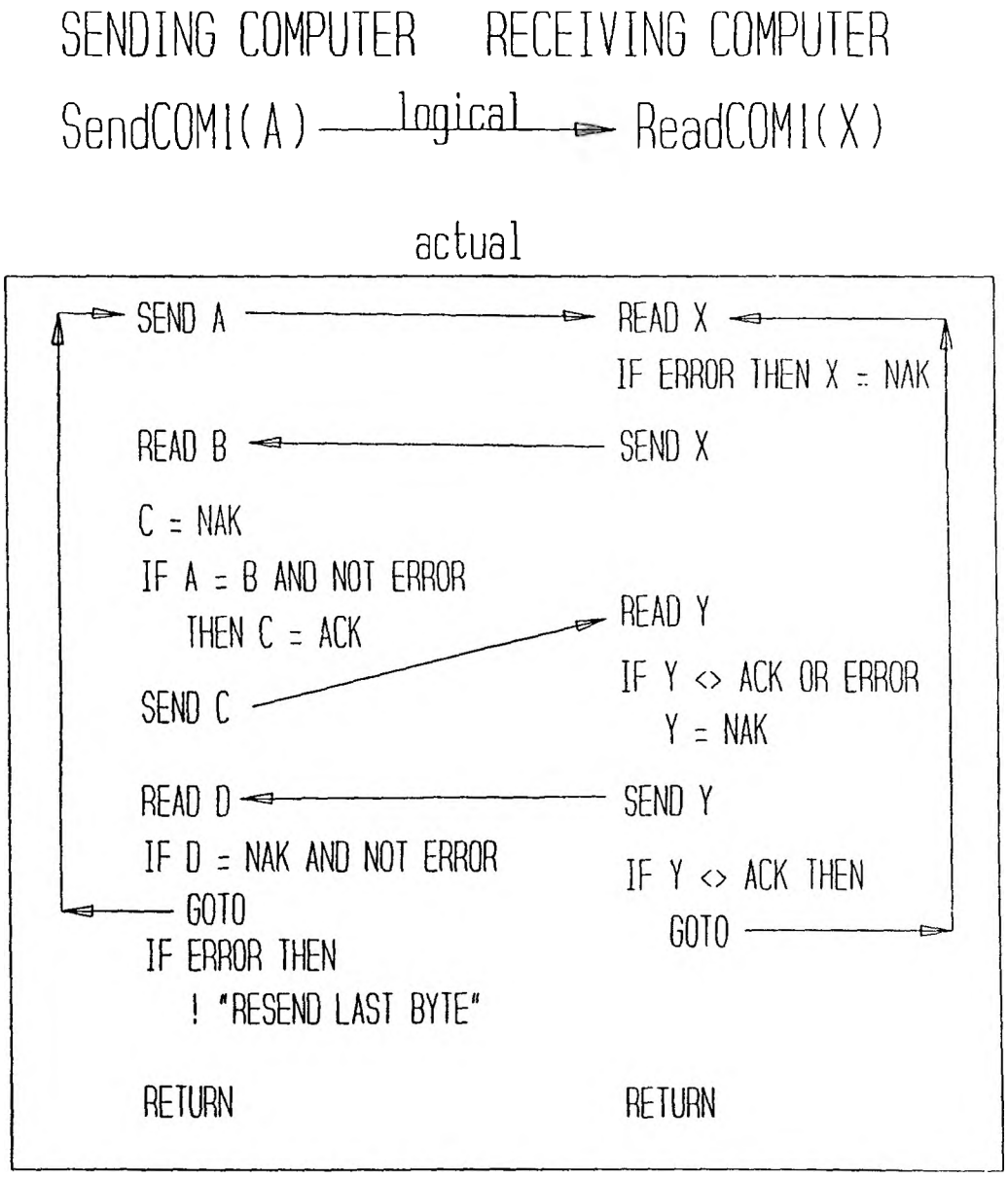

Figure 14. **BOCP Scheme with Error Handling.**

driver after a period of time elapses. This would occur if the transmission was lost. The first read driver for the receiving computer should not time out because of the unpredictablity of the time it will receive a byte. The other reads are more predictable once the protocol scheme starts.

A synchronization problem occurs with timeouts because neither side of the protocol knows what occurred at the other side after a timeout. For example, if the first read statement on the sending side timed out, the sending side does not know if the byte it sent was lost or if the byte was received by the other side and the return transmission was lost.

The quest for the perfect protocol for the monorail system would require a large effort. The answer to the perfect protocol is beyond the scope of this thesis. The reason for the presentation of the protocol used in this thesis is to allow the expansion of the protocol by future programmers who are given the task of updating the current protocol to handle new characteristics or requirements.

G. MOTOR CONTROL HARDWARE.

The hardware involved with the control of the monorail carrier's electric motor is the LAB 40 extended bus system. It connects the LAB 40-HP function module containing a HP HCTL-1000 motor control processor to the monorail computer. The driver card for the LAB 40 system[18] is referred to as LAB 40-PC in this thesis. It fits into a slot in the monorail computer's backplane. The HP HCTL-1000 is on an

external board which is known as the LAB 40-6[20] but it is referred to as LAB 40-HP in this thesis. A ribbon cable is used to connect the LAB 40-PC to the LAB 40-HP. A second external board is also connected to the LAB 40-PC with the ribbon cable. This board is an A/D input system that is known as LAB 40-2[19] but it is referred to as LAB 40-AD in this thesis. This section presents the theory of the hardware involved with the implementation of the motor control ExKey. Appendix A contains the listing for the slave ExKeys and Appendix B contains the listing for the master ExKeys.

1. LAB 40 Extended Bus System. The extended bus driver card is based on the Intel 8255A Programmable Peripheral Interface Device (8255)[30]. The 8255 is configured so that the bus contains two eight bit ports, a board select port and a set of power and control lines for the function modules[18]. The 8255 is composed of four registers.

The first two 8255 registers, known as PortA and PortB, translate into eight bit ports on the extended bus. When accessing one of these registers, the respective bus port is accessed. The third port is used to select a function module. This port is called PortC. The forth register is a control register. It determines the mode of operation for the other three 8255 registers.

The computer's address lines, A3 and A2, are used to select one of the four 8255 registers. The lowest two address lines, A1 and A0, are not used for decoding. Therefore, four consecutive memory locations address a single

register. The value used to access a port usually has an effect on the function module selected. The address lines, A1 and A0, are available on the extended bus for use by different function boards.

The control register is located at memory locations 748 through 751. The value 192 is used to initialize the 8255 for the configuration required by the monorail system. This value causes PortA to be initialized as a bidirectional port and PortB to be initialized as a latched output port. PortC is initialized to be used as the board select port.

With bidirectional PortA, bytes of data can be read and written between the computer and a function module. PortA can be addressed at 736 through 739. This port remains in tristate mode except during an I/O operation. A write operation involves writing a byte to this register. The value of the byte will appear on PortA and a OUT* control line on the bus will strobe low. (The notation '*' after a line name is used to indicate active low.) A read operation involves first reading this register which causes the 8255 to perform its own read operation on PortA. The register must be reread, after a short pause, to obtain the data read at PortA by the 8255. During the 8255's read operation, an IN* line on the bus is strobed low.

With PortB initialized as a latched output port, the value written to it remains on the bus until replaced with a new value. PortB is addressed at locations 740 through 743. The use of PortB depends on the function module addressed.

Both function modules used with the PBX pilot plant use this port to aid with I/O operations.

PortC is used to select different function modules. PortC is addressed at locations 744 through 747. Eight function modules can be addressed by one of the eight select lines on the bus. Only one of these lines should be set at a time. The lowest three bits of the output from this port are connected to a three to eight decoder. Only the values zero through seven should be sent to this register.

There are several features of the LAB 40 system that are not used on the monorail system. Those features of the LAB 40 system are not presented in this thesis. The presentation of the LAB 40 system is oriented for the monorail system.

2. Hewlett Packard HCTL-1000 Function Module. The function module uses a HP HCTL-1000 motor control processor to control the electric motor. There are two HP HCTL-1000s on this board and each one can select one of four different motors but this presentation treats the board as if it has only one HP HCTL-1000 connected to one motor.

a. Register Access. The HP HCTL-1000 has 32 user accessable registers. The I/O operations with the HP HCTL-1000 registers involve the use of PortA and PortB after LAB 40-HP is selected. The HP HCTL-1000 has a bidirectional address/data port that is connected to LAB 40's PortA. It allows bytes of data to be written and read between the HP

HCTL-1000 and the monorail computer. The individual bits of PortB have different functions.

The first bit of PortB (Bit 0) is connected to the read/write* line of the HP HCTL-1000. Bit 0 is set for a read operation and cleared for a write operation. The forth bit of PortB (Bit 3) is connected to a reset line on the HP HCTL-1000. The reset is activated when Bit 3 is cleared. For normal operations this bit should always be set.

The other bits on PortB are used for functions that are not used on the monorail system. These bits should always be cleared. Therefore only four values should be written to PortB. Those values are 1, 2, 8 and 9. The values 1 and 2 cause the HP HCTL-1000 to reset because the bit 3 is cleared. The values 8 and 9 mean that Bit 3 is set as required for normal operations. The value 8 prepares the HP HCTL-1000 for a read operation while the value 9 prepares it for a write operation.

Most of the motor control ExKeys have to access several of the HP HCTL-100 registers. Two subprograms are used by the motor control ExKey to perform read and write operations with a HP HCTL-1000 register. These subprograms are HPread and HPwrite respectively. The function HPread is used to read a byte of data from a register. The procedure HPwrite is used to write a byte of data into a register. The code for both subprograms is presented in Figure 15.

```
procedure HPwrite(addr,data: integer);  begin
   port[B] := 8; {lowers Rd bit, +8 for no reset}
   port[A] := addr;    {send addr of reg}
   port[A+1] := data;  {send data to reg}
   end;

function HPread(addr: integer): integer;  begin
   port[B] := 9; {sets Rd bit, +8 for no reset}
   port[A] := addr;     {send addr of reg}
   port[A+1] := 0;      {strobe CS low}
   HPread := port[A+1]; {start read}
   delay(110);   {pause for read operation}
   HPread := port[A+1]; {Read data from PortA}
   end;
```

Figure 15. HP HCTL-1000 Register Access Subprograms.

Two constants, A and B, were previously set to the first locations that address PortA and PortB respectively. By adding a value between 1 and 3 to one of these constants, the address lines A0 and A1 are used to aid I/O operations with HP HCTL-1000 registers. These two address lines are connected to the two input pins, A and B, of a 74HC155 Dual 2-to-4 Line Decoder/Demultiplexers[20,31].

Two bus control lines are also connected to the 74HC155. These two lines are the IN* and OUT* lines from the 8255. Their function is similar to the read and write lines of a microprocessor. Figure 16 contains a simplified block diagram of the 74HC155 with its connection between the monorail computer and the HP HCTL-1000.

The four outputs from the 74HC155 are inverted; all output signals are set except the one selected which will be low. When the 74HC155 is enabled with the board select line, the IN* and OUT* lines are connected such that a write operation will clear the 74HC155's output pin labeled Xn.

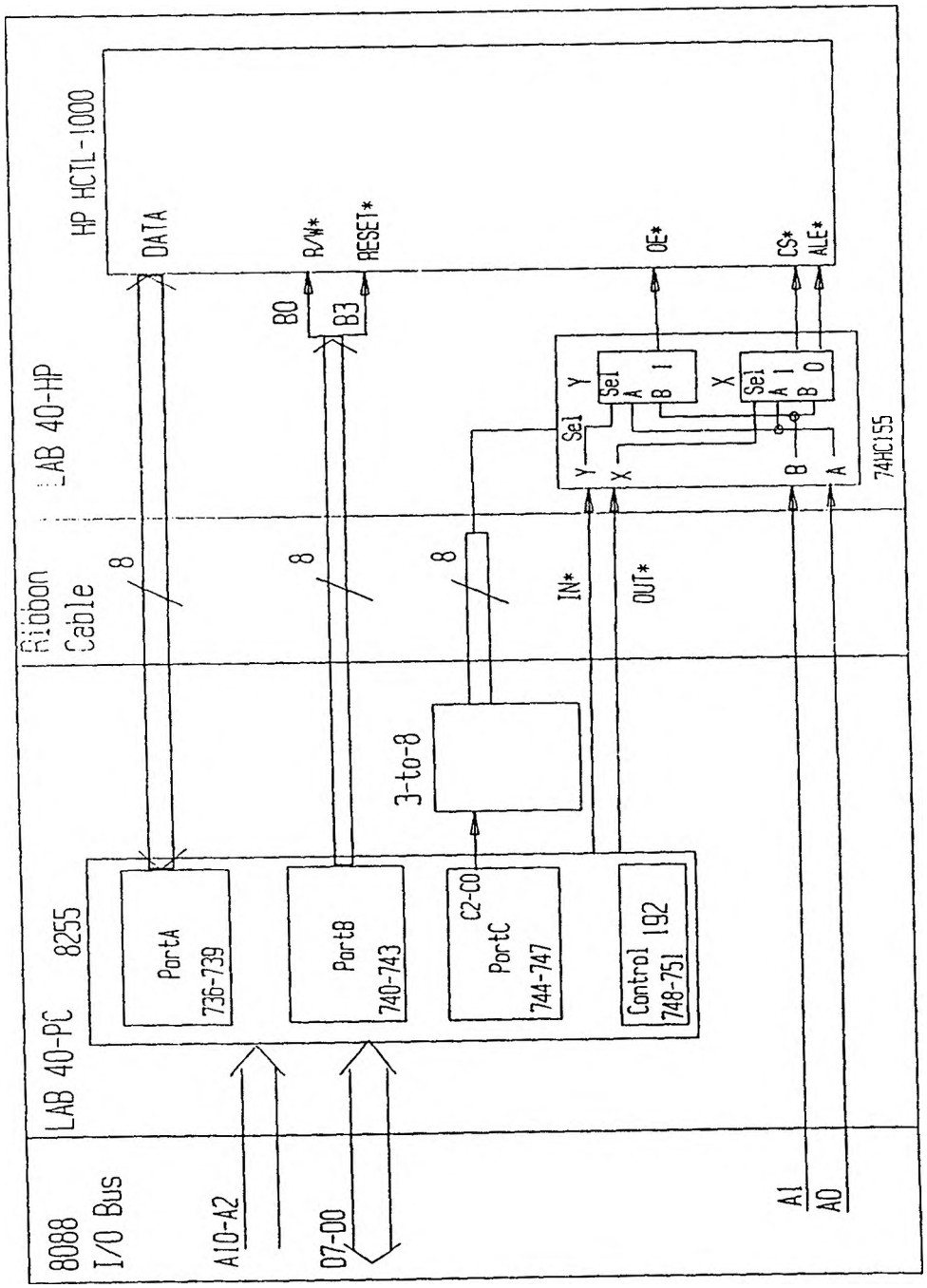

Figure 16. Simplified Block Diagram of LAB 40-HP Connection.

The value of n will be 0 to 3 according to the binary representation of A0 and A1. A read operation will clear the output pin labeled Yn where n is the binary representation of A0 and A1.

The HP HCTL-1000 has three pins that are used for its I/O operations. These pins are Output Enable* (OE*), Address Latch Enable* (ALE*) and Chip Select* (CS*). The 74HC155's Y1 pin is connected to the HP HCTL-1000's OE* pin, the 74HC155's X0 pin is connected to the HP HCTL-1000's ALE* pin and the 74HC155's X1 pin is connected to the HP HCTL-1000's CS* pin.

An I/O operation with the HP HCTL-1000[20,22] starts by setting the read/write* line on PortB to the appropriate value, it then sends the address of the register to be accessed to the HP HCTL-1000. The ALE* pin should be strobed low to latch the data on PortA into an internal address register. The statement

```
port[A] := addr;
```

performs this function. The OUT* from the 8255 is strobed low during a write operation with PortA and the address lines A0 and A1 are both zero so the ALE* will strobe low with the value, addr, on PortA.

A write operation involves sending the data byte to the HP HCTL-1000 after the address has been latched into the HP NGTL-1000 internal address register. The CS* pin has to be strobed low with the data on portA. This stores the data in an internal data latch in the HP HCTL-1000. The HP

HCTL-1000 then stores that data into the addressed register. The statement

```
port[A+1] := data;
```

performs this function. The OUT* from the 8255 is again strobed low and the address lines A0 and A1 equal 1 so the CS* will strobe low with the value, data, on PortA.

A read operation also has to strobe the CS* pin, after the internal address latch has the register number, but the data it stores in the internal data latch is not used. Given the address of the register and the read/write* pin set, the HP HCTL-1000 will store the addressed register's contents in an internal output latch. The final step involves three statements because of the method of reading PortA required by the 8255. The first

```
HPread := port[A+1];
```

statement strobes the OE* pin low which will cause the data in the output latch to be placed on the HP HCTL-1000's address/data port. A short delay, 110 microseconds, is needed for timing purposes before a second

```
HPread := port[A+1];
```

statement reads the byte from the 8255.

The HPread and HPwrite subprograms simplify the access to the HP HCTL-1000's registers. Figure 17 presents a simplified register block diagram which shows the relationship of the registers used during trapezoidal motor moves. The diagram is a modified version of the diagram presented in the Technical Data literature for the HP HCTL-1000. Only

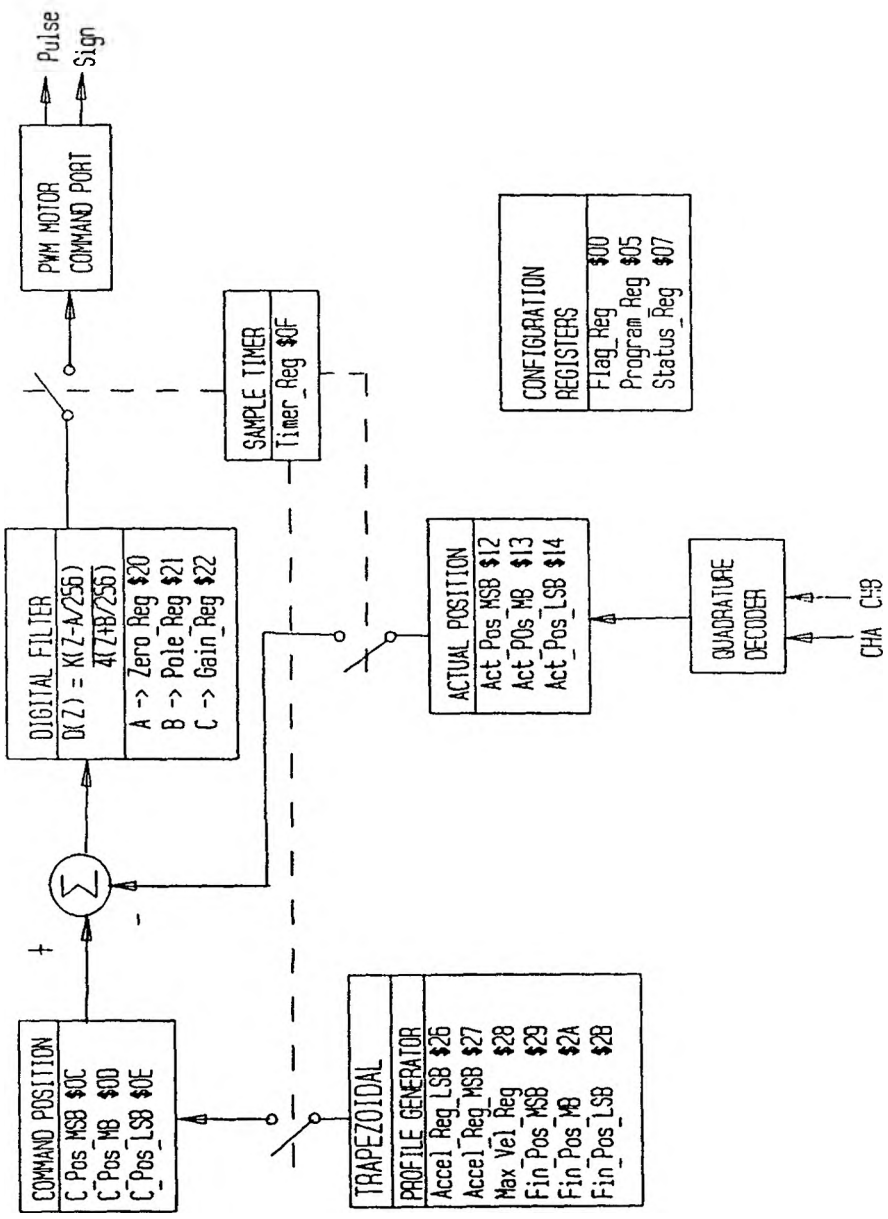

Figure 17. Simplified HP HCTL-1000 Registers Block Diagram.

part of the HP HCTL-1000's capabilities are presented in this thesis.

Some of the data used by the HP HCTL-1000 requires more than one eight bit register to hold the value. The Final, Command and Actual positions require three registers to hold their values. The acceleration rate uses two registers. The format of these registers will be presented later. The HPread and HPwrite subprograms are used several times when working with these values.

b. Trapezoidal Profile Generator. When instructed to start a motor move, the Position Profile Generator will determine a command position as defined by the trapezoidal profile. The Acceleration rate, Maximum Velocity and Final position values determine this profile. The Actual position is subtracted from the Command position. The difference is used by the digital filter to compute the motor compensation. The compensation is converted into Pulse Width Modulated (PWM) format and placed in the PWM Motor Command Port register. The value in that register is sent to an amplifier circuit which controls the sign and magnitude of voltage sent to the monorail carrier's motor.

The sample timer triggers each process of computing the motor compensation. At each clock cycle, a new Command position is computed, the Actual position is read, the compensation is computed, and the PWM Motor Command Port is adjusted. The sample time period affects the possible values for the acceleration rate and the maximum velocity.

The formulas that compute the possible range for these values will be presented later.

c. Initialization Process. The Init_LAB40 ExKey must be performed before any other motor control ExKey is performed. This ExKey first initializes the 8255 of the LAB 40-PC. Next, the reset line of the HP HCTL-1000 is strobed low by a procedure called Hard_Reset. This reset must be performed before the HP HCTL-1000 is used.

The Hard_Reset procedure was not presented as a ExKey because it normally does not have to be performed more than once. A corresponding Hard_Reset subroutine is included on the Master ExKey' file, if a hard reset option in a monorail program is desired.

The reset performed by the Soft_Reset ExKey is a different type of reset than the Hard_Reset. A soft reset performs register initialization while the hard reset initializes the HP HCTL-1000's internal circuits. The HP HCTL-1000 can be instructed to perform a soft reset. The Soft_Reset ExKey instructs the HP HCTL-1000 to execute a soft reset and then it performs extra functions which will be explained later. To distinguish between the soft resets, the HP HCTL-1000's soft reset will be called HP soft reset while the Extended keyword will still be called Soft_Reset ExKey. An HP soft reset is automatically performed after a hard reset has been executed.

The HP soft reset sets the filter parameters, the timer value and the position registers to default values set

inside the HP HCTL-1000. After the Soft_Reset ExKey executes the HP soft reset, it restores those registers to their old values. Program variables, which hold the values stored in these registers, are used to restore the registers. The Init_LAB40 ExKey initializes these variables and they are updated when any of these registers are changed. The procedure Restore_HPregisters is used to set the register values to the variable values.

d. Filter Parameters. The Set_Filter_Variables ExKey performs three functions. For convenience, this Exkey sets all three filter parameters when any one of them is changed. Both versions of this ExKey could easily be broken into three separate ExKeys that would allow a master program to change only one of these registers. All three of these registers have a scalar byte format. Therefore, the range of values is zero to 255. The master version of this ExKey performs range checking.

The LAB 40-HP manual recommends that the pole be set to 64 or less and the zero be set to 255. It also states that the gain must be determined experimentally. The gain value is the most sensitive. It could cause the motor to oscillate if it is set too high. If it is set too low, the accuracy of the actual position at the end of a motor move will not be as precise.

Through experimentation, it was determined that values starting around 90 and below 255 will cause oscillation with the motor used on the monorail carrier. The ExKey uses 45

as the default value. Different motors can react differently to the gain value.

A motor move usually does not end exactly at the final position. A deviation of approximately 100 encoder counts can be expected. After a motor move has completed, the gain value can be set to a high value of around 100, then immediately set back to a non-oscillating value. This helps move the actual position closer to the intended final position and reduces the deviation to around 50.

e. Position Profile Parameters. The Sample Time register must be set to a value between fifteen and 255 for trapezoidal profiles. The variable Timer_Reg contains the address for this register in the slave ExKey file. The formula that converts the register value into the time period in microseconds is:

t = 16*(TIME' + 1) / frequency.

The variable TIME' contains the value stored in the time register. The frequency is 2MHz so the resulting range of periods is 128 to 2048 microseconds.

The acceleration rate used in the position generator is stored in two registers. One of the registers holds the integer portion of the acceleration rate and the other holds the fractional portion. The register holding the integer portion is called the most significant byte (MSB). The variable Accel_Reg_MSB contains the address for this register in the slave ExKey file. The range of values this register can hold is zero to 127 (Hex 7F).

The fractional portion of the acceleration rate is internally divided by 256. The register holding the fractional portion is called the least significant byte (LSB). The variable Accel_Reg_LSB contains the address for this register in the slave ExKey file. This register can hold values in the range of zero to 255. Given the range of the two acceleration registers, the combined values for acceleration rate range from 0 (0 + 0/256) to approximately 127.996 (127 + 255/256).

The units of acceleration are in encoder position counts per simple time period squared (EC/SP2). If an acceleration rate is held in a real variable, the integer portion can be stored in the MSB acceleration register if it is less than 128. The fraction part has to be multiplied by 256 then the integer portion of the result can be stored in the LSB register.

Since the units for the acceleration rate are based on the sample time period, the length of the time period affects the actual acceleration range. For short time periods, the acceleration range increases because the position generator updates the next command position quicker. If the time period is long, the acceleration rate range decreases. For example, given a constant acceleration rate, a short time period means that the motor will accelerate twice as fast compared to a period twice as long.

A formula which changes the acceleration rate from units of EC/SP2 to an acceleration rate in units of Revolutions per microsecond squared (R/mS2) is

Accel * (2E6/(16.0*(TIME'+1)))^2 / PperR.

The acceleration value in these registers is multiplied by the square of the sample time period per second. At this point, the units of acceleration are in encoder positions per microsecond squared. It is converted to R/mS2 by dividing it by the encoder positions per revolution which is stored in the variable PperR.

Because an acceleration rate of zero is possible but invalid, the value of 1/256 EC/SP2 is considered the minimum acceleration rate. With this value, the MSB register contains zero and the LSB register contains one. It is used to compute the minimum acceleration in RPM given the sample time period. The formula for the minimum acceleration rate in RPMs (assuming not zero) is

1.0/256.0 * (2E6/(16.0*(TIME'+1)))^2 / PperR.

The largest acceleration rate in ECperSP is the value 127 + 255 / 256. This value causes the MSB register to contain its largest possible value of 127 and the LSB register contains its largest possible value of 255. The formula

(127+255.0/256.0) * (2E6/(16.0*(TIME'+1)))^2 / PperR

is used for the maximum acceleration rate in RPMs.

It is desirable to have the monorail carrier accelerate at the slowest possible rate so the time period is set to 255. The default acceleration rate in ECperSP is 1/256.

This creates the smallest possible acceleration since the time period is at its greatest possible value.

The velocity rate used by the position generator is held in one register. The variable Max_Vel_Reg contains the address for this register in the slave ExKey file. The range of values for this register is zero to 127 (Hex 7F). The units of velocity are encoder position counts per sample time period (EC/SP). The length of the sample time affects the range of the actual velocity. A conversion process is used to convert the register value, which is in EC/SP, to revolutions per microseconds (R/mS). The formula for this conversion is

Max_Vel_Reg * (2E6/(16.0*(TIME'+1))) / PperR

This conversion formula is similar to the acceleration rate conversion. The units of EC/SP are multiplied by sample time periods (SP) per microsecond (mS) then divided by the variable containing the number of positions per revolution (EC/R).

The velocity register can hold the value zero but the value of one will be considered the minimal velocity rate. The largest value this register can hold is 127. The conversion formula can be used to determine the possible ranges of velocity for a sample time period.

f. Position Registers. The Final position is used by the position generator to determine when a motor move should start decelerating so that the motor will stop at this position. It is held in three registers which hold a 24 bit

2's complement value. A position value has to be converted into three values in the range of zero to 255. These values can be stored into the position registers. The first value is the least significant byte portion of the position value. The variable Fin_Pos_LSB holds the address of this register in the slave ExKey file. The second and third values are the middle and the most significant byte values. The variables Fin_Pos_MB and Fin_Pos_MSB hold the addresses of these registers respectively. The listing for the master Enter_Final_Position ExKey in Appendix B contains code which converts a position to three byte values.

The HP HCTL-1000 uses two more position values other than the Final position. These are the Actual position and the Command position. These two positions have the same format as the Final position. The registers which hold the Actual position are Act_Pos_LSB, Act_Pos_MB and Act_Pos_MSB and the registers which hold the Command position are C_Pos_LSB, C_Pos_MB and C_Pos_MSB. The positions are computed by the HP HCTL-1000. These registers can be read through the LAB 40 system and their values converted into the position value. The MSB byte should be multiplied by 65536 (Hex 10000), the middle register byte should be multiplied by 256 (Hex 100) and those two results are then added to the LSB.

The Display_Position ExKey reads the actual position registers and displays the values on the plant computer. This ExKey uses a pair of routines that send three register values from the monorail computer to the plant computer.

Both ExKey files also have a pair of routines that send three byte values from the plant computer to the monorail computer. These routines are similar to the communication Exkeys. Both ExKey files contain their own versions of a Send_3_Byte_Integer subprogram and a Read_3_Byte_Integer subprogram.

g. Move Execution. Once a new final position has been placed into the HP HCTL-1000, a motor move can be executed. Two HP HCTL-1000 registers are involved with the execution of a motor move. These are the Program Counter register and the Flag register. The variables Program_Reg and Flag_Reg contain the addresses of these registers in the slave ExKey file. A procedure inside the Execute_Move ExKey is used to store the correct values into these registers. This procedure is Start_Trapezoidal_Move. It stores the correct values into the Flag register and the Program counter.

The Flag register is used to set one of the five flags inside the HP HCTL-1000. The flag involved with trapezoidal motor moves is Flag 0 (F0). This flag needs to be set to indicate that the HP HCTL-1000 is to perform a trapezoidal motor move. The first three bits of this register are used to address a flag. The fourth bit is used to set or clear the addressed flag. The value 8 is written to this register to set F0.

Once the trapezoidal flag, F0, is set, the motor can start. The Program Counter register is set to the value 3 to start the motor move. This is the register that is used

to perform an HP soft reset. A HP soft reset is performed by writing the value 0 into this register. When the HP HCTL-100 is not performing a motor move or a soft reset, the HP HCTL-1000 is in a mode called initialization/idle mode. This mode is achieved by writing the value 1 to the Program Counter register.

A third register is used during a motor move to detect the finish of the move. This register is the Status register. The variable Status_Reg contains the address of this register in the slave ExKey file. The fifth bit of this register is set during a trapezoidal motor move. When the move has completed, this bit will clear. Once the motor move has started, the Execute_Move ExKey will check this bit while in a loop which is also executing Display_Position. A boolean function inside the code for Execute_Move is used to check this bit of the Status register. The function, In_Trapezoidal_Move, reads the contents of the Status register and checks the fifth bit. If this bit is set, then this function returns as true. When the move has completed and this bit is no longer set, this function will return as false. Execute_Move will send the actual position of the motor to the plant computer until this function returns as false. The slave Execute_Move then sends a byte to the master Execute-_Move to indicate that the move has completed and the actual positions will stop being sent.

When a slave program that controls the motor is to be halted, the Quit ExKey should be performed. This procedure

performs a Hard_Reset which resets the signal to the motor's amplifier. It then deselects the LAB 40-HP function module and clears the control register.

IV. TABLE SYSTEM CONTROL.

The Table system connects a shell to the WOMBAT once a shell has been delivered to the washout area by the monorail system. It is a hydraulic table controlled by the plant computer. It is equipped with a grasping mechanism that can hold a shell. The front and back of the table move separately. The front of the table is the side nearest to the WOMBAT and the back is the side furthest from the WOMBAT. The top of the table will be at an angle defined by the positions of the front and back edges of the Table. It moves linearly towards and away from the WOMBAT.

A. TABLE SYSTEM ALGORITHM.

Given the possible movement of the Table and its purpose, an algorithm of the connection process has been determined. Shells are suspended by straps on the monorail carrier. The straps are flexible so they will remain connected to a shell while the shell is connected to the WOMBAT. The connecting process starts by simultaneously raising the front and back of the table until the grasping mechanism has a firm hold on the shell. After the operator uses the video system to visually verify that the shell has been correctly grasped, the table will raise to the height needed to align the shell to the WOMBAT. It will then move the top of the table towards the WOMBAT until the shell is pressed firmly against the WOMBAT faceplate.

When the shell has been completely washed out, the table system will back the shell away from the WOMBAT. The

front of the table will then lower to drain any remaining material from the shell. When the operator believes the shell is drained, the back of the table will lower until the table is level. It will continue to simultaneously lower the front and back of the table until the shell is released to the monorail carrier.

B. HYDRAULIC SYSTEM CONTROL.

The hydraulic systems of the Table and of the WOMBAT are controlled in a similar manner. The hydraulic system of the Table has circuits which control the hydraulic rams with voltage signals that represent positions. The MacBASIC AOT statement is used to send command positions to the circuits. The circuits receive actual positions values from LVDTs. These values can be read with the MacBASIC AIN statement.

All of the four channels of the previously mentioned D/A card are used to control the WOMBAT so another D/A card has been added to the Macsym 200. In the example programs presented later, the number for the slot of the second D/A card is stored in an integer variable named Table'. The integer variables Yfront', Yrear' and Ztop' hold the values for the channels controlling the front, rear and top of the table respectively.

The plant computer's A/D card has sixteen input channels. The WOMBAT system uses the first five channels of this card. The input functions of the A/D card are in the same order as the output functions of the D/A card on the WOMBAT system. This allowed the same variables to be used

for both the input and output channels. It is not possible to use the same constants for the the input and output channels on the Table system because the output channels are on a new D/A card. The input channels used on the Table system are connected after the WOMBAT channels. The output channels on the new D/A card are numbered zero thru three, the output channels are numbered five thru seven. An offset variable, FBoffset', is added to the output channel constants in the example programs. The value of FBoffset' is five because the first input channel used by the Table system is channel five. A different integer variable which contains the slot number for the A/D card is used to add readability in the example programs presented later. This variable, TabFB', contains the same value as the WOMBAT's variable W_IN'.

Figure 18 contains a program to demonstrate the control of the Table system. It allows the three dimensions of the table to be moved separately or the front and back of the table can be moved simultaneously if they are level. It uses conversion factors to change inches to reference voltage. The conversion factors are stored in an array which allows the same sequence of code to control any of the movements of the table.

C. STRAIN GAUGE EXTENDED KEYWORD.

The LAB 40 system on the monorail computer contains an A/D board that aids the Table system by measuring the load on the monorail carrier. Its primary use is to detect the

change in load that occurs when the Table system has risen to the point where it begins to come in contact with a shell or when the shell is released.

```
10 Yfront' = 0 Yrear' = 1  Ztop' = 2
20 Table' = 3  TabFB' = 0  FBoffset' = 5
30 DIM CONV(3)  CONV(Yfront'+1) = 1.027
40 CONV(Yrear'+1) = 1.027  CONV(Ztop'+1) = 2.35
50 cls ! !   @ MENU
60 ! "F)ront  R)ear  T)op  U)p - front/rear  Q)uit"
70 INPUT KEY$
80 IF KEY$ = "Q" THEN END
90 IF KEY$ <> "F" THEN GOTO 100   @ REAR
95   CHNL' = Yfront'  ST$="FRONT"  GOTO 140  @ TABLE MOVE
100 IF KEY$ <> "R" THEN GOTO 110  @ TOP
105   CHNL' = Yrear'  ST$="REAR"  GOTO 140  @ TABLE MOVE
110 IF KEY$ <> "T" THEN GOTO 120  @ UP
115   CHNL' = Ytop'  ST$="TOP"   GOTO 140   @ TABLE MOVE
120 IF KEY$ <> "U" THEN GOTO 130
125   CHNL' = Yfront'  CHNL2' = Yrear'
126   ST$="BOTH UP"   GOTO 140  @ TABLE MOVE
130 GOTO 50  @ MENU
140@  TABLE MOVE
150 Fbak1 = AIN (TabFB',CHNL'+FBoffset',0,0)
160  ! ST$;" at ";Fbak1'/CONV(CHNL'+1):" inches "
170 IF KEY$ <> "U" THEN GOTO 220   @ INPUT
180   Fbak2 = AIN (TabFB',CHNL2'+FBoffset',0,0)
190   IF Fbak1 = Fbak2 THEN GOTO 220   @  INPUT
200      ! "LEVEL Front and Back before Up"
210      GOTO 50  @ MENU
220@  INPUT
230 INPUT "NEW POSITION " X
240  X = X*CONV(CHNL'+1)
250 INPUT "NUMBER OF SUBSTEPS " n
260 TabSTEP = (X - Fbak1) / n
270@
280 FOR I = Fbak1 TO X  step TabSTEP
290   AOT (Table',CHNL') = I
300   IF KEY$ = "U" THEN AOT (Table',CHNL2') = I
320   NEXT I
330 GOTO 50  @ MENU
```

Figure 18. Sample Table Control Program

Figures 19 and 20 contain a pair of subprograms that are used to read the LAB 40-AD and send the result to the plant computer. The procedure Read_Strain_Gauge is included

in the file containing the slave ExKeys. The subroutine in Figure 19 is included in the file containing the master ExKeys. These two subprograms use other subprograms in the ExKey files for communication between the two computers.

```
@ Read_Strain_Gauge  SUBROUTINE
DECLARE (VOLT)
DIM GAIN(4), RANGE(4)
GAIN[2] = 1;  GAIN[3] = 1;  GAIN[4] = 1;

@*** change these variables for diff. setup ***
gain[1] = 1;
   @ range:  -1 -> +-5v, 0 -> 0-10v, 1 -> 0-5v
RANGE[1] = 1;
RANGE[2] = 1;
RANGE[3] = 1;
RANGE[4] = 1;

lbPERvolt = 50

INPUT "ENTER CHANNEL OF LAB 40-A/D (1-4) " CHNL'
GOSUB SendCOM1(CHNL')

GOSUB ReadCOM1(loByte')
GOSUB ReadCOM1(hiByte')

VALUE = hiByte' * 256 + loByte'
VOLT = VALUE/4096
IF RANGE(CHNL') = -1 THEN VOLT = VOLT * 10 - 5
IF RANGE(CHNL') = 0  THEN VOLT = VOLT * 10
IF RANGE(CHNL') = 1  THEN VOLT = VOLT * 5

VOLT = VOLT/GAIN(CHNL')
RETURN
```

Figure 19. Read Strain Gauge Subroutine.

When these two subprograms are executed on their computer, the master version prompts the user for the channel to be read. The LAB 40-AD has four channels. The first channel has an amplifier that is hardware adjustable. It can be set to amplify the signal at channel 1 before it is read. The amount of amplification is known as the gain.

The value read at this channel should be divided by the gain to convert the value to the actual value at the channel.

```
procedure Read_Strain_Gauge;
VAR
  chnl,value: integer;
  loByte,HiByte:  byte;
begin
  ReadCOM1(loByte);
  chnl:= ord(lobyte)

  Port[BoardSel] := ADboard;
  Port[A+chnl] := 1;  (starts A/D process on chnl)
  Port[B] := 0;  (sets low byte select)
  loByte := Port[A];  (inits 8255 input reg)
  loByte := Port[A];  (read low byte)
  Port[B] := 1;  (sets hi byte select)
  hiByte := Port[A];  (read high byte)

  SendCOM1(lobyte);
  SendCOM1(hibyte);

  value := hiByte*256 + loByte;
  writeln('value read = ',value);
  Port[BoardSel] := HPboard;
end.
```

Figure 20. Read Strain Gauge Procedure.

The slave Read_Strain_Gauge does not convert the value read on channel one. It reads any one of the channels and sends that value to the plant computer. The master version converts the values it receives the value to their correct values. Program constants in the master Read_Strain_Gauge are used for the gain settings for all four channels. The gain value for the first channel has been set to one in the listing. The other three channels are not amplified so their gain constant should always be one. This leaves their values unchanged.

If the LAB 40-AD hardware is changed so that channel one has a different gain value, then the gain constant in the master Read_Strain_Gauge needs to be set to the new value. This convention allows the hardware to be changed without changing the slave Read_Strain_Gauge's code.

Only one program should ever be executed on the monorail computer because of its dedicated purpose. This program has to control the electric motor of the monorail carrier and communicate with the plant computer. It should also allow the monorail carrier's strain gauges to be read and the values sent to the plant computer.

Figure 21 contains an example program for the plant computer which shows the use of the Read_Strain_Gauge subprograms while controlling the Table. It raises the table until it detects the grasping of a shell. When the table has risen to the point where it touches the shell, the values from Read_Strain_Gauge will start to change. A tolerence level on the possible deviations of readings from the strain gauges avoids false indications that might end this example program too soon.

This program assumes that the monorail computer is running a program which constantly executes the Read_Strain-_Gauge procedure. Before a Monorail program delivers a shell to the washout area, it should lower the table so the shell can be moved over the table. The program in Figure 21 also performs this function.

```
10 Yfront' = 0 Yrear' = 1  Ztop' = 2
20 Table' = 3  TabFB' = 0  FBoffset' = 5
30 DIM CONV(3)  CONV(Yfront'+1) = 1.027
40 CONV(Yrear'+1) = 1.027  CONV(Ztop'+1) = 2.35
50 UPSTEP = .05  TOLERENCE = .01
60 ALIGNMENT = 2.345
70 @@@ SETS THE TABLE TO THE LOWEST, LEVEL POSITION
80 AOT (Table',Yfront') = -10
90 AOT (Table',Yrear') = -10
100 @@@ MONORAIL PROGRAM SHOULD DO THIS

110 GOSUB READ_STRAIN_GAGE(REF_WEIGHT)

120@ MOVE TABLE UNTIL STRAIN DECREASES
130 GOSUB READ_STRAIN_GAGE(WEIGHT)
140 IF ABS(REF_WIEGHT - WIEGHT) > TOLERANCE THEN
      GOTO 200
150 Fbak1 = AIN (TabFB',Yfront'+FBoffset',0,0)
160 NEXT_STEP = Fbak1 + UP_STEP
170 AOT (Table',Yfront') = NEXT_STEP
180 AOT (Table',Yrear') = NEXT_STEP

190 GOTO 120

200 END
```

Figure 21. Table Control Program with Strain Detection.

The Read_Strain_Gauge subroutine should be modified when used by this program. The A/D channel read during this program will always be the same. The subroutine currently prompts for the channel number. This statement should be replaced to an assignment statement which sets the varible CHNL' to the channel number used.

This sample program illustrates the use of the Read-_Strain_Gauge subprograms. The table rises slightly then these subprograms are used to compare the new value to a reference value that was read before the table started to rise. This program performs only a beginning portion of the algorithm required to connect a shell to the WOMBAT.

V. CONCLUSION.

This thesis presents the methods of computer control used on the pilot PBX Washout Plant. Two of the components of the plant, the WOMBAT and the Table system, are hydraulically powered systems. The third component, the Monorail system, involves the control of an electric motor and communication between two computer systems.

Only the basic control algorithms were presented for two reasons. The first reason is the delivery system's stage of completion at the time of this thesis. The second reason is the frequent updates to the PBX Washout Plant components and their programs. The algorithms presented should remain valid and useful as long as the plant is under development or in operation. Most future changes to the physical design of any one of the components should only involve parameter changes in its control program.

At the time of this thesis, the delivery system was under development. Most of the installation of the components had been performed, but some work still remains. The development of the infrared communication system was hindered by hardware breakdowns. The electric motor's gear reduction system also had breakdowns which held up the construction of the pilot plant. These problems also restricted the development of the control programs because the testing of the programs required the completion of the installations.

Each control program for the different components can be developed separately because the algorithm of the plant only controls one component at a time. The status of the other components has to be considered during the development and execution of a program. For example, the WOMBAT program must execute in a manner that will not cause the delivery system to move the shell during a washout. The delivery programs should be developed such that once they have delivered and connected a shell to the WOMBAT, their execution can halt and the program for the next component can execute without affecting the previously performed actions. The integration of the separate control programs is important for the smooth operation of the pilot plant.

The plant operators' experience and knowledge is a factor of the control programs. Only highly qualified personnel who are familiar with explosives and with the operations of the plant's programs will ever operate the plant. The programs should still be user-friendly and convenient to use but the required parameters of the programs are complicated and can be confusing. The pilot plant is a research project; therefore several considerations of program development of production software are not performed to the same degree.

It is important to consider that pilot plant research should eventually lead to the development of a specialized plant that will be used to dispose a large stockpile of munition shells. The plant computer system lacks the

sophistication that will be required on a production plant. This computer was considered to be the best solution to controlling the WOMBAT system when it was first developed but now there are many computer systems available that would perform better. There is a factor of money and time required to install a new plant computer at the pilot plant so this system will likely have to be tolerated during the rest of the research and developments.

BIBLIOGRAPHY

1. Summers, Dr. David A., Hammerand, Ed, LaDiere, Ray. "A Phase One Report on Contract N00164-86C-0181." Rolla, MO: Rock Mechanics and Explosive Research Center, University of Missouri-Rolla.

2. Tyler, John. Personal Conversations. Rolla, MO: Rock Mechanics and Explosive Research Center, University of Missouri-Rolla, 1986-88.

3. Borland International, Inc. "Turbo Pascal Reference Manual" Version 3.0. Scotts Valley, CA: Borland International, Inc., 1985.

4. Analog Devices, Inc. "MACBASIC 3 Concepts." Norwood, MA: Analog Devices, Inc., 1983.

5. Analog Devices, Inc. "MACBASIC 3 Programming." Norwood, MA: Analog Devices, Inc., 1983.

6. Analog Devices, Inc. "MACBASIC 3 Command Reference." Norwood, MA: Analog Devices, Inc., 1983.

7. Analog Devices, Inc. "MACSYM 150 Operation." Norwood, MA: Analog Devices, Inc., 1983.

8. Analog Devices, Inc. "MACSYM 200 Operation." Norwood, MA: Analog Devices, Inc., 1983.

9. Analog Devices, Inc. "DS1100/101 Dual Serial Interface Card." Norwood, MA: Analog Devices, Inc., 1984.

10. Analog Devices, Inc. "MACSYM ADIO LIBRAY Operation Manual," AOC01/02 ANALOG OUTPUT CARDS, Rev 2. Norwood, MA: Analog Devices, Inc., 1980.

11. Analog Devices, Inc. "MACSYM ADIO LIBRAY Operation Manual," AIM03 ANALOG INPUT CARD. Norwood, MA: Analog Devices, Inc., 1980.

12. Norton, Peter. Inside the IBM PC. New York: Prentise Hall Press, 1986

13. Intel. "iAPX 86/88,186/188 User's Manual." Santa Clara, CA: Intel Corporation, 1985.

14. Eccles, William J. Microprocessor Systems A 16-Bit Approach. Reading, MA: Addison-Wesley Publishing Company, 1985.

15. Stone,Harold S. Microcomputer Interfacing. Reading, MA: Addison-Wesley Publishing Company, 1983.

16. Farady. "OEM PC BUS and AT BUS Product Catalog." Sunnyvale, CA: Faraday, 1986.

17. International Business Machines Corporation. "Technical Reference" Personal Computer XT and Portable Personal Computer. Boca Raton, FL: International Business Machines Corporation, 1984.

18. Computer Continuum. "LAB 40 Development System." Daly City, CA: Computer Continuum.

19. Computer Continuum. "LAB 40-2 12 Bit A/D Module." Daly City, CA: Computer Continuum.

20. Computer Continuum. "LAB 40-6 Smart Two Axis Motor Controller Module." Daly City, CA: Computer Continuum.

21. International Business Machines Corporation. "Disk Operating System" Version 2.10. Boca Raton, FL: International Business Machines Corporation, 1983.

22. Hewlett Packard. "General Purpose Motion Control IC" Technical Data. Pala Alto, CA: Hewlett Packard, 1985.

23. Jen Sriwattanathamma. "WOMBAT Control Software." Rolla, MO: Rock Mechanics and Explosive Research Center, University of Missouri-Rolla. Updates: John Tyler, Mark Hallett, Scott Sharp.

24. National Semiconductor Corporation. "Series 32000 Databook." Santa Clara, CA: National Semiconductor Corporation, 1986

25. Jameco Electronics. "RS-232 Card Manual." Belmont, CA: Jameco Electronics, 1987.

26. da Cruz, Frank. "Kermit Protocol Manual" Sixth Edition. New York: Columbia University Center for Computing Activities, 1986.

27. da Cruz, Frank. KERMIT A File Transfer Protocol. Bedford, MA: Digital Press, 1987.

28. Jennings, Fred. Practical Data Communications: Modems, Networks and Protocols. Osney Mead, Oxford: Blackwell Scientific Publications, 1986.

29. Stallings, William. Handbook of Computer-Communications Standards. New York: Macmillian Publishing Comany, 1987.

30. Intel. "Microsystem Components Handbook" Volume II. Santa Clara, CA: Intel Corporation, 1986.

31. National Semiconductor Corporation. "Logic Databook" Volume 1. Santa Clara, CA: National Semiconductor Corporation, 1984

VITA

Scott Cameron Sharp was born on September 14, 1962 in Rapid City, South Dakota. He received his primary and secondary education in Blue Eye, Missouri. In August 1981, he entered the University of Missouri-Rolla and received a Bachelor of Science in Computer Science in December 1985. During this time, he was a member of Upsilon Pi Epsilon, Kappa Mu Epsilon and other various campus organizations.

After receiving his B.S. degree, he entered the graduate school at the University of Missouri-Rolla in pursuit of the Master of Science Degree in Computer Science.

APPENDIX A

SLAVE EXTENDED KEYWORDS

```
const
{*********************************************************}
{***                                                   ****}
{***      R S 2 3 2    R E G I S T E R    P O R T S   ****}
{***                                                   ****}
{*********************************************************}
   RS232_base = $3F8;
   Tx_Buffer = $3F8;              {RS232_base + 0}
   Rx_buffer = $3F8;              {RS232_base + 0}
   Intr_enable_reg = $3F9;        {RS232_base + 1}
   Intr_ID_Reg = $3FA;            {RS232_base + 2}
   line_Control_reg = $3FB;       {RS232_base + 3}
   Modem_Control_Reg = $3FC;      {RS232_base + 4}
   Line_Status_Reg = $3FD;        {RS232_base + 5}
   Modem_Status_Reg = $3FE;       {RS232_base + 6}

{*********************************************************}
{***                                                   ****}
{***   H P - 1 0 0 0   R E G I S T E R   P O R T S   ****}
{***                                                   ****}
{*********************************************************}
   A = 736;  {connects to addr/data lines of HP HCTL-1000}
   B = 740;  {Bits- 0 r/w*,  1-2 intEn,  3 RESET*,
              4-5 motor bus 1 selects, 6-7 M.B. 2 select}
   Board_Select = 744;  {3 to 8 decode for board select}
   control = 748;  {control port of 8255}

   Flag_Reg = $00;
   Program_Reg = $05;
   Status_Reg = $07;
   C_Pos_MSB = $0C;
   C_Pos_MB = $0D;
   C_Pos_LSB = $0E;
   Timer_Reg = $0F;
   Act_Pos_MSB = $12;
   Act_Pos_MB = $13;
   Act_Pos_LSB = $14;
   Zero_Reg = $20;
   Pole_Reg = $21;
   Gain_Reg = $22;
   Accel_Reg_LSB = $26;
   Accel_Reg_MSB = $27;
   Max_Vel_Reg = $28;
   Fin_Pos_MSB = $29;
   Fin_Pos_MB = $2A;
   Fin_Pos_LSB = $2B;

   pulses_per_Rev = 2640;
```

```
var
   Act_pos,Fpos:                    integer;
   time,gain,pole,zero:             integer;
   speed,Accel:                     integer;
   position_packet:                 integer;
   Move_End_Flag,ACK,NAK:           integer;



{************************************************************}
{************************************************************}
{***                                                      ***}
{***   RS 232   C O M M U N I C A T I O N    SECTION     ***}
{***                                                      ***}
{***              HARDWARE LEVEL PROCEDURES               ***}
{***                                                      ***}
{************************************************************}
{************************************************************}


            {******************************}
            {**   Read_Line_Status_Reg   **}
            {******************************}

procedure Read_Line_Status_Reg
    (VAR  R_full, overrun, parity, framing,
          break, empty_TB, empty_TR:  boolean);
var  x:  byte;
begin
   x := port[Line_Status_Reg];
   R_full := false;
     if (x AND $01) = $01 then  R_full := true;
   overrun := false;
     if (x AND $02) = $02 then  overrun := true;
   parity := false;
     if (x AND $04) = $04 then  parity := true;
   framing := false;
     if (x AND $08) = $08 then  framing := true;
   break := false;
     if (x AND $10) = $10 then  break := true;
   empty_TB := false;
     if (x AND $20) = $20 then  empty_TB := true;
   empty_TR := false;
     if (x AND $40) = $40 then  empty_TR := true;
  end;
```

```
            {*********************}
            {**   Read_Driver   **}
            {*********************}

procedure Read_Driver (VAR i: integer;
                       VAR Read_errors: boolean);
var  x :  byte;
     R_full, E1,E2,E3,E4, B1,B2,  dataRDY:   boolean;
begin
  Read_errors := false;  dataRDY := false;

  while (NOT Read_problems) AND (NOT dataRDY) do begin
          {*  allows break key to work  *}
     if keypressed then read(x);

    read_Line_Status_Reg (R_full, E1,E2,E3,E4, B1,B2);
     if  E1 OR E2 OR E3 OR E4  then
         Read_errors := true
       else
         if R_full then
            dataRDY := true;
     end;  {while}

   x := port[Rx_buffer];
   if (Not read_errors) then
      i := ord(x);

   if read_problems then begin
      writeln('ERROR in read from COM1');
      if ovr then writeln('    overrun error occured');
      if par then writeln('    parity error occured');
      if frm then writeln('    framing error occured');
      if brk then writeln('    break error occured');
      end;

   end;  { proc Read_Driver  }



            {*********************}
            {**   Send_Driver   **}
            {*********************}

procedure Send_Driver (x: integer);
   var  B1, B2, B3, B4, B5, Empty_TB, B7:   boolean;
begin
   Empty_TB := false;
   while (NOT Empty_TB) do
     Read_Line_Status_Reg(B1,B2,B3,B4,B5,Empty_TB,B7);

  port[Tx_buffer] := Lo(x);
  end;  { proc Send_Driver  }
```

```
            {**********************************}
            {**   SetUpCOM1  K E Y W O R D   **}
            {**********************************}
procedure setUpCom1;
type
 registers = record
   AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS: INTEGER;
   end;
var   reg : registers;
      x:  byte;
      baud, parity, stop, data: integer;
begin
  {*** CHANGE THESE TO SET UP DIFFERENTLY}
  baud := 1200;  { choices>>> 300,1200,4800,9600 }
  parity := 0;   { choices>>> 0:none, 1:odd, 2:even }
  stop := 2;     { choices>>> 1 or 2 }
  data := 8;     { choices>>> 7 or 8 }


  reg.AX := $0003;
  if data = 7 then reg.AX := $0002;

  if stop = 2 then reg.AX := reg.AX + 4;

  if parity = 1 then reg.AX := reg.AX + 8;
  if parity = 2 then reg.AX := reg.AX + 24;

  CASE baud of
   9600:   reg.AX := reg.AX + 224;
   4800:   reg.AX := reg.AX + 192;
   1200:   reg.AX := reg.AX + 128;
   else    reg.AX := reg.AX + 64;
  end;

   {***  setup com1 to communicate with macsym   ***}
  reg.DX := $00;     {com1}
  Intr($14,regs);

      {  clear 7th bit of line control  }
  x := port[Line_control_Reg];
  x:= x AND $7F;
  port[Line_control_Reg] := x;

      {  Disable COM1 Interrupts  }
  x:= $00;  port[Intr_enable_Reg] := x;
  end;

      {  initialize COM Variables  }
  ACK := 6;   NAK := 7;
  position_packet := 2;
  Move_End_Flag := 3;

end;  {setUpCom1}
```

```
            {*********************************}
            {**   ReadCOM1  K E Y W O R D   **}
            {*********************************}
procedure ReadCOM1(VAR data: integer);
var
  d1,d2: integer;       E1,E2,looping:  boolean;
begin
   looping := true;
   while looping do begin
      Read_Driver(d1,E1);      {data field}

      if E1 then
         d1 := NAK
        else
         data := d1;

      Send_Driver(d1)

      Read_Driver(d2,E2);
      if (d2 <> ACK) or E2 then   d2 := NAK;

      Send_Driver(d1)

      if d2 = ACK then   looping := false;

      end;  {while}

   end;

            {*********************************}
            {**   SendCOM1  K E Y W O R D   **}
            {*********************************}
procedure SendCOM1(data: integer);
var
  d1,d2,d3: integer;
  E1,E2,looping:   boolean;
begin
   looping := true;
   while looping do begin
      Send_Driver(data);
      Read_Driver(d1,E1);

      d2 := NAK;
      if (data = d1) and Not E1 then d2 := ACK;

      Send_Driver(d2);
      Read_Driver(d3,E2);

      if E1 then
            writeln(chr(7),'*** ERROR - RESEND LAST BYTE')
           else
            if d3 = ACK then  looping := false;
     end; {while}
   end;
```

```
{*********************************************}
{*********************************************}
{***                                       ***}
{***    ADVANCE  COMMUNICATION  KEYWORDS   ***}
{***                                       ***}
{*********************************************}
{*********************************************}


procedure Get_3Byte_Integer_from_COM1
     (VAR h,m,l:  integer;
     VAR t:  real);
begin
   ReadCOM1(l);
   ReadCOM1(m);
   ReadCOM1(h);

   t := h*65536.0 + m*256.0 + l;
      {convert negative numbers to real}
   if t > 8388607.0 then
       t := t - 16777216.0;
   gotoXY(1,4);
   end;



procedure Send_3Byte_Integer_over_COM1
   (h,m,l:  integer);
begin
   SendCOM1(l);
   SendCOM1(m);
   SendCOM1(h);
   end;
```

```
{************************************************************}
{************************************************************}
{***                                                      ***}
{***     M O T O R   C O N T R O L   S E C T I O N        ***}
{***                                                      ***}
{***            HARDWARE LEVEL PROCEDURES                 ***}
{***                                                      ***}
{************************************************************}
{************************************************************}


procedure HPwrite(addr,data: integer);  begin
   port[B] := 8;       {lowers Rd bit and +8 for no reset}
   port[A] := addr;    {send addr of reg}
   port[A+1] := data; {send data to reg}
   end;



function HPread(addr: integer): integer;  begin
   port[B] := 9;         {sets Rd bit and +8 for no reset}
   port[A] := addr;      {send addr of reg}
   port[A+1] := 0;       {strobe CS low}
   HPread := port[A+1]; {start read}
   delay(110);           {pause for 8255's read}
   HPread := port[A+1]; {Read data form PortA}
   end;




procedure hard_reset;  begin
   port[B] := 0;    {bit 3 goes low}
   delay(500);
   port[B] := 8;    {bit 3 goes high}
      { HP Soft reset automatically occurs  }
      { NOT the included procedure soft reset  }
      { automatically goes into init/idle mode }
   end;
```

```
procedure Restore_HPregisters;   begin
   HPwrite(Timer_Reg,time);
   HPwrite(Gain_Reg,gain);
   HPwrite(Zero_Reg,zero);
   HPwrite(Pole_Reg,pole);

   HPwrite(Accel_Reg_MSB,Hi(Accel));
   HPwrite(Accel_Reg_LSB,Lo(Accel));

   HPwrite(Max_Vel_Reg,speed);

   HPwrite(Act_pos_MB,0);   {clears actual position}

      {clears command position}
   HPwrite(C_Pos_MSB,0);   HPwrite(C_Pos_MB,0);
   HPwrite(C_Pos_LSB,0);
      {clears final position}
   HPwrite(Fin_Pos_MSB,0); HPwrite(Fin_Pos_MB,0);
   HPwrite(Fin_Pos_LSB,0);
   end;

{************************************************************}
{************************************************************}
{***                                                      ***}
{***    M O T O R   C O N T R O L   K E Y W O R D S       ***}
{***                                                      ***}
{************************************************************}
{************************************************************}


          {*********************************}
          {**   Init_LAB40  K E Y W O R D   **}
          {*********************************}
procedure init_lab40;   begin
     {sets 8255 up as chA in/out, chB latched output}
   port[control] := 192;
   port[Board_Select] := 2;          {select board 2}

   hard_reset;

      {***  set HP-1000 registers vars   ***}

   time := 255; { represents 2048 microSec, (time*2E6/16)-1}
   gain := 30;
   pole := 64;   {*  internally divided by 256  *}
   zero := 244;  {*  internally divided by 256  *}

   speed := 128;    { revs/sec  }
   Accel := trunc(0.01*256);  {pulses/sqr(sec)*256 }

   Restore_HPregisters;

   end;
```

```
            {**********************************}
            {**   Soft_Reset  K E Y W O R D   **}
            {**********************************}
procedure Soft_reset;  begin
   HPwrite(Program_Reg,0);   {executes soft reset}
      { Changes filter parameters }
      { Changes sample timer reg }
      { Clears status reg. }
      { Clears position regs. }
      { goes to init./idle mode }
   Restore_HPregisters;
      {restores time, filters and vel/accel to last values }
   end;




            {******************************************}
            {**   Set_Filter_Variables  K E Y W O R D   **}
            {******************************************}
procedure Set_Filter_Variables;  begin

        {ORDER -->  gain, pole zero}

   ReadCOM1(gain);
   HPwrite(Gain_Reg,gain);
     writeln('gain = ',gain);

   ReadCOM1(pole);
   HPwrite(Pole_Reg,pole);
     writeln('pole = ',pole);

   ReadCOM1(zero);
   HPwrite(Zero_Reg,zero);
     writeln('zero = ',zero);
   end;



            {**********************************}
            {**   Set_Time  K E Y W O R D   **}
            {**********************************}
procedure Set_Time;  begin
   ReadCOM1(time);
   HPwrite(Timer_Reg,time);  writeln('time = ',time);
   end;
```

```
            {****************************************}
            {**   Set_Acceleration  K E Y W O R D   **}
            {****************************************}
procedure Set_Acceleration;
VAR x,y:  integer;
  begin
   ReadCOM1(x);
   ReadCOM1(y);
      {Hi byte is int portion and Lo byte is fract * 256}
   HPwrite(Accel_Reg_MSB,x);
   HPwrite(Accel_Reg_LSB,y);
   writeln('Accel = ',x*1.0 + (y*1.0)/256.0:15:9);
   end;



            {**************************************}
            {**   Set_Velocity  K E Y W O R D   **}
            {**************************************}
procedure Set_Velocity;  begin
   ReadCOM1(speed);
      {revs/sec * pulses/rev * sec/sample  =  pulses/sample}
   if speed > $7f then begin
      writeln('speed too large   ==>',speed);
      speed := $7F;
      end;

   HPwrite(Max_Vel_Reg,speed);
     writeln('velocity = ',speed);
   end;



            {********************************************}
            {**   Enter_Final_Position  K E Y W O R D   **}
            {********************************************}
procedure  Enter_Final_Position;
   var  hiB,midB,loB: integer;  Fpos: real;
begin
   clrscr;
   gotoxy(1,16);  writeln('Ready for position',' ':20);
   Get_3Byte_Integer_from_COM1(hiB,midB,loB, Fpos);
   gotoxy(1,16);
      write('Final position =  ',Fpos:16:1,' ':20);
   HPwrite(Fin_Pos_LSB,hiB);
   HPwrite(Fin_Pos_MB,midB);
   HPwrite(Fin_Pos_MSB,loB);
   end;
```

```
            {*****************************************}
            {**   Display_Position  K E Y W O R D   **}
            {*****************************************}

procedure display_position;
var  lsb,mb,msb, Act_lsb :integer;
     Act_pos:  real;
 begin
     {latches other two bytes}
   Act_lsb := HPread(Act_Pos_LSB);

     {latches other two bytes}
   lsb := HPread(C_Pos_LSB);

   mb := HPread(C_Pos_MB);
   msb := HPread(C_Pos_MSB);
   Act_pos := lsb + mb*256.0 + msb*65536.0;
     {convert negative numbers to real}
   if Act_pos > 8388607.0 then
      Act_pos := Act_pos - 16777216.0;
   write('Command pos=':5,Act_Pos:16:1,' ':7);



   lsb := Act_lsb;
   mb := HPread(Act_Pos_MB);
   msb := HPread(Act_Pos_MSB);

   sendCOM1(position_packet);
   Send_3Byte_Integer_over_COM1(msb,mb,lsb);

   Act_pos := lsb + mb*256.0 + msb*65536.0;
     {convert negative numbers to real}
   if Act_pos > 8388607.0 then
     Act_pos := Act_pos - 16777216.0;
   writeln(' Act pos = ',Act_Pos:16:1);
  end;
```

```
            {************************************}
            {**   Execute_Move  K E Y W O R D   **}
            {************************************}
procedure execute_move;

   procedure Start_Trapezoidal_Move;  begin
          {set trap. flag and starts movement}
      HPwrite(Flag_Reg,$08);
          {set porgram conter to control mode}
      HPwrite(Program_Reg,3);
      end;

   function In_Trapezoidal_profile: boolean;
    var
       i: byte;
       status: integer;
    begin
      In_Trapezoidal_Profile := false;
      status := HPread($07);
      if (status and $10) = $10 then
          {status reg 5th bit set when finished}
         In_Trapezoidal_Profile := true;
      end;

begin
   clrscr;
   Start_Trapezoidal_Move;
   repeat   display_position;
      until not  In_Trapezoidal_Profile;
   display_position;
   sendCOM1(Move_End_Flag);
   writeln('***  MOVE COMPLETED   ****':10,' ':20);
   end;



            {*****************************}
            {**   Quit  K E Y W O R D   **}
            {*****************************}
procedure quit; begin
   hard_reset;
   port[Board_Select] := 0;          {select Board 0}
   port[control] := 0;   {turn off Lab 40}
   clrscr;
   halt;
   end;
```

```
            {**************************}
            {**   Read_Strain_Gauge   **}
            {**************************}

procedure Read_Strain_Gauge;
VAR
  chnl,value: integer;
  loByte,HiByte:  byte;
begin
  ReadCOM1(loByte);
  chnl:= ord(lobyte)

  Port[BoardSel] := ADboard;
  Port[A+chnl] := 1;  {starts A/D process on chnl}
  Port[B] := 0;  {sets low byte select}
  loByte := Port[A];  {inits 8255 input reg}
  loByte := Port[A];  {read low byte}
  Port[B] := 1;  {sets hi byte select}
  hiByte := Port[A];  {read high byte}

  SendCOM1(lobyte);
  SendCOM1(hibyte);

  value := hiByte*256 + loByte;
  writeln('value read = ',value);
  Port[BoardSel] := HPboard;
end.
```

APPENDIX B

MASTER EXTENDED KEYWORDS

```
319 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
320 @  I N I T___L A B 4 0            (ONLY SETS VARIABLES)
321 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
325  PperR = 660*4
330  TIME' = 255
335  VEL = 127
340  ACCEL = 0.5/256.0 * (2E6/(16.0*(TIME'+1)))^2 / PperR
345  GAIN' = 45
350  POLE' = 64
355  ZERO0' = 244
360 RETURN




399 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
400 @  E N T E R   F I N A L   P O S I T I O N
410 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
420    ON ERROR ERRNO',450
430    INPUT "ENTER FINAL POSITION   <-8388608,8388607> "
             FINAL
440    OFF ERROR  GOTO 470
450    @ @ @ INPUT ERROR
460    PNT BELL'   GOTO 430
470    GOSUB 2310 (FINAL)  @Send 3 Byte Integer
590    RETURN




600@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
610@   E X E C U T E   C O M M A N D
620@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
630 @ @ @   POSITION FEEDBACK   @ @ @
640    GOSUB 1840 (A')   @  READ A' FROM COM1
650    IF A' = MOVE_END' THEN
          PNT 7 ! "  ---  MOVE HAS BEEN COMPLETED ---";
          RETURN
660    GOSUB 2220
670    ! ACT_POS
680    GOTO 630
```

```
690 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
700 @  D I S P L A Y   P O S I T I O N
710 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
720    GOSUB 1840 (A')     @  READ A' FROM COM1
730    GOSUB 2220
740    ! "ENCODER COUNTS ==> ";ACT_POS !
750    RETURN



760 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
770 @   S E T   V E L O C I T Y    @
780 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
790    MINV = 0.5*2E6 / (PperR*16*(TIME'+1))
800    MAXV = 127*2E6 / (PperR*16*(TIME'+1))
810    ! "Enter Revs/Sec of the MOTOR <"MINV; ","; MAXV;
          "> RETURN: MAX ";
820    INPUT VEL
830    IF VEL = 0 THEN VEL = MAXV A' =127  GOTO 870
840    IF (VEL > MAXV) THEN PNT BELL' GOTO 810
850    A' = INT(VEL *PperR*16*(TIME'+1)/2000000  + 0.5)
860       @ PULSES/SAMPLE = REVS/SEC*PULSES/REV*SEC/SAMPLE
870    ! "     ";VEL;" REVS/SEC CONVERTED TO ";A';
          " PULSES/SAMPLE"
880    GOSUB 1570 (A')    @ SEND THE PULSES/SAMPLE
890    RETURN



900 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
910 @   S E T   A C C E L E R A T I O N   @
20 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
930    MINA = 1.0/256.0 * (2E6/(16.0*(TIME'+1)))^2 / PperR
940    MAXA = 127.999 * (2E6/(16.0*(TIME'+1)))^2 / PperR
950    ! ! "Enter <"MINA;","MAXA;
          "> for Revs/Sec^2 of the MOTOR ";
960    INPUT ACCEL
970    IF ACCEL = 0 THEN ACCEL = MINA GOTO 990
980    IF (ACCEL < MINA) OR (ACCEL > MAXA) THEN
          PNT BELL' GOTO 950
990    tempA = ACCEL *PperR*(16.0*(TIME'+1)/2000000.0)^2
1000       @PULSES/SAMPLE^2=REVS/SEC*PULSES/REV*SEC/SAMPLE^2
1010    HI' = INT(tempA)
1015    LO' = INT((tempA-INT(tempA))*256.0 + 0.5)
1020    ! "     ";ACCEL;" REVS/SEC^2 = "; HI' + LO'/256.0;
1030    ! " PULSES/SAMPLE:  HI=";HI';", LO=";LO'
1040    GOSUB 1570 (HI')    @ SEND INTEGER PORTION OF ACCEL
1050    GOSUB 1570 (LO')    @ SEND FRACT PORTION OF ACCEL
1060    RETURN
```

```
1070 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1080 @   S E T   F I L T E R    V A R s   @
1090 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1100    INPUT "Enter GAIN <0,255>  (TYPICALLY 30-60) ",
           GAIN'
1110     IF GAIN' = 0 THEN GAIN' = 45 GOTO 1130
1120     IF (GAIN' < 0) OR (GAIN' > 255) THEN
           PNT BELL' GOTO 1100
1130     GOSUB 1570 (GAIN')      @ SEND GAIN
1140    INPUT "Enter POLE <0,255>  (TYPICALLY 64) ",POLE'
1150     IF POLE' = 0 THEN POLE' = 64 GOTO 1170
1160     IF (POLE' < 0) OR (POLE' > 255) THEN
           PNT BELL' GOTO 1140
1170     GOSUB 1570 (POLE')      @ SEND POLE
1180    INPUT "Enter ZERO <0,255>  (TYPICALLY 244) ",ZERO0'
1190     IF ZERO0' = 0 THEN ZERO0' = 244 GOTO 1210
1200     IF (ZERO0' < 0) OR (ZERO0' > 255) THEN
           PNT BELL' GOTO 1180
1210     GOSUB 1570 (ZERO0')      @ SEND POLE
1220    RETURN



1230 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1240 @   S E T   T I M E    R E G   @
1250 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1260    INPUT "Enter TIME <15,255> (RETURN: 255=2048 uSEC)",
           TIME'
1270     IF TIME' = 0 THEN TIME' = 255
1280     IF (TIME' < 15) OR (TIME' > 255) THEN
           PNT BELL' GOTO 1260
1290     GOSUB 1570 (TIME')      @ SEND TIME
1300     ! TIME';" = ";8.0*(TIME'+1);" uSECS"      !
1310    RETURN



1320 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1330 @     S O F T    R E S E T     @
1340 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1350    CLS ! ! ! !
1360    ! "SOFT RESET EXECUTED"
1370    ! "    *  Status reg cleared"
1380    ! "    *  Command, Actual and Final Positions RESET"
1390    ! "    *  Filter, Timer, Accel and Vel reg UNchanged"
1400    ! "    *  Placed in init./idle mode"
1410    ! ! ! !"    - - -  PRESS A KEY TO CONTINUE"
1420    INP JUNK'  CLS
1430    RETURN
```

```
1440 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1450 @     H A R D   R E S E T     @
1460 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1470    CLS ! ! ! !
1480    ! "HARD RESET EXECUTED"
1490    ! "   Internal soft reset occurs automatically
1500    ! "   *** Filter, Timer, position regs are default"
1510    ! "   *** soft reset COMMAND restores their values"
1511    ! ! ! !"   - - -  PRESS A KEY TO CONTINUE"
1512    INP JUNK'  CLS
1513    RETURN

1520 @@@@@@@@@@@@@@@@@@@@@@@@@
1521 @   S E T U P C O M 1    @
1522 @@@@@@@@@@@@@@@@@@@@@@@@@
1530 RS232IN' = 5: RS232OUT' = 6   ACK'=6  NAK'=7
1550 MOVE_END' = 113  POS_PAC' = 112   BELL' = 7
1555 OPENR:RS232IN' "$QTI:0"
1560 RETURN

1569 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1570 DECLARE (A')          @      SEND COM1     @
1580 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1590@
1600 ON ERROR ERRNO',1730,LINE'
1610 PNT:RS232OUT' A'
1620 INP:RS232IN' C'
1630 IF C' = A' THEN GOTO 1690
1640    @ ELSE WRONG CHAR RECEIVED
1650 PNT:RS232OUT' NAK'
1660 INP:RS232IN' B'
1670 IF B' <> NAK' THEN
        ! "  ***   RETURNING NAK NOT RECEIVED, ";
           B';" WAS RECEIVED INSTEAD"
1680 GOTO 1610
1690  PNT:RS232OUT' ACK'
1700  INP:RS232IN' B'
1710 IF B' <> ACK' THEN
        ! "  ***   RETURNING ACK NOT RECEIVED, ";
           B';" WAS RECEIVED INSTEAD"
1720  OFF ERROR  RETURN

1730@   * * * * *   COMMUNICATION ERROR HANDLING   * * * * *
1740 IF ERRNO' = 22 THEN GOTO 1790   @@@  PARITY  @@@
1750 IF ERRNO' = 23 THEN GOTO 1800   @@@  TIMEOUT @@@
1760 ! "ERROR # ";ERRNO';" OCCURED AT LINE ";LINE'
1770 END
1780 @@@ PARITY ERROR @@@
1790    ! "PARITY ERROR"   GOTO 1640   @ SEND NAK
1800 @@@ TIME OUT ERROR
1810    ! "TIMEOUT"  END
1820@
```

```
1830 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1840 DECLARE (A')          @     READCOM1     @
1850 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
1860@
1870 ON ERROR ERRNO',1950,LINE'
1880 INP:RS232IN' A'
1890 PNT:RS232OUT' A'
1900 ON ERROR ERRNO',1950,LINE'
1910 INP:RS232IN' B'
1920 IF B' <> ACK' THEN PNT:RS232OUT' NAK'   GOTO 1880
1930 PNT:RS232OUT' ACK'
1940 OFF ERROR   RETURN

1950@   * * * * *   FIRST INP ERROR HANDLING   * * * * *
1960 IF ERRNO' = 22 THEN GOTO 2010   @@@  PARITY  @@@
1970 IF ERRNO' = 23 THEN GOTO 2030   @@@  TIMEOUT @@@
1980 IF ERRNO' = 166 THEN GOTO 2050   @@@  FRAMING @@@
1990 ! "ERROR # ";ERRNO';" OCCURED AT LINE ";LINE'
2000 END
2010 @@@ PARITY ERROR @@@
2020    ! "PARITY ERROR"   END
2030 @@@ TIME OUT ERROR
2040    ! "TIMEOUT"  END
2050 @@@ FRAMING ERROR  @@@
2060    ! "FRAMING ERROR OCCURED AT LINE ";LINE'
2070    A' = NAK' GOTO 1890  @@@  FIRST INP
2080@   * * * * *   SECOND INP ERROR HANDLING   * * * * *
2090 IF ERRNO' = 22 THEN GOTO 2140   @@@  PARITY  @@@
2100 IF ERRNO' = 23 THEN GOTO 2160   @@@  TIMEOUT @@@
2110 IF ERRNO' = 166 THEN GOTO 2180   @@@  FRAMING @@@
2120 ! "ERROR # ";ERRNO';" OCCURED AT LINE ";LINE'
2130 END
2140 @@@ PARITY ERROR @@@
2150    ! "PARITY ERROR"   END
2160 @@@ TIME OUT ERROR
2170    ! "TIMEOUT"  END
2180 @@@ FRAMING ERROR  @@@
2190 ! "FRAMING ERROR OCCURED AT LINE ";LINE'
2200 B' = NAK' GOTO 1920  @@@  SECOND INP

2210 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2220 @@@   READ 3 BYTE INTEGER   @@@
2230 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2240    IF A' <> POS_PAC' THEN
           ! "POSITION PAC INDICATOR NOT RECEIVED'  END
2250    INP:RS232IN' LSB'
2260    INP:RS232IN' MB'
2270    INP:RS232IN' MSB'
2280    ACT_POS = LSB' + MB'*256.0 + MSB'*65536.0
2290    IF ACT_POS > 8388607 THEN
           ACT_POS = ACT_POS - 16777216
2300    RETURN
```

```
2309@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2310 DECLARE (BIGI)  @ SEND_3_BYTE_INTEGER
2311@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2315     IF (BIGI < - 8388608) OR (BIGI > 8388607) THEN
           GOTO 450
2320     IF BIGI < 0 THEN
           BIGI = 16777216.0-ABS(BIGI) ! "NEG # ==> ";BIGI
2325     MSB' = 0
2330     IF BIGI >= 65536.0 THEN
           MSB' = INT(BIGI/65536.0)
           BIGI = BIGI - (MSB'*65536)
2335     MB' = 0
2340     IF BIGI >= 256 THEN MB' = INT(BIGI/256.0)
           LSB' = BIGI - (MB'*256)   GOTO 540
2345     LSB' = BIGI
2350     @  THE THREE BYTES HAVE BEEN DETERMINED
2355      @ "MSB/MB/LSB = ";MSB',MB',LSB'
2360     GOSUB 1570 (LSB')
2365     GOSUB 1570 (MB')
2370     GOSUB 1570 (MSB')
2380     return

2400@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2401@ Read_Strain_Gauge  SUBROUTINE
2402@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2410 DECLARE (VOLT)
2420 DIM GAIN(4), RANGE(4)
2430 GAIN[2] = 1;  GAIN[3] = 1;  GAIN[4] = 1;

2440 @*** change these variables for diff. setup ***
2450 gain[1] = 1;
2460  @ range:  -1 -> +-5v, 0 -> 0-10v, 1 -> 0-5v
2470 RANGE[1] = 1;
2480 RANGE[2] = 1;
2490 RANGE[3] = 1;
2500 RANGE[4] = 1;

2505 lbPERvolt = 50

2510 INPUT "ENTER CHANNEL OF LAB 40-A/D (1-4) " CHNL'
2520 GOSUB SendCOM1(CHNL')

2530 GOSUB ReadCOM1(loByte')
2540 GOSUB ReadCOM1(hiByte')

2550 VALUE = hiByte' * 256 + loByte'
2560 VOLT = VALUE/4096
2570 IF RANGE(CHNL') = -1 THEN VOLT = VOLT * 10 - 5
2580 IF RANGE(CHNL') = 0  THEN VOLT = VOLT * 10
2590 IF RANGE(CHNL') = 1  THEN VOLT = VOLT * 5

2600 VOLT = VOLT/GAIN(CHNL')
2610 RETURN
```