1990

Nuclear fuel rod leak detection system

Iztok Marjanovic
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Marjanovic, Iztok, "Nuclear fuel rod leak detection system" (1990). *Master's Theses*. 3205.
DOI: <https://doi.org/10.31979/etd.bbnd-nnp3>
https://scholarworks.sjsu.edu/etd_theses/3205

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1339631

Nuclear fuel rod leak detection system

Marjanovic, Iztok, M.S.

San Jose State University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

NUCLEAR FUEL ROD LEAK DETECTION SYSTEM

A Thesis

Presented to

The Faculty of the

Department of Electrical Engineering

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Iztok Marjanovic

June 1990

Approved for the Department of Electrical
Engineering

Peter Reischl

27 June 89

Dr. Peter Reischl

Lincoln Jones

5/25/89

Lincoln Jones

John Kervinen

5/30/89

John Kervinen, General Electric Company

Approved for the University

Serena H. Stanford

Abstract

Nuclear Fuel

Rod Leak Detection System

Iztok Marjanovic

Fuel rods in a nuclear reactor consist of uranium pellets housed in a protective zircaloy cladding. In normally operating fuel rods the cladding keeps fuel pellets separated from water in the reactor. If a rod fails, however, water from the reactor enters the cladding and collects at the bottom of the rod. A leak detection system has been developed which identifies these failed rods.

A pulsed ultrasonic wave is transmitted from a crystal through water into the fuel rod cladding. If a rod contains no water much of the sound wave is reflected back to the crystal from the gas cladding interface since gas inside an intact rod is a poor conductor of sound relative to solids such as the fuel rod cladding. In a rod which has leaked, however, the water inside the rod acts as a couplant allowing much of the sound wave to continue on rather than being reflected. The portion of the ultrasonic wave which is reflected is much smaller in magnitude for the failed rod. This difference in amplitude of reflected waves is employed to detect leaking fuel rods.

Contents

List of Tables	iii
List of Figures	iv
Chapter 1 Introduction	1-1
Nuclear Fuel Description	1-1
Reason for System Development	1-1
Principle of Operation	1-3
Hardware Description	1-5
Chapter 2 System Design	2-1
Hardware Design	2-1
Computer Interface Design	2-3
Motion Control Design	2-6
Data Acquisition System Design	2-12
Distinguishing Failed Rods	2-14
Software Design	2-14
Chapter 3 System Performance	3-1
Ability to Distinguish Wet and Dry Rods	3-1
Performance of the Control	3-3
Chapter 4 Discussion	4-1
System Performance	4-1
Future Applications	4-2

	ii
Appendix A Hardware Interface	A-1
Sony Gauges	A-3
USIP-12	A-3
Protection Circuit	A-4
124-A DC Motor Control	A-6
DT707 Screw Terminal Panel	A-7
DT2801 Board	A-8
TECMAR Board	A-11
Appendix B Listing of Computer Code	B-1
Appendix C Listing of Manufacturers	C-1
Bibliography	BIB-1

Tables

Table	Page
2-1. List of all include files and the functions which the routines in each perform.	2-16
A-1. DT2801 digital I/O port access on DT707 board.	A-8
A-2. Jumper positions for DT2801 board.	A-11
A-3. Jumper positions for TECMAR board.	A-13
A-4. Switch positions for TECMAR board.	A-13
A-5. Connections to TECMAR board.	A-14

Figures

Figure	Page
1-1. Schematic diagram of nuclear reactor fuel. A fuel bundle with its upper and lower tie plates and spacers is shown. One of the rods from the bundle is shown in detail. The rod is made from uranium fuel pellets housed in a protective zirconium alloy cladding.	1-2
1-2. Typical UT signal for sound and leaky rods. In the sound rod most of the UT signal is reflected back and forth between the crystal and inner diameter of the cladding. The water in the leaky rod conducts the UT signal into the rod. The reflections from the leaky rods inner diameter are 50 % lower in magnitude than those from the sound rod.	1-4
1-3. Probe movement platform and the lower portion of a fuel bundle.	1-6

- 1-4. System test setup of the system. The probe movement platform is shown in the tank along with a mock-up of a bundle. On the left is the flaw detector showing the reflections from one of the rods. In the center are the Sony Magnescale displacement gauges and on the right is the Compaq computer which controls the system. 1-7
- 2-1. Ultrasonic probe motion, showing the position of the probe at both ends of its Y-traverse. The dry and wet standards move with the probe as it moves along the Y-axis. 2-2
- 2-2. Ultrasonic probe design. The crystal is .125 inches in diameter and is mounted in a hole extending through the probe. 2-4
- 2-3. System interface diagram. The DT2801 and TECMAR boards read BCD data corresponding to probe position from the Sony gauges. An analog voltage from the USIP-12 flaw detector proportional to the magnitude of the reflected UT signal is read by the DT2801 boards A/D via the DT707 board. The DT2801 also controls the motors with its D/A port. 2-5

- 2-4. Block diagram of system. The compensation implemented in software is shown as a nonlinearity. The Sony gauges are shown as pure samplers. 2-7
- 2-5. Data relating motor input voltage and probe speed. The motor constant for motion along the Y-axis is .012 in/s/V and .050 in/s/V along the X-axis. 2-9
- 2-6. Models of motor transfer function nonlinearities shown in Figure 2-5. The stall voltages for both axis of motion are approximately 8 V. 2-10
- 2-7. Model of compensation implemented in software. V_m is the maximum D/A voltage allowed, V_s overcomes the stall voltage of the motors, and E_r determines the width of the dead zone for the bang-off-bang control. 2-11
- 2-8. Data Acquisition System (DAS) monitoring activity. Between rods no data is taken. In a .060 inch window over a rod, the probes velocity is slowed, and reflected UT signal data is taken. 2-13

- 2-9. Main software loop flow diagram. When no character has been entered by the user continuous tasks are performed as required by the current menu. If a character is entered it is also interpreted as required by the current menu. 2-17
- 2-10. Probe control flow diagram. After the probe is started in the right direction the main program loop is entered. When the probe is near its final position, control is performed in the control loop. 2-18
- 3-1. Magnitude of reflected UT signal from both wet and dry rods. The echo from the dry rod is much larger in magnitude than the echo from the wet rod. 3-2
- 3-2. Frequency spectrum of the reflected UT data shown in Figure 3-1. The spectrum shows a significant difference in signal power between wet and dry rods at frequencies below 60Hz. The power of the dry rods UT data is approximately 200 times that of the wet rod. 3-4

A-1.	System interface diagram.	A-2
A-2.	Protection circuit.	A-5
A-3.	DT2801 board jumper and switch locations.	A-10
A-4.	TECMAR board jumper locations.	A-15

Chapter 1

Introduction

An overview of the nuclear fuel rod leak detection system is given below. First a brief description of nuclear fuel is presented to give meaning to terms used later. The reason for system development is then given along with the principle behind its operation. Finally a brief description of the hardware is presented.

Nuclear Fuel Description

Fuel in a nuclear reactor consists of uranium pellets housed in a protective cladding made of a zirconium alloy. The cladding is capped at top and bottom by upper and lower end plugs, respectively. The cladding, pellets, and end plugs make up a fuel rod (Figure 1-1). The fuel rods are arranged in a matrix held together by upper and lower tie plates and intermediate spacers to maintain the space between rods. The fuel rods, spacers, and tie plates make up a fuel bundle. The fuel bundles themselves are arranged in a matrix within the reactor vessel when in use.

Reason for System Development

In a normally operating fuel rod the cladding keeps the interior of the rod isolated from the water surrounding

Figure 1-1. Schematic diagram of nuclear reactor fuel. A fuel bundle with its upper and lower tie plates and spacers is shown. One of the rods from the bundle is shown in detail. The rod is made of uranium fuel pellets housed in a protective zirconium alloy cladding.

it. If a rod fails, however, water from the reactor leaks into the rod and collects near the lower end plug. The leak detection system is used to identify rods which have failed in this manner, without removing them from the fuel bundle. Methods already in use can establish if a bundle has leaked but to determine which individual rod in the bundle is leaking the bundle must first be dismantled and each rod inspected individually. A savings of approximately five hours in inspection time can be achieved by using the in bundle leak detection system rather than inspecting each of the rods individually.

Principle of Operation

A pulsed ultrasonic (UT) wave is transmitted from a crystal, through a couplant (water), into the fuel cladding near the lower end plugs (Figure 1-2). When a rod contains no water, most of the UT wave is reflected back from the cladding gas interface. If a rod has leaked, however, the water in the rod acts as a couplant which allows much of the sound wave to continue on into the rod rather than being reflected. The reflected wave is thus much smaller in magnitude for the failed rod. The difference in amplitude of reflected waves is employed to determine if a fuel rod has leaked. See Walton et al. [1985] and Light et al. [1983] for a discussion of similar ultrasound techniques for finding leaky fuel rods. Ogawa, Lyons, and

SOUND FUEL ROD

LEAKY FUEL ROD

Figure 1-2. Typical UT signal for sound and leaky fuel rods. In the sound rod most of the UT signal is reflected back and forth between the crystal and inner diameter of the cladding. The water in the leaky rod conducts the UT signal into the rod. The reflections from the leaky rods inner diameter are 50 % lower in magnitude than those from the sound rod.

Soulis [1973] discuss other techniques of nuclear fuel inspection.

Hardware Description

The crystal used to send the UT signal is mounted on a probe which is in turn mounted on a moving platform (Figure 1-3). The platform moves the probe along two axis of motion so the crystal at the end of the probe traverses past each fuel rod in a bundle. Two geared DC motors drive the platform, one for each axis of motion. The entire platform sits twenty feet under water at the base of a fuel bundle in the fuel preparation station outside the reactor vessel itself.

The entire system is controlled by a Compaq computer [Compaq Portable, Compaq Computer Corp., Houston, Tex.] and associated cards which make the computer capable of controlling the motion of the probe as well as reading analog and digital data. The sound signal is transmitted and received by one crystal using an ultrasonic flaw detector [USIP-12, Krautkramer Branson, Lewistown, Pa.]. Probe position is measured using Sony Magnescales [LY201A, Sony Magnescales Inc., San Carlos, Ca.]. All electronics are located above water and connected to the probe movement platform via cables.

Figure 1-4 below shows the lab test setup of the system. The tank in the background contains

Figure 1-3. Probe movement platform and the lower portion of a fuel bundle.

Figure 1-4. Lab test setup of the system. The probe movement platform is shown in the tank along with a mock-up of a bundle. On the left is the flaw detector showing the reflections from one of the rods. In the center are the Sony Magnescale displacement gauges and on the right is the Compaq computer which controls the system.

Chapter 2

System Design

Several aspects of the leak detection system are described below. A general overview of the hardware design is presented first, with a detailed description given in Appendix A. The method used to control motion of the ultrasonic probe is then described followed by a description of the data acquisition system. Finally a description of the overall design of the software is given.

Hardware Design

Figure 2-1 shows the motion of the probe through the bundle. Separate motors are used to drive the probe along the two axis of motion. The motors are geared and operate from 0 to 24 VDC. The configuration results in a maximum probe velocity of 1.0 in/sec along the X axis and 0.25 in/sec along Y. The angular motion of the motors is translated to linear motion of the probe via a pulley-cable arrangement (Figure 1-3). The pulleys allow slippage if the probe hits an obstruction so as to avoid damage to the probe or platform.

The platform is mounted at the base of a 13 foot long bundle (Figure 1-3), the top of which is itself

Figure 2-1. Ultrasonic probe motion, showing the position of the probe at both ends of its Y-axis traverse. The dry and wet standards move with the probe as it moves along the Y-axis.

approximately 6 feet under water. A bundle weighs approximately 800 lbs and sits on top of a bearing. The bearing allows the bundle to be rotated so it can be closely aligned with the platform.

Figure 2-2 on the following page shows the ultrasonic probe. The probe must traverse very tight areas (on the order of .050 inches) and be able to bend around small irregularities in the bundle while keeping the UT crystal at a very consistent distance from the rods as it passes over them. Differences in crystal to rod distances as small as .010 inches can cause a sound rod to appear failed. The probe in Figure 2-2 is made of .035 inch stainless steel which gives it enough flexibility to follow the contours of the bundles and to bend around any small irregularities.

Computer Interface Design

A block diagram of the system component interfaces is shown in Figure 2-3 below. X and Y positions are read using Sony Magnescales. The Magnescales are made of ferromagnetic material on which a magnetic pattern of specified wavelength is recorded. A magnetic head is used to detect the magnetic pattern and transform it to an electrical signal. This electrical signal is interpreted by the Sony gauges which produce a binary coded decimal (BCD) output corresponding to the position of the probe.

Figure 2-2. Ultrasonic probe design. The crystal is .125 inches in diameter and is mounted in a hole extending through the probe.

Figure 2-3. System interface diagram. The DT2801 and TECMAR boards read BCD data corresponding to probe position from the Sony gauges. An analog voltage from the USIP-12 flaw detector proportional to the magnitude of the reflected UT signal is read by the DT2801 boards A/D via the DT707 board. The DT2801 also controls the motors with its D/A port.

The BCD outputs are read by the digital I/O ports on the DT2801 [DT2801, Data Translation, Marlborough, Ma.] and TECMAR [Lab Tender, Scientific Solutions, Solon, Ohio] boards in the Compaq computer. Appendix A contains more detailed information regarding the DT2801 and TECMAR boards connection as well as other components of the system.

The motors are driven from the D/A ports on the DT2801 board. Signals from the computer are amplified using a motor controller [124-A, Powr-Ups, Bohemia, N. Y.] and then fed on to a protection circuit which protects the system during boot up and before the program is run. The protection circuit also directs power between X and Y motors using relays so only one amplifier is needed to drive both motors (Appendix A).

The actual UT signal is generated and processed by the ultrasonic flaw detector. The flaw detector generates a 0 - 10 VDC signal proportional to the magnitude of the reflected wave from the fuel rod. This is then read using the A/D port on the DT2801 board. For more details regarding the computer interface see Appendix A.

Motion Control Design

A simplified block diagram of the system is shown in Figure 2-4. The plant gain constant is found empirically by measuring the speed of the probes motion at various voltages from the D/A converter. Plots of the data are

Figure 2-4. Block diagram of system. The compensation implemented in software is shown as a nonlinearity. The Sony gauges are shown as pure samplers.

shown in Figure 2-5. From the data, models for the plant nonlinearities are constructed as in Figure 2-6.

Figures 2-5 and 2-6 show that significant static friction must be overcome when the probe is moved from rest. This results in a large stall voltage (or dead zone) in the plant transfer function. When the probe breaks free from its rest position its velocity makes a step increase since static friction is greater than kinetic friction (Figures 2-5 and 2-6).

The compensation used is a combination of proportional and bang-off-bang controls as shown in Figure 2-7. When the probe is far enough from its desired final location that $e(t)K_c = (r - x)K_c > V_m$ the output is held at its maximum value of V_m . When the probe comes within a distance of $e(t) = V_m/K_c$ the control becomes proportional with a gain of K_c . Finally, very near the final destination where the output $e(t)K_c < V_s$, the control is bang-off-bang with a dead zone controlled by E_r .

The gain, K_c , the limiting voltage, V_m , the dead zone, E_r , and the stall voltage compensation, V_s , are all software configurable for both axis of motion. K_c is a proportional gain. V_m is the maximum voltage allowed by the D/A converter and amplifier combination. At the motor V_m is 25 V. The parameter V_s eliminates the dead zone due to the motors stall voltage and must be made at least as

Figure 2-5. Data relating motor input voltage and probe speed. The motor constant for motion along the Y-axis is 0.012 in/s/V and 0.050 in/s/V along the X-axis.

Figure 2-6. Models of the motor transfer function nonlinearities shown in Figure 2-5. The stall voltages for both axis of motion are approximately 8 V.

Compensation

Position Error (in)

Figure 2-7. Model of compensation implemented in software. V_m is the maximum D/A voltage allowed, V_s overcomes the stall voltage of the motors, and E_r determines the width of the dead zone for the bang-off-bang control.

large as the motors stall voltage else certain output voltages will result in no probe motion. E_r determines the dead zone for the bang-off-bang control.

Care must be taken when setting E_r because the system has a tendency to oscillate about its final position if E_r is made too small since the system is essentially bang-bang when E_r is very small. To pick the actual values for E_r and K_c an iterative process is used. First a small value for E_r is arbitrarily chosen. The gain is then varied to try to eliminate oscillation. If such a gain exists, E_r is lowered again and other gains are tried until finally E_r is so small that it is not possible to stabilize the output. In this way the minimum E_r is found along with a K_c which does not cause the system to oscillate.

For X motion the minimum E_r is found to be 0.003 inches with a corresponding K_c of 225 V/in. For Y motion minimum E_r is 0.001 inches with a K_c of 1800 V/in. The Sony gauges as they are implemented, can resolve probe position down to 0.001 inches. E_r in Y is therefore at its minimum possible value given the resolution of the system.

Data Acquisition System Design

Figure 2-8 shows the data acquisition process. Between rods no UT data is taken, only the position of the probe is monitored and the keyboard checked for input. When the crystal comes near a rod the probe is slowed and

Figure 2-8. Data Aquisition System (DAS) monitoring activity. Between rods no data is taken. In a .060 inch window over a rod, the probes velocity is slowed, and reflected UT signal data is taken.

data from the output of the flaw detector is taken. This data represents the magnitude of the UT signal reflected from a rod. The data acquisition rate is approximately 500Hz. As the data is taken it is stored to memory. The data is not processed until after an entire run is completed.

Distinguishing Failed Rods

A fast fourier transform (FFT) of the data is done at the end of a run, and the power at the lower end of the spectrum (below 60Hz) is used to distinguish between wet and dry rods. The power at these frequencies is calculated and compared to threshold values set during calibration. If a rod has a power which is greater than an upper threshold it is labeled intact, if it is below a lower threshold it has leaked, and if it lies between these two threshold it is uncertain if it has leaked or not and it is labeled as uncertain.

Software Design

The system software is written in Turbo Pascal [Turbo Pascal Version 3.0, Borland International, Scotts Valley, Ca.] and included in Appendix B. All procedures and functions are organized into several include files according to their function. Each include file contains all the declarations needed by other routines to use the procedures in the file. If any global variables are

accessed they are listed at the beginning of the include file. Table 2-1 contains a list of all include files along with their function.

A flow diagram of the main software loop is shown in Figure 2-9. After initializing variables, the main program tests the Tecmar and DT2801 boards as well as the Sony gauges to see if they are responding to commands.

Following the hardware check the main program loop is entered (Figure 2-9). At the top of the loop a keyboard buffer is checked for keystrokes. If the operator enters a character, the character is interpreted according to the menu the operator is presently in since each menu has its own commands. A separate keyboard interpreter is therefore needed for each menu. The keyboard interpreters are in the include file KBD.INC.

If no character is entered the continuous tasks associated with the present menu are executed. Continuous tasks may include monitoring the probe's position as it passes through the bundle, updating a clock on the screen, or updating various other screen parameters. The procedures which perform the continuously executed tasks are located in the EXEC.INC include file.

The probe control flow diagram is shown in Figure 2-10 below. To get the probe to some final position it is first started in the correct direction and the main program loop

DECLARE.INC	Contains all global constant, type, and variable declarations.
UTILITY.INC	Contains general purpose utility routines.
TECMAR.INC	Contains all routines needed to access and use the Scientific Solutions TECMAR board.
DT2801.INC	Contains all routines needed to access and use the Data Translation DT2801 board.
SONY.INC	Contains all routines needed to read the Sony Magnescales.
PROBE.INC	Contains routines which control the motion of the probe.
SCRPRB.INC	Contains routines which give a graphic representation of the probe moving through the bundle on the computer screen.
DATA.INC	Contains routines which take reflected UT wave data from the flaw detector.
MENU.INC	Contains routines which paint the various menus on the screen.
KBD.INC	Contains the keyboard interpreters associated with each menu.
EXEC.INC	Contains the routines which perform the continuously executed tasks associated with each menu.
INIT.INC	Contains routines which initialize variables.

Table 2-1. List of all include files and the function which the routines in each perform.

Figure 2-9. Main program loop. When no character has been entered by the user continuous tasks are performed as required by the current menu. If a character is entered it is also interpreted as required by the current menu.

Figure 2-10. Probe control flow diagram. After the probe is started in the right direction the main program loop is entered. When the probe is near its final position, control is performed in the control loop.

(Figure 2-9) is entered. In the main program loop the keyboard is continuously being polled for input so the user can intervene to stop the probe in case of problems. One of the continuous tasks performed in the main program loop is monitoring the position of the probe, and when it gets near its final position calling a routine which will actually stop the probe.

The new routine begins by initializing variables and starting two timers. One timer makes sure the sample period is long enough and if it is not a flag is set. The other timer makes sure that it is not taking the probe too long to stop because of some type of problem. The most likely problems are that the probe is stuck and cannot reach its final destination or the gain and dead zone are set incorrectly so the system is oscillating about its final position but not stopping inside the dead zone. If it is taking too long to stop the probe, the control loop is exited. The actual control is done at the top of the control loop where the probe position is read and a control voltage calculated and output to the motors. When the probe remains inside the dead zone for three consecutive sample periods it is assumed the probe has actually stopped and the control loop is exited.

Chapter 3

System Performance

Two basic performance aspects of the In Bundle Nuclear Fuel Rod Leak Detection System are discussed below. The ability of the system to distinguish between wet and dry rods is considered, and a brief discussion of how precisely the control algorithm is able to control the probe is given.

Ability to Distinguish Wet and Dry Rods

Actual data from a data acquisition run for both sound and failed rods is shown in Figure 3-1. The data shows the magnitudes of an ultra sound wave at different crystal positions across both dry and wet fuel rods. The dry rod shows a significant increase in reflected signal magnitude as the crystal nears the center of the rod. The wet rod on the other hand shows no real increase in reflected wave magnitude. The displacement axis in Figure 3-1 is marked in $10E-3$ inch increments which are an approximation. Displacement information is not taken with each data point since this would significantly slow down the sample rate. The displacement just before the first data point and displacement right after the last data point are recorded,

Echo Magnitude Data for Dry and Wet Rods

Figure 3-1. Magnitude of reflected UT signal (echo) from both wet and dry rods. The echo from the dry rod is much larger in magnitude than the echo from the wet rod.

however, and the difference used to estimate the distance between data points.

Figure 3-2 shows the Fourier spectrum of the data in Figure 3-1. This spectrum is used to decide if a rod is sound or has leaked. The spectrum shows maximum power in the data to be at the lower frequencies, below approximately 60Hz. The power at frequencies below 60Hz for the dry rod is approximately 200 times that of the wet rod at these same frequencies. The data shows a wide difference between the powers of the reflected UT signal from wet and dry rods exists ensuring that failed rods will be distinguished if they are present.

Performance of the Control

For X motion the smallest stopping error distance (dead zone) which does not produce oscillation is 0.003 inches. For the present application this is better accuracy than is actually needed. The magnitude data in Figure 3-1 shows that the reflected signal is strong over a window of at least 0.010 inches on either side of the center of a rod, making the 0.003 inch error small by comparison.

In the Y direction the stopping accuracy is 0.001 inches. Accuracy in the Y direction is needed to stop the probe in line with a row of rods in the bundle. Since the probe is made to follow the contours of the rods and is

Figure 3-2. Frequency spectrum of the reflected UT data shown in Figure 3-1. The spectrum shows a significant difference in signal power between wet and dry rods at frequencies below 60 Hz. The power of the dry rods UT data is approximately 200 times that of the wet rod.

made flexible it is able to accommodate errors larger than 0.001 inches and still function in a satisfactory manner.

Chapter 4

Discussion

Several issues regarding the performance of the leak detection system are discussed below. The testing of rods as well as probe design are considered. Possible modification of the system for future applications is also discussed.

System Performance

The In Bundle Leak Detection System depends on water in a failed rod being at the base of the rod. From experience in the past with leaky fuel rods it has been found that this is indeed a very reliable phenomena. Corrosion on the exterior of the rods is significant higher up the bundle which can also make failure detection high above the lower tie plate difficult to impossible. Finally, at distance well above the lower tie plate the rods can have significant bowing. The control system has no feedback as to the position of the rods themselves; it only knows where the crystal is and assumes the rod positions are fairly well known. Near the bottom of the bundle this assumption is very good since the rods are held rigid at the lower tie plate. Higher up from the lower tie

this assumption is likely to be invalid. Therefore the only reliable point on the bundle where data can be taken is at the base of the bundle as is being done.

The distance of the probe from the rods as it passes over them is also critical in getting accurate readings. If the probe does not keep the crystal at consistent distances from the fuel rods, a failed rod may appear to pass or a sound rod fail. The design of the probe must ensure that the crystal probe distance remains constant. A first cut at the design of the probe used 0.060 inch thick stainless steel for the probe material, with an additional clearance of 0.060" added at the end of the probe near the crystal to separate the crystal from the rods. Not only was this probe not able to follow the contours of the rods as it passed through the bundle it actually could not pass through the bundle without getting stuck. The probe shown in Figure 2-2 is much better able to pass over each of the rods and keep the crystal at a consistent distance from the rods.

Future Applications

The probe movement platform is a general means of moving a small object inside a bundle. At present the system is designed so as to minimize the need for an accurate control. This has allowed the use of very sturdy parts which have been used before in the harsh environments

around a fuel bundle. In future applications more precise control of the probe may be necessary which will require several modifications to the present system.

First the effects of stiction can be minimized if linear bearings are used instead of bushings. The types of bushings used have previously been employed on equipment used under water in high radiation environments and have performed well. The bushings are also much easier to clean from contaminants than linear bearings. They are, however, a considerable source of stiction and if a smooth accurate control is needed they will likely have to be replaced.

The resolution of the displacement sensors (Sony gauges) can also be made much better. At present displacement can only be resolved down to 0.001 inches. This is due to lack of digital I/O space on the DT2801 and TECMAR boards in the present configuration. The Sony gauges themselves are capable of considerably better resolution. This could easily be changed by using a different digital I/O cards in the computer. The present configuration of digital I/O was chosen so as to be compatible with other systems already in use but may need to be modified for future applications.

Appendix A

Hardware Interface

The interface between hardware components of the system is described below. Descriptions of each component and its function within the system are included. All the necessary pin connections and switch settings are also listed in tables. A block diagram of the systems hardware interface is shown in Figure A-1. Figure A-1 is intended as a reference in the discussion to follow.

Figure A-1. System interface diagram.

Sony Gauges

Sony gauges are used to measure the displacement of the probe from its home position. There are two sensors on the probe fixture which feed displacement information to the Sony gauges located above the pool. The Sony gauges then convert this information into a numeric format which can be read by the operator from a digital display and by a computer through a BCD output port.

X and Y displacements must be monitored. The BCD output of the Sony gauge measuring X displacement is read using the digital I/O ports on the Tecmar board, while Y displacement is read by the Data Translation DT2801 board. All connections to the Sony gauges are described in the sections covering the Data Translation DT707 terminal board and the Tecmar board.

USIP 12

The USIP 12 is used to measure the intensity of an ultra sound wave which has bounced off a fuel rod and returned back to the USIP 12 probe. The intensity of the reflected wave is output from the USIP 12 in the form of an analog voltage which is read by an analog to digital converter on the DT2801 board. The output signal ranges from 0 volts to 10 volts and corresponds to the height (in number of divisions seen on the USIP 12's CRT screen) of the reflected signal. No voltage implies no reflected

signal is seen on the CRT screen while 10 volts means the reflected signal reaches the top of the screen which is ten divisions high.

The only output taken from the USIP 12 is the reflected signal magnitude. A 50 pin D-type subminiature connector is used to connect the USIP 12 to the DT707 screw terminal panel (see description below) through a coaxial cable. This transfers the reflected signal magnitude from the USIP 12 to the DT2801's analog to digital converter through the DT707 board.

Protection Circuit

The protection circuit (see Figure A-2 below) is used to direct voltages from the 124-A to either the X or Y motors as well as to protect the system when the Compaq is powered up.

When power is applied to the Compaq, the output voltages from the DT2801 boards D/A converters are set to the lower limits of their range. Since the DT2801 is configured for D/A operation in the range +/-5 volts, this means the output of the D/A's is set to -5 volts on bootup. If this voltage were allowed to be fed on to the 124-A motor controller (see description below) the motors would be turned on every time the compaq is booted up from a power off condition. This could potentially cause damage to the fixture itself, to the motors, and/or the motor

Figure A-2. Protection circuit.

controller since they may be trying to move the probe after it has hit something.

To remedy the problem relay 1 (Figure A-2) is installed between the output of D/A converter 0 and the input to the 124-A motor controller. Relay 1's coil is connected, via diodes, to D/A 1 so that it switches the voltage from D/A 0 to the normally open position whenever a negative voltage is present on D/A 1. Since nothing is connected to the normally open position of relay 1 a safe condition for the system is produced on power up when the D/A voltages are both negative. After the D/A voltages are initialized by the program, 0 volts is present on D/A 1 which puts relay 1 in the normally closed position thus connecting D/A 0 with the 124-A motor controller.

The output of the 124-A is directed to either motor X or motor Y with relay 2. The coil of relay 2 is connected to D/A 1 through a diode. The diode allows only positive voltages from the D/A to move relay 2 from its normally closed position (motor X connected to 124-A output) into its normally open position (motor Y connected to 124-A output).

124-A DC Motor Control

The 124-A isolated analog input 24V DC motor control is used to drive the two motors which move the probe. Command voltages of +/-5V from the DT2801 boards D/A

produce a +/- 24VDC response from the 124-A which drives the motors. Relays are used to direct the output of 124-A to either the X or Y motors. The stall current of the motors should not exceed 2.5 amperes.

DT707 Screw Terminal Panel

The DT707 screw terminal panel provides a connection, via screw terminals, for all analog and digital signals to be processed by the DT2801 board. The DT707 connects to the DT2801 through a ribbon cable which plugs directly into the DT2801 board inside the Compaq.

The digital I/O port connections on the DT707 are used to read the BCD output of the Y sony gauge. A ribbon cable connects the sony gauge with a D-type subminiature connector which in turn is connected to the DT707 board. Detailed connections between the Y Sony gauge and the DT707 are shown in Table A-1 below.

DT707 (DT2801 function)	D-connector (pin)	Y-Sony (pin/function)
port 0, bit 0	17	20 / 2nd digit 1
port 0, bit 1	4	22 / 2nd digit 2
port 0, bit 2	16	24 / 2nd digit 4
port 0, bit 3	3	26 / 2nd digit 8
port 0, bit 4	15	11 / 3rd digit 1
port 0, bit 5	2	13 / 3rd digit 2
port 0, bit 6	14	15 / 3rd digit 4
port 0, bit 7	1	17 / 3rd digit 8
port 1, bit 0	29	12 / 4th digit 1
port 1, bit 1	12	14 / 4th digit 2
port 1, bit 2	30	16 / 4th digit 4
port 1, bit 3	11	18 / 4th digit 8
port 1, bit 4	31	3 / 5th digit 1
port 1, bit 5	10	5 / 5th digit 2
port 1, bit 6	32	7 / 5th digit 4
port 1, bit 7	9	9 / 5th digit 8
DIG GND	20	1 / GND

Table A-1. DT2801 digital I/O port access on DT707 board.

DT2801 Board

The Data Translation DT2801 board is used to read the BCD output from the Y Sony gauge as well as outputting analog voltages to the 124-A motor controller and the relays in the protection circuit (see description above). The DT2801 is also used to read the analog output from the USIP 12.

Both digital I/O ports are used to read the output from the Sony. D/A port 0 is used as the input to the 124-A and D/A port 1 is used to drive the relays in the protection circuit. A/D converter 0 is used to read the output from the USIP 12.

The Data Translation DT2801 board is accessed from the

exterior of the Compaq through one male IDC header. A single ribbon cable then goes to the Data Translation DT707 screw terminal panel. All DT2801 functions are accessed from this panel.

Jumpers are installed on the DT2801 board by adding a wire or inserting a jumper plug between wire wrap posts. Jumper locations are designated in Table A-2 below by an upper case "W" followed by a number. These designations are also marked on the board itself. Refer to Table A-2 for the positions of jumpers and connectors. Figure A-3 gives the jumper locations on the board itself.

Jumpers determining configurations for applications not used are set to their default value. These jumpers are marked "not used." Generally the default value is factory set.

Figure A-3. DT2801 board jumper and switch locations.

Install	Remove	Purpose
W4	W5	For bipolar A/D conversion.
W3,W7	W6,W8	For differential A/D inputs.
W9,W12	W10,W11	For +/- 5 volt D/A conversion on DAC 0.
W13,W16	W14,W15	For +/- 5 volt D/A conversion on DAC 1.
W1,W19,W23	W2,W17,W18,W20, W21,W22,W24	Select base address of board to be 02EC Hex.
W26,W27	W25,W28,W29,W30	Select channel 1 as DMA channel (not used).

Table A-2. Jumper positions for DT2801 board.

Tecmar Board

The Tecmar board is used to read the BCD output of the X Sony gauge as well as reading the ready signals from both X and Y Sony gauges and issuing the latch signals to both gauges.

To initiate a read of one of the Sony gauges a latch signal is output from the upper half of port C on the Tecmar DIO port. This means that the upper half of port C must be software configured for output. After a read is initiated the lower part of port C is used to read the ready signal from the Sony gauge to be read. Ports A and B are used to read the actual BCD data from the X Sony gauge.

The Tecmar board is accessed through a D-type subminiature connector on the outside of the Compaq. A

ribbon cable connects the D-type access connector and the X-Sony gauge. Two wires are spliced out of this ribbon cable and are connected to the RDY and LATCH lines of the Y-Sony gauge.

Tables A-3, A-4, and A-5 give positions of jumpers, switches, and connectors while Figure A-4 gives the locations of the jumpers and switches on the Tecmar board itself. In the tables, jumper locations are designated by the letters "JPR" followed by a number, switch locations are designated by the letters "SW" followed by a number and connector positions are designated by the letter "P" followed by a number. Switches determininig configurations for applications not used are set to a default value. These switches are marked "not used." Generally the default values are factory set.

- JPR2: jumpers in location "A" and "C" - configures bits 1 and 5 of parallel port C for non-inverted output.
- JPR3: jumper in location "A" - selects timer output OUT2 as a possible interrupt source (not used).
- JPR4: jumper placed across its pins - provides an external 'start conversion' signal (not used).

Table A-3. Jumper positions for Tecmar board.

- SW1,SW3: configured as shown in figure yyy - sets the base address of the board to 0330 Hex.
- SW2: configured as shown in figure yyy - selects the differentia mode for the A/D converter (not used).

Table A-4. Switch positions for Tecmar board.

Tecmar board (function)	D-con. (pin)	X-Sony (pin/function)	Y-Sony (pin/function)
port A, bit 0	17	20 / 2nd digit 1	
port A, bit 1	4	22 / 2nd digit 2	
port A, bit 2	16	24 / 2nd digit 4	
port A, bit 3	3	26 / 2nd digit 8	
port A, bit 4	15	11 / 3rd digit 1	
port A, bit 5	2	13 / 3rd digit 2	
port A, bit 6	14	15 / 3rd digit 4	
port A, bit 7	1	17 / 3rd digit 8	
port B, bit 0	29	12 / 4th digit 1	
port B, bit 1	12	14 / 4th digit 2	
port B, bit 2	30	16 / 4th digit 4	
port B, bit 3	11	18 / 4th digit 8	
port B, bit 4	31	3 / 5th digit 1	
port B, bit 5	10	5 / 5th digit 2	
port B, bit 6	32	7 / 5th digit 4	
port B, bit 7	9	9 / 5th digit 8	
port C, bit 0	33	28 / X-RDY	28 / Y-RDY
port C, bit 1	8		
port C, bit 2	34	NC	
port C, bit 3	7	NC	
port C, bit 4	35	2 / X-LATCH	2 / Y-LATCH
port C, bit 5	6		
port C, bit 6	36	NC	
port C, bit 7	5	NC	
GND	20	1 / GND	
GND	22	NC	
GND	23	NC	
GND	24	NC	
GND	25	NC	
GND	26	NC	
GND	27	NC	
GND	28	NC	
NC	18	NC	
NC	19	NC	
NC	37	NC	

Table A-5. Connections to Tecmar board.

Figure A-4. TECMAR board jumper locations.

Appendix B

Listing of Computer Code

A listing of the computer code used by the In Bundle Fuel Rod Leak Detection System is given below. Discussion of the code is done in Chapter 2.

PROGRAM LEAK;

{%C-}

{ Constant, type, and variable declarations. }
{%I DECLARE.INC}

{ Utility routines. }
{%I UTILITY.INC}

{ Tecmar board related routines. }
{%I TECMAR.INC}

{ DT2801 board related routines. }
{%i DT2801.INC}

{ SONY gauge related routines. }
{%I SONY.INC}

{ Probe movement routines. }
{%i PROBE.INC}

{ Screen probe routines. }
{%I SCRPRB.INC}

{ Data acquisition routines. }
{%I DATA.INC}

{ Menu's. }
{%I MENU.INC}

{ Keyboard command interpreters. }
{%I KBD.INC}

{ Execution routines. }
{%I EXEC.INC}

{ Variable initialization routines. }
{%I INIT.INC}

{

```

}
{*****}
{                                     }
{                               Main program.                               }
{*****}

BEGIN
  { Initialize variables. }
  INIT_FROM_DISK(ERR_FLG); { Initialize the variables from disk. }
  INITIALIZE;             { Initialize variables not on disk. }
  STARTUP_MENU;           { Put up STARTUP menu on screen. }

  { Configure and test hardware. }
  IF NOT ERR_FLG THEN
  BEGIN
    TCHR_CONFIG;          { Configure TCHMAR board. }

    { Configure and load timers on TCHMAR board. }
    TIMER_CONFIG;
    LOAD_TIMER(DISP_TIMER, DISP_PERIOD);
    LOAD_TIMER(STCK_TIMER, STCK_PERIOD);

    { Test TCHMAR and DT2801 boards and SONY gauges for major faults. If a }
    { problem is found don't let the program continue until it is resolved }
    { or set a hardware fault flag (HRDW_FAULT) which will not allow the }
    { user to enter any part of the program where the hardware is needed. }
    { If a hardware fault is found don't test anymore of the hardware. }
    TERMINATE := FALSE;    { Not ready to quit program. }
    HRDW_FAULT := FALSE;   { No hardware fault found yet. }

    { Test TCHMAR board. }
    WRITELN('Testing Tecmar board. ');
    WHILE (NOT TCHMAR_OK) AND (NOT TERMINATE) AND (NOT HRDW_FAULT) DO
    BEGIN
      TEXTCOLOR(WHITE+BLINK);
      WRITELN
      ('PROBLEM WITH TCHMAR BOARD: CANNOT MOVE PROBE UNTIL PROBLEM IS RESOLVED. ');
      TEXTCOLOR(WHITE);
      CASE Q_R_I OF
        'Q' : TERMINATE := TRUE;
        'I' : HRDW_FAULT := TRUE;
      END; { case }
    END; { while }

    { Test DT2801 board. }
    WRITELN('Testing DT2801 board. ');
    WHILE (NOT TERMINATE) AND (NOT HRDW_FAULT) AND (NOT DT2801_OK) DO
    BEGIN
      TEXTCOLOR(WHITE+BLINK);
      WRITELN

```

```

('PROBLEM WITH DT2801 BOARD: CANNOT MOVE PROBE UNTIL PROBLEM IS RESOLVED.');
```

TEXTCOLOR(WHITE);
 CASE Q_R_I OF
 'Q' : TERMINATE := TRUE;
 'I' : HRDW_FAULT := TRUE;
 END; { case }
 END; { while }

{ Test X SONY gauge. }
 WRITELN('Testing X Sony gauge.');

WHILE (NOT TERMINATE) AND (NOT HRDW_FAULT) AND (NOT SONY_OK(X)) DO
 BEGIN
 TEXTCOLOR(WHITE+BLINK);
 WRITELN
 ('PROBLEM WITH X SONY GAUGE: CANNOT MOVE PROBE UNTIL PROBLEM IS RESOLVED.');

 TEXTCOLOR(WHITE);
 CASE Q_R_I OF
 'Q' : TERMINATE := TRUE;
 'I' : HRDW_FAULT := TRUE;
 END; { case }
 END; { while }

{ Test Y SONY gauge. }
 WRITELN('Testing Y Sony gauge.');

WHILE (NOT TERMINATE) AND (NOT HRDW_FAULT) AND (NOT SONY_OK(Y)) DO
 BEGIN
 TEXTCOLOR(WHITE+BLINK);
 WRITELN
 ('PROBLEM WITH Y SONY GAUGE: CANNOT MOVE PROBE UNTIL PROBLEM IS RESOLVED.');

 TEXTCOLOR(WHITE);
 CASE Q_R_I OF
 'Q' : TERMINATE := TRUE;
 'I' : HRDW_FAULT := TRUE;
 END; { case }
 END; { while }
 END; { if }

IF (NOT ERR_FLG) AND (NOT TERMINATE) THEN
 BEGIN
 WRITE('Enter plant name ("",PLANT,"" will be used if none is entered) : ');
 READLN(TPLANT); IF TPLANT <> '' THEN PLANT := TPLANT;
 MAIN_MENU;
 END
 ELSE TERMINATE := TRUE;

{ Put up the main menu on the screen. }
 WHILE NOT TERMINATE DO

{ Check the keyboard for input and if there is input interpret it }

```

{ according to which menu the program is presently in. If there is no }
{ input from the keyboard, see if the system timer has timed out. If }
{ the system timer has timed out perform the continuous tasks associated }
{ with the present menu. }
BEGIN
  KBD_INP(KBD_DATA,KBD_VALID);
  IF KBD_VALID THEN
    BEGIN
      MSG(MSG_13);      { Clear message window after each new character. }
      CASE MENU OF
        RUN : RUN_KBD_CMD(KBD_DATA);
        WAIT : WAIT_KBD_CMD(KBD_DATA);
        RESULT : RESULT_KBD_CMD(KBD_DATA);
        CAL : CAL_KBD_CMD(KBD_DATA);
        INIT : INIT_KBD_CMD(KBD_DATA);
        MNL : MNL_KBD_CMD(KBD_DATA);
      END; { case }
    END
  ELSE
    BEGIN
      GET_PROBE_POSITION;
      CASE MENU OF
        RUN : EXEC_RUN;
        CAL : EXEC_CAL;
        INIT : EXEC_INIT;
        WAIT : EXEC_WAIT;
        RESULT : EXEC_RESULT;
        MNL : EXEC_MNL;
      END; { case }
    END; { else }
  END; { while }
  WINDOW(1,1,80,25);
  CURSORON;
  CLRSCR;
END. { program }

```

```

{*****}
{          DECLARE.INC          }
{*****}
{
Description of this module:

    This module contains all the global variable, type, and constant
    declarations needed by the program.
}
CONST
{ Probe movement constants }
X = 0;           { probe is moving in X direction }
Y = 1;           { probe is moving in Y direction }
DONE = 3;        { finished with run }
NOTHING = 50;    { probe is not moving }

{ Messages for the user. }
TO_STOP_PRB = 'Hit any key to stop probe.';
PRB_AT_ROD = 'Probe is over rod.';
PRB_AT_ROW = 'Probe is at a row.';
PRB_AT_ZEROX = 'Probe is at X stop.';
PRB_AT_HOME = 'Probe is at home position.';
MSG_9 = 'Input is out of range, re-enter the correct value.';
MSG_12 = 'Probe is over rod.';
MSG_13 = '';
MSG_16 = 'Probe is at home position.';
MSG_19 = 'Probe is at row.';

PROG_NAME = 'In Bundle Leak Detection System';
VERSION = '0.02';

TYPE
PLANT_TYPE = STRING[16];
BUNDLE_TYPE = STRING[6];
FILE_NAME_TYPE = STRING[20];

VAR
MENU : BYTE;           { Present program menu. }
TERMINATE : BOOLEAN;   { End program flag. }
ERR_FLG : BOOLEAN;     { Flag implying that some type of error has occurred.}
HRDW_FAULT : BOOLEAN;  { Flag signifying that a hardware fault exists. }
X1,Y1,X2,Y2 : INTEGER; { Window size parameters. }

ROD : BYTE;            { Rod being examined. }
ROW : BYTE;            { Row being examined. }
KBD_DATA : BYTE;       { Data from keyboard. }
KBD_VALID : BOOLEAN;   { Flag indicating if the KBD_DATA is valid. }

BUNDLE : BUNDLE_TYPE;  { Bundle number of bundle presently being examined. }

```

PLANT : PLANT_TYPE; { Plant name. }
TPLANT : PLANT_TYPE; { Temporary plant name. }
DRIVE : INTEGER; { Current data drive. }
DEF_DRV : STRING[1]; { Default data drive. }
DEF_DIR : STRING[16]; { Default data directory. }
DEF_DRV_DIR : STRING[30]; { Default data path. }
MIN_DRY_POWER, { Power of echo reflected from dry standard. }
MAX_POWER_RATIO : REAL; { Ratio of wet standard power to dry standard power. }

DISP_PERIOD : INTEGER; { Period (in ms) between display parameter updates. }
DISP_TIMER : INTEGER; { Period (in ms) between display parameter updates. }


```

{*****}
{                                UTILITY.INC                                }
{*****}
{

```

Description of this module:

This module contains general (and a few not so general) utility routines.

Procedures in this module:

```

PROCEDURE KBD_INP(VAR DATA : BYTE; VAR FLAG : BOOLEAN);
PROCEDURE CLR_KBD_BUF;
PROCEDURE GET_INT_VEC(INT_NUM,VEC_SEG,VEC_OFF : INTEGER);
PROCEDURE SET_INT_VEC(INT_NUM,VEC_SEG,VEC_OFF : INTEGER);
PROCEDURE MSG (VAR MESSAGE: MSG_TYPE);
PROCEDURE MAKEBOX(X1,Y1,X2,Y2, BD :INTEGER);
PROCEDURE CURSOROFF;
PROCEDURE CURSORON;
PROCEDURE SAVE_SCR(VAR SAVESCR: SAVEBUFPTR);
PROCEDURE RECALL_SCR(VAR SAVESCR: SAVEBUFPTR);
PROCEDURE GET_TIME(VAR TIME_DATE : TIME_DATE_TYPE);
PROCEDURE DISKSTATUS( DRIVE : INTEGER; VAR TRACKS, TOTALTRACKS,
                     BYTES, SECTORS : INTEGER );
FUNCTION DISK_BYTES( DRIVE : INTEGER ) : REAL;
PROCEDURE SET_DTA(VAR DTA : DTA_TYPE);
PROCEDURE FIRST_DIR(MASK : MASK_TYPE; VAR ERROR : INTEGER);
PROCEDURE NEXT_DIR(VAR ERROR : INTEGER);
PROCEDURE DEL_FILE(MASK : MASK_TYPE; VAR ERROR : INTEGER);
PROCEDURE MAKE_BORDER;
PROCEDURE ILLEGAL_KEY;
PROCEDURE WAIT_FOR_KEY;
PROCEDURE ANY_KEY;
FUNCTION YES : BOOLEAN;
FUNCTION Q_R_I : CHAR;
PROCEDURE INIT_UTILITY;

```

Global variables accessed by procedures in this module :

```

X1,X2,Y1,Y2
}
{*****}
{                                Declarations.                                }
{*****}

```

CONST

```

{ Constants global to this module. }
MAX_MASK = 20;      { Maximum dir path mask length. }
SAVEBUFLN = 4000;   { Length (in bytes) of video screen memory map buffer. }

```

TYPE

```

{ TYPE declarations needed to use procedures in this module. }
SAVEBUFPTR = ^IMAGETYPE; { Pointer to videoseen memory map buffer. }
IMAGETYPE = ARRAY [1..SAVEBUFPTR] OF CHAR; { Video screen memory map buffer. }
TIME_DATE_TYPE = RECORD { Filled in by GET_TIME routine. }
    MONTH : STRING[2];
    DAY : STRING[2];
    YEAR : STRING[4];
    HOUR : STRING[2];
    MINUTE : STRING[2];
    SECOND : STRING[2];
END;
DTA_TYPE = ARRAY [ 1..43 ] OF BYTE; { Data transfer area. }
MASK_STRING_TYPE = STRING[MAX_MASK]; { Drive and directory mask. }

{ TYPE declarations needed by procedures in this module. }
MSG_TYPE = STRING [79];

VAR
{ Variables global to this module. }
VIDEOOFFSET : INTEGER;
DTA : DTA_TYPE;

{*****}

PROCEDURE KBD_INP(VAR DATA : BYTE; VAR FLAG : BOOLEAN);
{
    This procedure checks the keyboard for input and returns the value in DATA
    if present and returns TRUE in FLAG. If no data, FALSE is returned in FLAG
    and DATA is not changed. All lower case characters are capitalized.
}
TYPE
    REG_TYPE = RECORD
        AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : INTEGER;
    END;
VAR
    MS_DOS_PAR : REG_TYPE;

BEGIN
    FLAG := FALSE; { set flag to indicate no data ready from kbd }
    MS_DOS_PAR.AX := $600; { keyboard input DOS function call }
    MS_DOS_PAR.DX := $FF;
    MSDOS(MS_DOS_PAR); { call MS DOS function. }
    IF LO(MS_DOS_PAR.AX) <> 0 THEN
        BEGIN
            DATA := LO(MS_DOS_PAR.AX); { get character from AL }

            { If the keyboard input is a lower case character then convert it }
            { to upper case. }
            IF (DATA >= $61) AND (DATA <= $7A) THEN DATA := DATA - $20;
        END
    END

```

```

    FLAG := TRUE;
END;
END; { kbd_inp }

{*****}

PROCEDURE CLR_KBD_BUF;
{
  Clears the standard input buffer of any pretyped characters.
}
TYPE
  REG_TYPE = RECORD
    AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : INTEGER;
  END;
VAR
  MS_DOS_PAR : REG_TYPE;

BEGIN
  MS_DOS_PAR.AX := $0C06; { Clear keyboard buffer followed by a direct }
                        { console I/O DOS function calls. }
  MS_DOS_PAR.DX := $FF;   { Console input during console I/O. }
  MSDOS(MS_DOS_PAR);      { Call MS DOS function. }
END; { clr_kbd_buf }

{*****}

PROCEDURE GET_INT_VEC(INT_NUM,VEC_SEG,VEC_OFF : INTEGER);
{
  Gets address (VEC_SEG:VEC_OFF) of interrupt = INT_NUM.
}
TYPE
  REG_TYPE = RECORD
    AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : INTEGER;
  END;
VAR
  REGS : REG_TYPE;

BEGIN
  REGS.AX := $3500 + INT_NUM; { Get interrupt vector function call. }
  MSDOS(REGS);               { Make call to MSDOS routine. }
  VEC_SEG := REGS.ES;         { Vector segment. }
  VEC_OFF := REGS.BX;         { Vector offset. }
END;

{*****}

PROCEDURE SET_INT_VEC(INT_NUM,VEC_SEG,VEC_OFF : INTEGER);
{
  Sets interrupt vector for interrupt = INT_NUM to VEC_SEG:VEC_OFF.
}

```

```

}
TYPE
  REG_TYPE = RECORD
    AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : INTEGER;
  END;
VAR
  REGS : REG_TYPE;

BEGIN
  REGS.AX := $3500 + INT_NUM; { Get interrupt vector function call. }
  REGS.DS := VEC_SEG;        { Vector segment. }
  REGS.DX := VEC_OFF;        { Vector offset. }
  MSDOS(REGS);
END;

{*****}

PROCEDURE MSG (MESSAGE: MSG_TYPE);
{
  Procedure MSG writes the contents of the variable MESSAGE to line 25 on the
  screen. It leaves the window settings (assumed to be in X1,Y1,X2,Y2) and
  cursor position as it found them.
}
VAR
  CX,CY : INTEGER;

BEGIN
  CX := WHEREX; CY := WHEREY; { Store present cursor location. }
  WINDOW(1,24,80,25);        { Set up message window. }
  GOTOXY(1,2); CLREOL; WRITE(MESSAGE); { Clear line and write message. }
  WINDOW(X1,Y1,X2,Y2);        { Restore window settings. }
  GOTOXY(CX,CY);               { Restore cursor position. }
END; { msg }

{*****}

procedure MakeBox(x1,y1,x2,y2, BD :integer);
{
  Draws a box. To make a box on the screen, call the procedure
  MakeBox(x1,y1,x2,y2,BD); The x and y coordinates define the window placement
  and are the same as the Turbo Pascal Window coordinates. The border
  parameters (BD) are 0 = No border 1 = Single line border 2 = Double line
  border 3 = Double Top/Bottom Single sides
}
var
  I,
  TB,SID,TLC,TRC,BLC,BRC :integer;

begin

```

TEXTCOLOR(YELLOW);

```
Case BD of
0 : ;
1 : begin
    TB := 196;
    SID := 179;
    TLC := 218;
    TRC := 191;
    BLC := 192;
    BRC := 217;
end;
2 : begin
    TB := 205;
    SID := 186;
    TLC := 201;
    TRC := 187;
    BLC := 200;
    BRC := 188;
end;
3 : begin
    TB := 205;
    SID := 179;
    TLC := 213;
    TRC := 184;
    BLC := 212;
    BRC := 190;
end;
End; {Case}

IF BD > 0 then
begin
    { Top }
    gotoxy(X1,Y1);
    Write( chr(TLC) );
    For I:=2 to x2-x1 do
        Write( chr(TB));
    Write( chr(TRC) );

    { Sides }
    for I:=2 to y2-y1 do
    begin
        gotoxy(X1,Y1+I-1);
        write( chr(SID) );
        gotoxy(x2,Y1+I-1);
        write( chr(SID) );
    end;

    { Bottom }

    { Make Border characters }
    { No border option }
    { Single line border option }
    { Top Border }
    { Side Border }
    { Top Left Corner }
    { Top Right Corner }
    { Bottom Left Corner }
    { Bottom Right Corner }

    { Double line border option }
    { Top Border }
    { Side Border }
    { Top Left Corner }
    { Top Right Corner }
    { Bottom Left Corner }
    { Bottom Right Corner }

    { Double Top/Bottom with single sides }
    { Top Border }
    { Side Border }
    { Top Left Corner }
    { Top Right Corner }
    { Bottom Left Corner }
    { Bottom Right Corner }

    { User want a border? }

    { Window Origin }
    { Top Left Corner }
    { Top Bar }
    { Top Right Corner }

    { Left Side Bar }
    { Right Side Bar }
```

```

        if mem[$0040:$0049] = 7 then { 7 = monochrome }
            CX := $0B0C           { 3 = Color      }
        else
            CX := $0607;
        end;
        intr(16,regs)
    end; { CursorOn }

{*****}

PROCEDURE SAVE_SCR(VAR SAVESCR: SAVEBUFPTR);
{
    Moves the memory map of the present display screen to a dynamically
    allocated memory location (SAVESCR) of type SAVEBUFPTR. The memory location
    of the display memory map is assumed to be stored in the variable
    VIDEOOFFSET.
}
BEGIN
    NEW(SAVESCR);
    MOVE(MEM[VIDEOOFFSET:0000],SAVESCR,SAVEBUFLN);
END; { save_scr }

{*****}

PROCEDURE RECALL_SCR(VAR SAVESCR: SAVEBUFPTR);
{
    Moves a previously stored memory map of a display screen (pointed to by
    SAVESCR) to the display screen memory map. The memory location of the
    display memory map is assumed to be stored in the variable VIDEOOFFSET.
}
BEGIN
    MOVE(SAVESCR, MEM[VIDEOOFFSET:0000],SAVEBUFLN);
END; { recall_scr }

{*****}

PROCEDURE GET_TIME(VAR TIME_DATE : TIME_DATE_TYPE);
{
    This procedure returns the time and date in TIME_DATE as strings.
}
TYPE
    REG_TYPE = RECORD
        AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : INTEGER;
    END;
VAR
    MS_DOS_PAR : REG_TYPE;
    I, RESULT : INTEGER;

```

```

        gotoxy(x1,y2);          { Bottom Left Corner }
        write( chr(BLC) );
        for l:=2 to x2-x1 do    { Bottom Bar      }
            write( chr(TB) );
        gotoxy(x2,y2);
        write( chr(BRC) );      { Bottom Right Corner }
    end;
    TEXTCOLOR(WHITE);
end; { MakeBox }

```

```

{*****}

```

```

Procedure CursorOff;

```

```

{
    This procedure uses the DOS interrupt 16 to turn the cursor off.
}

```

```

type

```

```

    RegPack = record
        AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : integer
    end;

```

```

var

```

```

    Regs : RegPack;

```

```

begin

```

```

    with regs do

```

```

        begin

```

```

            AX := $0100;

```

```

            CX := $2000;          { set CH bit 5 hi to suppress cursor }

```

```

        end;

```

```

        intr(16,regs)

```

```

    end; { CursorOff }

```

```

{*****}

```

```

Procedure CursorOn;

```

```

{
    This procedure uses the DOS interrupt 16 to turn the cursor back on.
}

```

```

type

```

```

    RegPack = record
        AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : integer
    end;

```

```

var

```

```

    Regs : RegPack;

```

```

begin

```

```

    with regs do

```

```

        begin

```

```

            AX := $0100;

```

```

BEGIN
  { TURBO PASCAL version }
  MS_DOS_PAR.AX := $2A00; { date DOS function call      }
  MSDOS(MS_DOS_PAR);      { call MS DOS function      }
  STR(HI(MS_DOS_PAR.DX):2,TIME_DATE.MONTH);
  STR(LO(MS_DOS_PAR.DX):2,TIME_DATE.DAY);
  STR(MS_DOS_PAR.CX:4,TIME_DATE.YEAR);
  MS_DOS_PAR.AX := $2C00;
  MSDOS(MS_DOS_PAR);
  STR(HI(MS_DOS_PAR.CX):2,TIME_DATE.HOUR);
  STR(LO(MS_DOS_PAR.CX):2,TIME_DATE.MINUTE);
  STR(HI(MS_DOS_PAR.DX):2,TIME_DATE.SECOND);
  VAL(TIME_DATE.MONTH,1,RESULT);
  IF I<=9 THEN
    TIME_DATE.MONTH[1] := '0';
  VAL(TIME_DATE.DAY,I,RESULT);
  IF I<=9 THEN
    TIME_DATE.DAY[1] := '0';
  VAL(TIME_DATE.MINUTE,I,RESULT);
  IF I<=9 THEN
    TIME_DATE.MINUTE[1] := '0';
  VAL(TIME_DATE.SECOND,I,RESULT);
  IF I<=9 THEN
    TIME_DATE.SECOND[1] := '0';
END; { get_time }

{*****}

Procedure DiskStatus( Drive : integer; var Tracks, TotalTracks,
                     Bytes, Sectors : integer );
{
  Makes MSdos call to read status of Drive; returns data in Tracks,
  TotalTracks, Bytes, and Sectors.
}
TYPE
  REG_TYPE = RECORD
    AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : INTEGER;
  END;
VAR
  REGS : REG_TYPE;

begin
  Regs.AX := $3600;      { Get Disk free space }
  Regs.DX := Drive;      { Store Drive number }
  MSDos( Regs );         { Call MSdos to get disk info }
  Tracks := Regs.BX;      { Get number of Tracks Used }
  TotalTracks := Regs.DX; { - - - total Tracks }
  Bytes := Regs.CX;       { - - - Bytes per sector }
  Sectors := Regs.AX      { - - - Sectors per cluster }

```



```

END;    { DiskStatus }

{*****}

FUNCTION DISK_BYTES( DRIVE : INTEGER) : REAL;
{
  Returns the number of bytes left on DRIVE.
}
VAR
  Tracks,                { number of available Tracks }
  TotalTracks,           { number of total Tracks }
  Bytes,                 { number of Bytes in one sector }
  Sectors                : Integer;    { number of total Sectors }

begin
  DiskStatus(Drive,Tracks,TotalTracks,Bytes,Sectors);
  DISK_BYTES := (( Sectors * Bytes * 1.0 ) * Tracks );
END; { disk_bytes }

{*****}

PROCEDURE SET_DTA(VAR DTA : DTA_TYPE);
{
  Sets the disk transfer address to the address of DTA.
}
TYPE
  REGREC = RECORD
    AX, BX, CX, DX, BP, SI, DI, DS, ES, FLAGS : INTEGER;
  END;

VAR
  REGS : REGREC;

BEGIN
  Regs.AX := $1A00;    { Function used to set the DTA }
  Regs.DS := Seg(DTA); { store the parameter segment in DS }
  Regs.DX := Ofa(DTA); { " " " " offset in DX }
  MSDos(Regs);        { Set DTA location }
END;

{*****}

PROCEDURE FIRST_DIR(MASK_STRING : MASK_STRING_TYPE; VAR ERROR : INTEGER);
{
  Gets first directory entry which fits MASK. MASK contains the drive, path,
  and filename of the file to be found. MASK can contain global filename
  characters. Error returns error codes if any occur otherwise error is set
  to 0. The results of the search are put in the DTA which has to be defined
  using SET_DTA before this procedure is called. Refer to the IBM DOS
  technical reference for how the DTA is filled in. Subsequent entries are be

```

```

    found using NEXT_DIR procedure below.
}
TYPE
    REGREC = RECORD
        AX, BX, CX, DX, BP, SI, DI, DS, ES, FLAGS : INTEGER;
    END;
    MASK_TYPE = ARRAY [1..MAX_MASK] OF CHAR;

VAR
    REGS : REGREC;
    I : INTEGER;
    MASK : MASK_TYPE;

BEGIN
    FILLCHAR(MASK, SIZEOF(MASK), 0); { Initialize the mask }
    FOR I := 1 TO LENGTH(MASK_STRING) DO MASK[I] := MASK_STRING[I];
    Regs.AX := $4B00; { Get first directory entry }
    Regs.DS := Seg(Mask); { Point to the file Mask }
    Regs.DX := Off(Mask);
    Regs.CX := 22; { Store the option }
    MSDos(Regs); { Execute MSDos call }
    Error := Regs.AX and $FF; { Get Error return }
END;

{*****}

PROCEDURE NEXT_DIR(VAR ERROR : INTEGER);
{
    Returns directory entries after the first one is found using FIRST_DIR
    described above. The results of the search are put in the DTA which has to
    be defined using SET_DTA before this procedure is called. Refer to the IBM
    DOS technical reference for how the DTA is filled in. Error returns error
    codes if any occur otherwise error is set to 0.
}
TYPE
    REGREC = RECORD
        AX, BX, CX, DX, BP, SI, DI, DS, ES, FLAGS : INTEGER;
    END;

VAR
    REGS : REGREC;

BEGIN
    Regs.AX := $4F00; { Function used to get the next directory entry }
    Regs.CX := 22; { Set the file option }
    MSDos( Regs ); { Call MSDos }
    Error := Regs.AX and $FF; { get the Error return }
END;

{*****}

```

```

PROCEDURE DEL_FILE(MASK_STRING : MASK_STRING_TYPE; VAR ERROR : INTEGER);
{
  Del all files which fit MASK. MASK contains the drive, path, and filename
  of the file to be found. MASK can contain global filename characters.
  Error returns error codes if any occur otherwise error is set to 0. The
  results of the search are put in the DTA which has to be defined using
  SET_DTA before this procedure is called. Refer to the IBM DOS technical
  reference for how the DTA is filled in.
}
TYPE
  REGREC = RECORD
    AX, BX, CX, DX, BP, SI, DI, DS, ES, FLAGS : INTEGER;
  END;
  MASK_TYPE = ARRAY [ 1..MAX_MASK ] OF CHAR;

VAR
  REGS : REGREC;
  MASE : MASK_TYPE;
  I : INTEGER;

BEGIN
  FILLCHAR(MASK,SIZEOF(MASK),0); { Initialize the mask }
  FOR I := 1 TO LENGTH(MASK_STRING) DO MASK[I] := MASK_STRING[I];
  Regs.AX := $4100; { Delete file. }
  Regs.DS := Seg(Mask); { Point to the file Mask }
  Regs.DX := OfS(Mask);
  MSDos(Regs); { Execute MSDos call }
  Error := Regs.AX and $FF; { Get Error return }
END;

{*****}

PROCEDURE MAKE_BORDER(Y : INTEGER);
{
  Makes a border around the screen. Y holds the y coordinate of the line
  separating the command description area of the screen and the scratch pad
  area.
}
VAR I : INTEGER;

BEGIN
  WINDOW(1,1,80,25); CLRSCR;
  MAKEBOX(1,1,80,24,3);
  IF Y <> 0 THEN
    BEGIN
      GOTOXY(1,Y); { Go to beginning of present line. }
      TEXTCOLOR(YELLOW); { Change character color to yellow. }
      WRITE($195); { "-" }
    END
  END

```

```

        FOR I := 1 TO 78 DO WRITE(#196); { "-" }
        WRITE(#180); { "|" }
        TEXTCOLOR(WHITE); { Return text color to white. }
    END;
END; { make_border }

{*****}

PROCEDURE ILLEGAL_KEY;
{
    Returns a BEL and a message to the user telling him that he pressed an
    illegal key.
}
BEGIN
    MSG('Invalid character entered. '); WRITE('G');
END; { illegal_key }

{*****}

PROCEDURE WAIT_FOR_KEY;
{
    Wait for user to hit any key.
}
VAR
    KBD_DATA : BYTE;
    KBD_VALID : BOOLEAN;

BEGIN
    CLR_KBD_BUF;
    REPEAT KBD_INP(KBD_DATA, KBD_VALID); UNTIL KBD_VALID;
END; { any_key }

{*****}

PROCEDURE ANY_KEY;
{
    Ask user to hit any key to continue ie. exit the procedure ANY_KEY.
}
BEGIN
    MSG('Hit any key to continue. '); WRITE('G');
    WAIT_FOR_KEY;
    MSG(MSG_13);
END; { any_key }

{*****}

FUNCTION YES : BOOLEAN;
{
    Waits for a yes or no response from the user and is set to true if user

```

responds with a "Y" and FALSE if the response is "N", otherwise it beeps the user and waits for a valid response.

```

}
VAR
  KBD_DATA : BYTE;
  KBD_VALID : BOOLEAN;

BEGIN
  CLR_KBD_BUF;
  REPEAT
    { Wait for user to respond. }
    REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID = TRUE;

    CASE KBD_DATA OF
      $59: YES := TRUE;
      $4E: YES := FALSE;
      ELSE
        BEGIN
          KBD_VALID := FALSE;
          WRITE('G');
        END;
    END;
  UNTIL KBD_VALID;      { Wait for correct response. }
  MSG(MSG_13);          { Clear message window. }
END; { yes }

{*****}

FUNCTION Q_R_I : CHAR;
{
  Asks user if he wants to Quit, Retry, or Ignore and returns 'Q', 'R', or 'I'
  depending on the users response. If an illegal character is entered the
  user is "beeped" and a valid response is waited for.
}
VAR
  KBD_DATA : BYTE;
  KBD_VALID : BOOLEAN;

BEGIN
  CLR_KBD_BUF;
  MSG('Press "Q" to Quit, "R" to Retry, or "I" to Ignore. ');
  REPEAT
    { Wait for user to respond. }
    REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID = TRUE;

    CASE KBD_DATA OF
      $51,$1B: Q_R_I := 'Q'; { User wants to quit. }
      $52: Q_R_I := 'R';     { User wants to try again. }
      $49: Q_R_I := 'I';     { The user wants to ignore the problem. }
    
```

```

ELSE                                { Illegal key entered. }
BEGIN
    KBD_VALID := FALSE;
    WRITE('G');
END;
END;
UNTIL KBD_VALID;      { Wait for correct response. }
MSG(MSG_13);          { Clear message window after a valid response. }
END; { q_r_i }

{*****}

PROCEDURE INIT_UTILITY;
{
    Initializes UTILITY module.
}
BEGIN
    { Determine display type and set offset (VIDEOOFFSET) to memory where the }
    { memory map for the display is stored. }
    IF MEM[0000:$0449] = 7 THEN VIDEOOFFSET := $B000 { monochrome display }
    ELSE VIDEOOFFSET := $B800;                      { color graphics display }

    FillChar(DTA,SizeOf(DTA),0);      { Initialize the DTA buffer }
    SET_DTA(DTA);
END; { init_utility }

```

```

{*****}
{          TECHAR.INC          }
{*****}
{
Description of this module:

    This module contains general routines which deal with the TECHAR board.

Procedures in this module:

PROCEDURE TIMER_CONFIG;
PROCEDURE LOAD_TIMER(TIMER : INTEGER; TIME_VAL : INTEGER);
FUNCTION  TIME_OUT(TIMER : INTEGER) : BOOLEAN;
PROCEDURE TCMR_CONFIG;
PROCEDURE OUT_TCMR_DIO(DIO_PORT, PAR : INTEGER);
PROCEDURE IN_TCMR_DIO(DIO_PORT INTEGER; VAR PAR : INTEGER);
FUNCTION  TECHAR_OK : BOOLEAN;
PROCEDURE INIT_TCHAR;

Global variables used : NONE
}
{*****}
{          Declarations.          }
{*****}

CONST
{ Constants needed to use procedures in this module. }
TIMER_1 = $01;      { Timers }
TIMER_2 = $02;
TIMER_3 = $04;
TIMER_4 = $08;
TIMER_5 = $10;

TCMR_DIO_A = 0;      { Digital IO port A. }
TCMR_DIO_B = 1;      { Digital IO port B. }
TCMR_DIO_C = 2;      { Digital IO port C. }

{ Constants global to this module. }
BASE = $0330;        { base address of TECHAR board }
TMR_CMD_PORT = $0339; { BASE + 9;}
TMR_DATA_PORT = $0338; { BASE + 8;}
AD_CMD_PORT = $0330;  { BASE + 0;}
AD_DATA_PORT = $0331; { BASE + 1;}
PAR1_CMD_PORT = $033F; { BASE + 15;}
PAR1_DATA_PORT = $033C; { BASE + 12;}

{*****}

```

```

PROCEDURE TIMER_CONFIG;
{
  This procedure configures all timers.
}
BEGIN
  { Disarm all counters. }
  PORT[TMR_CMD_PORT] := $DF;

  { Load master mode register. } { Load master mode register to : }
  PORT[TMR_CMD_PORT] := $17; { set BCD division }
  { Low byte. } { disable pointer sequencing }
  PORT[TMR_DATA_PORT] := $B0; { set 8 bit data bus width }
  { High byte. } { turn on FOUT gate }
  PORT[TMR_DATA_PORT] := $C1; { set divide by 1 }
  { set F1 as source (freq = 1 MHz) }
  { disable compare 1 and 2 }
  { disable TOD }

  { Load counter 1 mode register. } { Load counter 1 mode register to : }
  PORT[TMR_CMD_PORT] := $01; { set no gating }
  { Low byte. } { count on rising edge }
  PORT[TMR_DATA_PORT] := $02; { source = F4 (freq = 1 MHz/1000 }
  { High byte. } { => .001 sec/count) }
  PORT[TMR_DATA_PORT] := $0E; { disable special gate }
  { reload from load }
  { count once }
  { binary count }
  { count down }
  { output active }

  { Load counter 2 mode register. } { Load counter 2 mode register to : }
  PORT[TMR_CMD_PORT] := $02; { set no gating }
  { Low byte. } { count on rising edge }
  PORT[TMR_DATA_PORT] := $02; { source = F4 (freq = 1 MHz/1000 }
  { High byte. } { => .001 sec/count) }
  PORT[TMR_DATA_PORT] := $0E; { disable special gate }
  { reload from load }
  { count once }
  { binary count }
  { count down }
  { output active }

  { Load counter 3 mode register. } { Load counter 3 mode register to : }
  PORT[TMR_CMD_PORT] := $03; { set no gating }
  { Low byte. } { count on rising edge }
  PORT[TMR_DATA_PORT] := $02; { source = F4 (freq = 1 MHz/1000 }
  { High byte. } { => .001 sec/count) }
  PORT[TMR_DATA_PORT] := $0E; { disable special gate }
  { reload from load }

```



```

        { count once }
        { binary count }
        { count down }
        { output active }

{ Load counter 4 mode register. } { Load counter 4 mode register to : }
PORT[THR_CMD_PORT] := $04; { set no gating }
{ Low byte. } { count on rising edge }
PORT[TMR_DATA_PORT] := $02; { source = F4 (freq = 1 MHz/1000 }
{ } => .001 sec/count) }
{ High byte. } { disable special gate }
PORT[TMR_DATA_PORT] := $0E; { reload from load }
{ count once }
{ binary count }
{ count down }
{ output active }

{ Load counter 5 mode register. } { Load counter 5 mode register to : }
PORT[THR_CMD_PORT] := $05; { set no gating }
{ Low byte. } { count on rising edge }
PORT[TMR_DATA_PORT] := $02; { source = F2 (freq = 1 MHz/1000 }
{ } => .001 sec/count) }
{ High byte. } { disable special gate }
PORT[TMR_DATA_PORT] := $0E; { reload from load }
{ count once }
{ binary count }
{ count down }
{ output active }

END; { timer_config }

{*****}

PROCEDURE LOAD_TIMER(TIMER : INTEGER; TIME_VAL : INTEGER);
{
  This procedure loads TIMER so that it times out after TIME_VAL counts. The
  timer is left counting after the procedure is exited. TIMER refers to
  TIMER_1 through TIMER_5 as defined in the constant section above.
}
VAR
  I, T : INTEGER;

BEGIN
  { Disarm timer. }
  PORT[THR_CMD_PORT] := ($C0 + TIMER);

  { Convert timer code for load register command. }
  T := 0;
  I := -1;
  REPEAT

```

```

BEGIN
  I := I + 1;
  T := TIMER SHR 1;
END;
UNTIL T = 1;
I := I + 1;

{ Load time out value into timer load register. }
PORT[TMR_CMD_PORT] := ($08 + I);
PORT[TMR_DATA_PORT] := LO(TIME_VAL);
PORT[TMR_DATA_PORT] := HI(TIME_VAL);

{ Clear OUT associated with this timer. }
PORT[TMR_CMD_PORT] := ($E0 + I);

{ Load and arm counter to start counting }
PORT[TMR_CMD_PORT] := ($60 + TIMER);
END;

{*****}

FUNCTION TIME_OUT(TIMER : INTEGER) : BOOLEAN;
{
  TIME_OUT is set to true if TIMER has timed out. If TIMER has timed out then
  it is reset to start counting again so after this procedure if a counter has
  timed out it is timing again. TIMER refers to TIMER_1 through TIMER_5 as
  defined in the constant section above.
}
VAR
  I,T,STATUS : INTEGER;

BEGIN
  TIME_OUT := FALSE;
  STATUS := PORT[TMR_CMD_PORT];
  IF (STATUS AND (TIMER SHL 1)) = (TIMER SHL 1) THEN
    BEGIN
      TIME_OUT := TRUE;

      { Convert timer code for clear OUT command. }
      T := 0;
      I := -1;
      REPEAT
        BEGIN
          I := I + 1;
          T := TIMER SHR I;
        END;
      UNTIL T = 1;
      I := I + 1;
    END;
  END;

```

```

    { Clear OUT associated with this timer. }
    PORT[TMR_CMD_PORT] := ($E0 + 1);

    { Load and arm counter to start counting }
    PORT[TMR_CMD_PORT] := ($E0 + TMRB);
END;
END;

{*****}

PROCEDURE TCMR_CONFIG;
{
    This procedure configures the parallel ports of the 8255 on the TECHMAR Lab
    Tender board for ports A and B to read the SONY magnescale BCD output
    (middle four SONY BCD digits). Port C is configured so the lower four bits
    are also inputs, reading the SONY RDY lines, and the upper four bits are set
    up as outputs, setting the SONY LATCH lines.
}
BEGIN
    { Configure TECHMAR 8255 for mode 0, upper part of DIOPORT C set as output }
    { the rest as input. }
    PORT[PAR1_CMD_PORT] := $93;
END;

{*****}

PROCEDURE OUT_TCMR_DIO(DIO_PORT : INTEGER; PAR : BYTE);
{
    OUT_TCMR_DIO sends PAR out the TECHMAR digital IO port. DIO_PORT refers to
    TCMR_DIO_A, TCMR_DIO_B, and TCMR_DIO_C as defined in the constant
    declarations above.
}
BEGIN
    PORT[PAR1_DATA_PORT+DIO_PORT] := PAR;
END;

{*****}

PROCEDURE IN_TCMR_DIO(DIO_PORT : INTEGER; VAR PAR : BYTE);
{
    IN_TCMR_DIO reads PAR from the TECHMAR digital IO port. DIO_PORT refers to
    TCMR_DIO_A, TCMR_DIO_B, and TCMR_DIO_C as defined in the constant
    declarations above.
}
BEGIN
    PAR := PORT[PAR1_DATA_PORT+DIO_PORT];
END;

{*****}

```

```

FUNCTION TECMAR_OK : BOOLEAN;
{
  Does a rough test to see if the Tecmar board is installed. Returns true if
  the tecmar board is ok.
}
VAR ERR_FLG : BOOLEAN;

BEGIN
  LOAD_TIMER(TIMER_4,10); { Set timer 5 to time out in 10 ms. }
  DELAY(100);             { Wait 100 ms. }

  { If timer 5 has not timed out there's a problem. }
  ERR_FLG := NOT TIME_OUT(TIMER_4);

  LOAD_TIMER(TIMER_4,100); { Set timer 5 to time out in 100 ms. }
  DELAY(10);              { Wait 10 ms. }

  { If timer 5 has timed out there's a problem. }
  IF TIME_OUT(TIMER_4) THEN ERR_FLG := TRUE;

  TECMAR_OK := NOT ERR_FLG; { The tecmar is ok if there is no error. }
END;

{*****}

PROCEDURE INIT_TECMAR;

BEGIN
END;

```

```

{*****}
{                               DT2801.INC                               }
{*****}
{
Description of this module:

```

This module contains general routines which deal with the Data Translation DT2801 board.

Procedures in this module:

```

PROCEDURE DT_STOP;
PROCEDURE IN_DT_DIO(DTPORT : INTEGER; VAR DATA : BYTE);
FUNCTION DTAD_READ(AD_CHANL:INTEGER; VAR ERR_FLG:BOOLEAN) : INTEGER;
FUNCTION DTAD_CONV(AD_DATA : INTEGER) : REAL;
PROCEDURE DT_DA(VAR ERR_FLG : BOOLEAN; CHAN : BYTE; VOLTS : REAL);
PROCEDURE SET_INT_CLOCK_PER(PERIOD : REAL);
FUNCTION DT2801_OK : BOOLEAN;
PROCEDURE INIT_DT2801;

```

Global variables used : NONE

```

}
{*****}
{                               Declarations.                               }
{*****}

```

CONST

```

{ Constants needed to use procedures in this module. }
DT_AD_0 = 0;      { DT2801 A/D converter channel 0 }
DT_AD_1 = 1;      { DT2801 A/D converter channel 1 }
DT_DIO_PORT0 = $00; { DT2801 DIO port 0 }
DT_DIO_PORT1 = $01; { DT2801 DIO port 1 }

```

{ Constants global to this module. }

```

DATA_REG = $02EC; {base }
CMD_REG = $02ED; {base + 1}
STATUS_REG = $02ED; {base + 1}
CCLEAR = $01;
CADIN = $0C;
CSTOP = $0F;
CSIN = $04;
CDIOIN = $06;
CDAOUT = $06;
CCLOCK = $03;

```

```

DT_DA_TOP_RANGE = 5; { Top of D/A range. }
DT_DA_BOT_RANGE = -5; { Bottom of D/A range. }
BI = 0; { Bipolar A/D operation. }

```

```

UNI = 1;           { Unipolar A/D operation. }
BASE_RATE = 2.5;   { Base clock rate in microseconds. }

VAR
  { Variables global to this module. }
  AD_GAIN : BYTE;   { Gain code for DT2801: 0 => gain = 1 }
                    { 1 => gain = 2 }
                    { 2 => gain = 4 }
                    { 3 => gain = 8 }
  DT_AD_OP : BYTE;  { DT2801 A/D operating mode: UNIPolar (0) or BIPolar (1). }

{*****}

PROCEDURE DT_STOP;
{
  This procedure stops and clears the DataTranslation board.
}
VAR
  TEMP : BYTE;

BEGIN
  PORT[CMD_REG] := CSTOP; { Stop the DT2801 board. }
  TEMP := PORT[DATA_REG]; { Clear data out ready flag. }

  { Wait until data in full flag is clear and ready flag is set then write }
  { clear error command to command register. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
  PORT[CMD_REG] := CCLEAR;
END; { dt_stop }

{*****}

PROCEDURE IN_DT_DIO(DTPORT : BYTE; VAR DATA : BYTE);
{
  This procedure reads the DT2801 port given by DTPORT, and returns the result
  in DATA. DTPORT refers to DT_DIO_PORT0 and DT_DIO_PORT1 as defined in the
  constant declarations above.
}
BEGIN
  DT_STOP;      { clear and stop DT2801 }

  { Wait until DATA IN FULL flag is clear and READY flag is set, then }
  { write the READ DIO IMMEDIATE command byte to the command register. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
  PORT[CMD_REG] := CDIOIN;

  { Wait until the DATA IN FULL flag is clear, then write the DIO PORT }
  { SELECT byte to the data in register. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;

```

```

PORT[DATA_REG] := DTPORT; { select port }

DELAY(3); { This delay added to keep program from hanging up. }
          { It's not clear why its needed or why it works. }

{ Wait until the DATA OUT READY flag is set, then read the DIO data }
{ byte from the data out register. }
REPEAT UNTIL (PORT[STATUS_REG] AND $05) <> $00;
DATA := PORT[DATA_REG]; { read the data on port }

{ Wait until DATA IN FULL flag is clear and READY flag is set, indicating }
{ command completion, then check the status register ERROR flag. }
REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
IF (PORT[STATUS_REG] AND $80) = $80 THEN ERR_FLG := TRUE;
END; { in_dt_dio }

{*****}

FUNCTION DTAD_READ(AD_CHANL:INTEGER; VAR ERR_FLG:BOOLEAN) : INTEGER;
{
  Reads the DT2801 AD channel given by AD_CHANL and returns the result. The
  boolean ERR_FLG is set to true if an error is encountered.
}
VAR
  AD_HI,AD_LO : INTEGER;

BEGIN
  DT_STOP; { Stop and clear DT2801 board. }

  { Wait until DT2801 boards data in full flag is clear and ready flag is }
  { set, then write the read D/A immediate command to the command register. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
  PORT[CMD_REG] := CADIN;

  { Wait until DT2801 boards data in full flag is clear, then write the A/D }
  { gain byte to the data in register. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
  PORT[DATA_REG] := AD_GAIN;

  { Wait until DT2801 boards data in full flag is clear, then write the A/D }
  { channel byte to the data in register. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
  PORT[DATA_REG] := AD_CHANL;

  { Read two bytes of A/D data from the data out register waiting for a set }
  { data out ready flag before each read, and combine the two bytes into }
  { one word. }
  REPEAT UNTIL (PORT[STATUS_REG] AND $05) <> $00;
  AD_LO := PORT[DATA_REG];

```

```

REPEAT UNTIL (PORT[STATUS_REG] AND $05) <> $00;
AD_HI := PORT[DATA_REG];
DTAD_READ := AD_LO + AD_HI SHL 8;

{ Wait until data in full flag is clear and ready flag is set, indicating }
{ command completion, then check the status register for errors.          }
REPEAT UNTIL (PORT[STATUS_REG] AND $04) = $04;
IF (PORT[STATUS_REG] AND $80) = $80 THEN ERR_FLG := TRUE;
END; { dtad_read }

{*****}

FUNCTION DTAD_CONV(AD_DATA : INTEGER) : REAL;
{
  Converts the digital data from the A/D converter to a REAL value. The
  programmed gain is expected to be in the global variable AD_GAIN, and the
  the operation of the A/D (unipolar or bipolar) is expected in DT_AD_OP.
}
CONST
  { translates the programmed gain code into actual gains. }
  GAIN : ARRAY[1..4] OF BYTE = (1,2,4,8);

VAR
  FACTOR,          { Digital to real conversion factor. }
  UNI_VOLTS : REAL; { Result for unipolar ad operation.  }

BEGIN
  { Calculate digital to real conversion factor and use it to get unipolar }
  { voltage.                                                                }
  FACTOR := (10/4096) / GAIN[AD_GAIN + 1];
  UNI_VOLTS := AD_DATA * FACTOR;

  { If the D/A is actually working in unipolar operation return the      }
  { unipolar voltage, else calculate the bipolar voltage from the unipolar }
  { voltage calculated above.                                              }
  IF DT_AD_OP = UNI THEN DTAD_CONV := UNI_VOLTS
  ELSE DTAD_CONV := UNI_VOLTS * 2 - (10 / GAIN[AD_GAIN + 1]);
END; { dtad_conv }

{*****}

PROCEDURE DT_DA(VAR ERR_FLG : BOOLEAN; CHAN : BYTE; VOLTS : REAL);
{
  Outputs the voltage, VOLTS to channel, CHAN using the DT2801 D/A. The range
  settings of the D/A are expected to be in the constants DT_DA_BOT_RANGE and
  DT_DA_TOP_RANGE.
}
VAR
  RANGE : REAL;          { Output voltage range. }

```



```
DATA_VALUE : REAL;
HIGH,LOW : BYTE;
```

```
BEGIN
```

```
DT_STOP;           { Stop and clear DT2801 board. }
```

```
{ Calculate DAC data word value. }
RANGE := DT_DA_TOP_RANGE - DT_DA_BOT_RANGE;
DATA_VALUE := (VOLTS - DT_DA_BOT_RANGE) * 4096/RANGE;
DATA_VALUE := ABS(DATA_VALUE);
IF DATA_VALUE > 4095 THEN DATA_VALUE := 4095;
```

```
{ Wait until DATA IN FULL flag is clear and READY flag is set, indicating }
{ command completion, then write the WRITE DAC IMMEDIATE command byte to }
{ the to command register. }
REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
PORT[CMD_REG] := CDAOUT;
```

```
{ Wait until the DATA IN FULL flag is clear, then write the DAC SELECT byte }
{ to the data in register. }
REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
PORT[DATA_REG] := CHAN; { select port }
```

```
{ Divide DATA_VALUE into high and low bytes and write both bytes to the }
{ data in register, waiting for a clear DATA IN FULL flag before each }
{ write. }
HIGH := TRUNC(DATA_VALUE / 256);
LOW := TRUNC(DATA_VALUE - HIGH * 256);
REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
PORT[DATA_REG] := LOW;
REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
PORT[DATA_REG] := HIGH;
```

```
{ Wait until DATA IN FULL flag is clear and READY flag is set, indicating }
{ command completion, then check the status register ERROR flag. }
REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
IF (PORT[STATUS_REG] AND $80) = $80 THEN ERR_FLG := TRUE;
END; { dt_da }
```

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

```
PROCEDURE SET_INT_CLOCK_PER(VAR ERR_FLG : BOOLEAN; PERIOD : REAL);
```

```
{
  PERIOD must be between 2*BASE_RATE and BASE_RATE*65535.
}
```

```
VAR TICKS : INTEGER;
    HIGH,LOW : BYTE;
```

```
BEGIN
```

```

DT_STOP; { Stop and clear DT2801 board. }
TICKS := TRUNC(PERIOD / BASE_RATE);

{ Wait until DATA IN FULL flag is clear and READY flag is set, indicating }
{ command completion, then write the SET CLOCK RATE command byte to }
{ the to command register. }
REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
PORT[CMD_REG] := CCLOCK;

{ Divide TICKS into high and low bytes and write both bytes to the }
{ data in register, waiting for a clear DATA IN FULL flag before each }
{ write. }
HIGH := TICKS DIV 256;
LOW := TICKS - HIGH * 256;
REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
PORT[DATA_REG] := LOW;
REPEAT UNTIL (PORT[STATUS_REG] AND $02) = $00;
PORT[DATA_REG] := HIGH;

{ Wait until DATA IN FULL flag is clear and READY flag is set, indicating }
{ command completion, then check the status register ERROR flag. }
REPEAT UNTIL (PORT[STATUS_REG] AND $06) = $04;
IF (PORT[STATUS_REG] AND $80) = $80 THEN ERR_FLG := TRUE;
END; { set_int_clock_per }

{*****}

FUNCTION DT2801_OE : BOOLEAN;
{
  Check DT2801 Digital I/O ports A, B, and lower half of C for input, upper
  half of C for output, and put zero volts on D/A outputs.
}
VAR
  TEMP : BYTE;
  ERR_FLG : BOOLEAN;

BEGIN
  ERR_FLG := FALSE;
  LOAD_TIMER(TIMER_4,1000);

  { stop and clear DataTranslation board }
  PORT[CMD_REG] := CSTOP;
  TEMP := PORT[DATA_REG];
  REPEAT ERR_FLG := TIME_OUT(TIMER_4);
  UNTIL ((PORT[STATUS_REG] AND $06) = $04) OR ERR_FLG;
  PORT[CMD_REG] := CCLEAR;

  { Wait until DATA IN FULL flag is clear and READY flag is set, }
  { then write the SET DIGITAL PORT FOR INPUT command byte to the }

```

```

{ DATA IN register. }
REPEAT ERR_FLG := TIME_OUT(TIMER_4);
UNTIL ((PORT[STATUS_REG] AND $06) = $04) OR ERR_FLG;
PORT[CMD_REG] := CSIN;

{ Wait until DATA IN FULL flag is clear then write the DIGITAL PORT }
{ select byte to the DATA IN register }
REPEAT ERR_FLG := TIME_OUT(TIMER_4);
UNTIL ((PORT[STATUS_REG] AND $02) = $00) OR ERR_FLG;
PORT[DATA_REG] := $01; { choose both DIO ports (0 and 1) }

{ Wait until DATA IN FULL flag is clear and READY flag is set, indicating }
{ command completion, then check the status register ERROR flag. }
REPEAT ERR_FLG := TIME_OUT(TIMER_4);
UNTIL ((PORT[STATUS_REG] AND $06) = $04) OR ERR_FLG;
IF (PORT[STATUS_REG] AND $80) = $80 THEN ERR_FLG := TRUE;

IF NOT ERR_FLG THEN
BEGIN
  { set output voltages of DT2801 D/A to zero volts. }
  IN_DA[ERR_FLG,0,0];
  IN_DA[ERR_FLG,1,0];
END;
IF ERR_FLG OR NOT ERR_FLG;
END; { $-SCN01_2 }

{*****}

PROGRAMME INIT_DT2801;
{
  Initialization module DT2801.
}
BEGIN
  IN_DA[0] := 0; { assumed that dt2801 is jumpered for bipolar a/d }
  { operation. }
  AD_GAIN := 0; { => gain = 1 }
END;

```

```

{*****}
{                               SONY.INC                               }
{*****}
{
Description of this module:

    This module contains routines which deal with the SONY gauges.

Procedures in this module:

PROCEDURE WAIT_SONY_RDY(SONY : BYTE);
PROCEDURE LATCH_SONY;
PROCEDURE UNLATCH_SONY;
PROCEDURE SONY_CONFIG;
FUNCTION CONV(DATA : MAGNESCALE): REAL;
FUNCTION SONY(SNY : INTEGER) : REAL;
FUNCTION SONY_OK : BOOLEAN;
PROCEDURE INIT_SONY;

Global variables used : NONE
}
{*****}
{                               Declarations.                               }
{*****}

CONST
{ Constants global to this module. }
X_RDY = $01;      { Bit 0, port C of TECHAR digital      }
                  { IO port, where the SONY measuring X   }
                  { displacement has its RDY line connected. }
Y_RDY = $02;      { Bit 1, port C of TECHAR digital      }
                  { IO port, where the SONY measuring Y   }
                  { displacement has its RDY line connected. }
X_LATCH = $10;    { Bit 4, port C of TECHAR digital      }
                  { IO port, where the SONY measuring X   }
                  { displacement has its LATCH line connected. }
                  { The upper half of port C must be configured for }
                  { output.                                         }
Y_LATCH = $20;    { Bit 5, port C of TECHAR digital      }
                  { IO port, where the SONY measuring Y   }
                  { displacement has its LATCH line connected. }
                  { The upper half of port C must be configured for }
                  { output.                                         }

TYPE
{ TYPE declarations needed by procedures in this module. }
MAGNESCALE = ARRAY[1..3] OF BYTE;

{*****}

```

```

PROCEDURE WAIT_SONY_RDY(SONY : BYTE);
{
  This procedure waits for the SONY RDY line to go low indicating that data
  can be read from the SONY. The byte variable SONY contains the name of the
  SONY gauge to be read (X or Y).
}
VAR
  STATUS : BYTE;
  RDY     : BYTE;

BEGIN
  IF SONY = X THEN RDY := X_RDY { Look at bit 0, port C of TECHMAR digital }
                                { IO port, where the SONY measuring X      }
                                { displacement has its RDY line connected. }
  ELSE RDY := Y_RDY;           { Look at bit 1, port C of TECHMAR digital }
                                { IO port, where the SONY measuring Y      }
                                { displacement has its RDY line connected. }

  { Check if RDY bit from SONY is low implying that the SONY is ready to be }
  { read. }
  REPEAT
    IN_TCMR_DIO(TCMR_DIO_C,STATUS);
  UNTIL (STATUS AND RDY) = $00;      { check if RDY line is set low }
END; { wait_sony_rdy }

{*****}

PROCEDURE LATCH_SONY;
{
  This procedure latches the SONY Magnescale data port so it can be read at a
  later time. There is a time delay of 2 to 4 milliseconds before data is
  ready on the port.
}
BEGIN
  OUT_TCMR_DIO(TCMR_DIO_C,$00);      { send out low bit to latch SONY's }
END;

{*****}

PROCEDURE UNLATCH_SONY;
{
  This procedure unlatches the SONY Magnescale data port so it can be read at
  a later time. There is a time delay of 2 to 4 milliseconds before data is
  ready on the port.
}
BEGIN
  OUT_TCMR_DIO(TCMR_DIO_C,$FF);      { send out hi bit to unlatch SONY's }
END; { latch_sony }

```

```

{*****}

PROCEDURE SONY_CONFIG;
{
  Sets up the SONY so it can be read at a later time.
}
BEGIN
  UNLATCB_SONY;
END; { sony_config }

{*****}

FUNCTION CONV(DATA : MAGNESCALE): REAL;
{
  This function is used to convert SONY data read by the DT2801 or TECHMAR
  boards from a set of BCD digits to a REAL number.
}
VAR
  DIGIT : ARRAY [1..4] OF BYTE;

BEGIN
  DIGIT[1] := (DATA[2] AND $F0) DIV $10; { 10E 0 digit }
  DIGIT[2] := (DATA[2] AND $0F);        { 10E-1 digit }
  DIGIT[3] := (DATA[1] AND $F0) DIV $10; { 10E-2 digit }
  DIGIT[4] := (DATA[1] AND $0F);        { 10E-3 digit }
  CONV := DIGIT[1] + DIGIT[2]/10 + DIGIT[3]/100 + DIGIT[4]/1000;
END; { conv }

{*****}

FUNCTION SONY(SNY : INTEGER) : REAL;
{
  Reads the SONY scale associated with the SNY sony gauge.
}
VAR
  DATA : MAGNESCALE;

BEGIN
  { Latch both sony gauges and wait until the SONY associated SNY signals }
  { its ready. }
  LATCB_SONY;
  WAIT_SONY_RDY(SNY);

  { Read and store SONY BCD data. }
  CASE SNY OF
    X:
      BEGIN
        IN_TCHB_DIO(TCHB_DIO_B, DATA[2]); { get high BCD digits, port B }
        IN_TCHB_DIO(TCHB_DIO_A, DATA[1]); { get lower digits, port A }
      END
  END

```

```

END;
Y:
BEGIN
  IN_DT_DIO(DT_DIO_PORT0,DATA[1]); { get lower digits, port 0 }
  IN_DT_DIO(DT_DIO_PORT1,DATA[2]); { get high BCD digits, port 1 }
END;
END; { case }
UNLATCH_SONY;          { Unlatch both sony gauges. }
SONY := CONV(DATA);    { Convert SONY BCD data to a REAL. }
END; { sony }

{*****}

FUNCTION SONY_OK(SONY : BYTE) : BOOLEAN;
{
  Returns true if the sony gauge specified by SONY is functioning, otherwise
  returns a false.
}
VAR
  STATUS : BYTE;
  RDY : BYTE;
  ERR_FLG : BOOLEAN;

BEGIN
  ERR_FLG := FALSE;          { Assume no errors will occur. }
  IF SONY = X THEN RDY := X_RDY { Look at bit 0, port C of TECMAR digital }
                                { 10 port, where the SONY measuring X }
                                { displacement has its RDY line connected. }
  ELSE RDY := Y_RDY;         { Look at bit 1, port C of TECMAR digital }
                                { 10 port, where the SONY measuring Y }
                                { displacement has its RDY line connected. }

  { Check if RDY bit from SONY is low implying that the SONY is ready to be }
  { read. }
  LATCH_SONY;                { Tell Sony to get data ready on BCD port. }
  LOAD_TIMER(TIMER_4,10);    { Give the SONY RDY line 10 ms to reply. }
  REPEAT
    ERR_FLG := TIME_OUT(TIMER_4); { Don't wait forever. }
    IN_TCMR_DIO(TCMR_DIO_C,STATUS); { Read DIO port. }
  UNTIL ((STATUS AND RDY) = $00) OR ERR_FLG; { Check if RDY line is set low. }

  UNLATCH_SONY;              { Tell Sony it no longer needs to have data ready. }
  DELAY(1);                  { Give LATCH signal chance to go high. }
  IN_TCMR_DIO(TCMR_DIO_C,STATUS); { Read DIO port. }

  { If the LATCH line doesn't return to high there is a problem. }
  IF (STATUS AND RDY) <> RDY THEN ERR_FLG := TRUE;

  SONY_OK := NOT ERR_FLG; { The Sony is OK if no errors occurred. }

```

```
END; { sony_ok }
```

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

```
PROCEDURE INIT_SONY;
```

```
{  
  Initializes SONY module.
```

```
}
```

```
BEGIN
```

```
  SONY_CONFIG;
```

```
END;
```



```

{*****}
{               PROBE.INC               }
{*****}
{

```

Description of this module: This module contains routines which are used to move and stop the probe. To get the probe to a given location along either axis the function TO_LOC is used. This function starts the probe moving to the given location. The function NEAR_LOC is then used to monitor the position of the probe to see if it is near enough to use control to stop it. When the probe is near the rod STOP_AT_LOC is used to stop it within a given error. The record PROBE contains all information about the motion of the probe. PROBE must be reinitialized whenever the axis along which the probe is moving changes. Several utility routines are included which allow other modules to determine the status of the probe without having to access PROBE.

Procedures in this module:

```

PROCEDURE MOTOR(MOTOR,SPEED : BYTE);
PROCEDURE GET_PROBE_POSITION;
FUNCTION PRB_STK : BOOLEAN;
FUNCTION PROBE_STATUS: INTEGER;
PROCEDURE CLEAR_PROBE_STATUS;
FUNCTION PROBE_STOPPED: BOOLEAN;
FUNCTION PROBE_DIR: INTEGER;
FUNCTION PROBE_AXIS: INTEGER;
FUNCTION PROBE_POSITION: REAL;
FUNCTION NEAR_LOC(LOC : INTEGER; ERROR : REAL): BOOLEAN;
FUNCTION PROBE_AT_LOC(LOC,X_Y): BOOLEAN;
FUNCTION PROBE_PAST_LOC(LOC,X_Y): BOOLEAN;
PROCEDURE STOP_PROBE;
PROCEDURE STOP_AT_LOC(LOC : INTEGER; ERROR : REAL);
FUNCTION TO_LOC(LOC,X_Y : INTEGER) : BOOLEAN;
PROCEDURE GO_HOME(RESET : BOOLEAN);
PROCEDURE INIT_ROD_DISP;
PROCEDURE INIT_PROBE;
}
{*****}
{               Declarations.               }
{*****}

```

CONST

```

{ Constants needed to use procedures in this module. }
POS = 1;      { Indicates that the probe is moving in "positive" direction. }
NEG = -1;     { Indicates that the probe is moving in "negative" direction. }

```

```

{ Probe status information. }
CLEAR = 1;
MOVING = 2;
STUCK = 3;

```

```

AT_LOC = 4;
HARD_STOP = 5;

{ bundle information }
DRY_ROD = 1;      { dry rod number }
WET_ROD = 2;      { wet rod number }
FIRST_ROD = 1;
FIRST_ROW = 1;
LAST_ROD = 10;    { number of rods per row, including standards }
LAST_ROW = 8;     { number of rows }
DRY_DISP = 0.230; { displacement of dry standard from stop }
WET_DISP = 0.854; { displacement of wet standard from stop }
DELTA_ROD = 0.641; { distance between rods }

{ Constants global to this module. }
MAX_V = 25;      { Maximum motor voltage. }
MOTOR_CONTROL_GAIN = 5.0; { Gain of the 124-A 24V Motor Control (25/5). }
AMP = 0;         { DA channel associated with motor controller. }
RELAY = 1;       { DA channel used to switch between motors. }
RLY_X = 0;       { Voltage required to switch relay so motor X gets }
                { the input from the motor controller. }
RLY_Y = 4.15;    { Voltage required to switch relay so motor Y gets }
                { the input from the motor controller. }

{ TYPE declarations needed by procedures in this module. }
TYPE
  PROBE_TYPE = RECORD
    STOPPED : BOOLEAN; { TRUE if probe has been stopped. }
    AXIS : INTEGER;    { X or Y. }
    DIR : INTEGER;     { POS or NEG. }
    PRE_POS : REAL;    { Latest position of probe. }
    STK_POS : REAL;    { Probe position when last checked to see if stuck. }
    STATUS : INTEGER;  { NONE, STUCK, etc. }
    STOP_DIST : REAL;  { distance to stop (STOPX or STOPY). }
    CONTROL_GAIN : REAL; { control gain volts/inch (GAINX or GAINY). }
    ERR : REAL;        { maximum allowed error in final position (ERRX or ERRY). }
    STALL_V : REAL;    { Stall voltage of motor. }
    DISP : ARRAY [0..11] OF REAL; { Displacements of rods or rows from stop. }
  END;
  STRING60 = STRING[60];

VAR
  { Variables global to this module. }
  STCK_PERIOD : INTEGER; { Period (in ms) between checks for a stuck probe. }
  SAMPLE_PERIOD : INTEGER; { Period (in ms) of sample for control. }
  STCK_TIMER : INTEGER; { Timer which checks for a stuck probe. }
  STOPX : REAL;        { distance to stop in X direction (inches). }
  STOPY : REAL;        { distance to stop in Y direction (inches). }
  ERRX : REAL;         { Maximum error from final X location (inches). }

```

```

ERRY : REAL;          { Maximum error from final Y location (inches). }
GAINX : REAL;          { Gain of control in X direction (volts/inch) }
GAINY : REAL;          { Gain of control in Y direction (volts/inch) }
STALLX : REAL;         { Stall voltage of X motor. }
STALLY : REAL;         { Stall voltage of Y motor. }
MIN_PRB_MOV : REAL;    { Minimum distance (in mils) probe must move }
                        { between checks before its considered to be stuck. }

DISP_ROW1 : REAL;      { Displacement of first row from stop. }
DISP_ROD1 : REAL;      { Displacement of first rod from stop. }
ROD_DISP : ARRAY [0..11] OF REAL; { displacement of the centers of the }
                        { rods from the stop }
ROW_DISP : ARRAY [0..9] OF REAL; { displacement of the rows from the stop }
MNL_MOTOR_SPEED : BYTE; { 0 - 10 }
PROBE : PROBE_TYPE;    { probe status information. }

```

```

PROCEDURE MOTOR(SPEED : INTEGER);

```

```

{
  Starts a motor in direction determined by the polarity of SPEED. SPEED
  needs to be an integer in the range +/- 10. A zero in SPEED stops the
  motor, a +10 in SPEED applies +24VDC to the motor given by MOTOR, -10 in
  speed applies -24VDC to the motor. SPEED determines what fraction of the
  stall voltage (given by constant STALL_V) to maximum voltage (given by
  constant MAX_V) will be applied to the motors using the D/A converters on
  the DT2801 board and the 124-A motor controller. The global variable PROBE
  is also initialized as appropriate.

```

```

}
VAR
  MOTOR_VOLT : REAL;    { Voltage to be output through the D/A. }
  OLD : REAL;          { OLD displacement of probe. }

```

```

BEGIN

```

```

  WITH PROBE DO

```

```

    BEGIN

```

```

      { If a motor is to be started switch the relay so the appropriate MOTOR }
      { will have power, calculate the appropriate voltage to output from the }
      { D/A, and output the voltage. }

```

```

      IF SPEED = 0 THEN

```

```

        { Take away power from the motor and wait for it to stop coasting. }

```

```

      BEGIN

```

```

        STOPPED := FALSE;    { The motor has not stopped yet. }

```

```

        DT_DA(ERR_PLG,AMP,0); { Take power from motor. }

```

```

        OLD := SONY(AXIS); { Initialize OLD probe position. }

```

```

        REPEAT

```

```

          DELAY(50);          { Wait a while. }

```

```

          PRB_POS := SONY(AXIS); { Get new probe position. }

```

```

        { If the probes position hasn't changed the probe has stopped. }

```

```

        IF OLD = PRB_POS THEN STOPPED := TRUE;

```

```

        OLD := PRB_POS;
    UNTIL STOPPED;
END
ELSE
BEGIN
    IF STOPPED THEN
    BEGIN
        CASE AXIS OF
            X:DT_DA(ERR_FLG,RELAY,RLY_X); { Switch relay so power goes to X motor.}
            Y:DT_DA(ERR_FLG,RELAY,RLY_Y); { Switch relay so power goes to Y motor.}
        END;
        DELAY(200); { Wait for relay to switch over. }
        PRB_POS := SONY(AXIS);
        STK_POS := PRB_POS;
        STOPPED := FALSE;
        STATUS := MOVING;
    END;

    { Calculate the voltage to be sent to the 124-A motor controller. }
    MOTOR_VOLT :=
    SPEED/ABS(SPEED)*(STALL_V+(MAX_V-STALL_V)*ABS(SPEED)/10)/MOTOR_CONTROL_GAIN;

    DT_DA(ERR_FLG,AMP,MOTOR_VOLT); { Send voltage to the 124-A controller. }
    LOAD_TIMER(STCK_TIMER,STCK_PERIOD); { Start probe stuck check timer. }
    END;
END;
END; { motor }

```

```

{*****}

```

```

PROCEDURE GET_PROBE_POSITION;

```

```

{
    Gets the present positon of the probe and updates the variable PROBE. If
    the stuck check timer (STCK_TIMER) has timed out then the variable STK_POS
    is also updated and checked to see if the probe is stuck. If the probe has
    not moved since the last probe check is is is assumed that it is stuck so
    the motor is stopped and the probe status is set to STUCK.
}

```

```

BEGIN
    WITH PROBE DO
    BEGIN
        IF NOT STOPPED THEN
        BEGIN
            PRB_POS := SONY(AXIS);
            IF TIME_OUT(STCK_TIMER) THEN
            BEGIN
                IF ABS(PRB_POS - STK_POS) < MIN_PRB_MOV THEN
                BEGIN
                    STATUS := STUCK;
                END;
            END;
        END;
    END;
END

```

```

        MOTOR(0);
    END;
    STK_POS := PRB_POS;
END; { if }
END; { if }
END; { with }
END; { get_probe_position }

{*****}

FUNCTION PRB_STK : BOOLEAN;
{
    A utility routine used by other modules to see if probe status = STUCK.
}
BEGIN
    IF PROBE.STATUS = STUCK THEN PRB_STK := TRUE ELSE PRB_STK := FALSE;
END; { prb_stk }

FUNCTION PROBE_STATUS: INTEGER;
{
    A utility routine used by other modules to access the probe status flag.
}
BEGIN
    PROBE_STATUS := PROBE.STATUS;
END;

PROCEDURE CLEAR_PROBE_STATUS;
{
    A utility routine used by other modules to clear the probe status flag.
}
BEGIN
    PROBE.STATUS := CLEAR;
END;

FUNCTION PROBE_STOPPED: BOOLEAN;
{
    A utility routine used by other modules to see if the probe has stopped.
}
BEGIN
    PROBE_STOPPED := PROBE.STOPPED;
END;

FUNCTION PROBE_DIR: INTEGER;
{
    A utility routine used by other modules to see the direction in which the
    probe is moving.
}
BEGIN
    PROBE_DIR := PROBE.DIR;

```

```

END;

FUNCTION PROBE_AXIS: INTEGER;
{
  A utility routine used by other modules to find the axis along which the
  probe is presently moving.
}
BEGIN
  PROBE_AXIS := PROBE.AXIS;
END;

FUNCTION PROBE_POSITION: REAL;
{
  A utility routine used by other modules to find the probes present position.
}
BEGIN
  PROBE_POSITION := PROBE.PRB_POS;
END;

{*****}

FUNCTION NEAR_LOC(LOC : INTEGER; ERROR : REAL): BOOLEAN;
{
  Checks to see if the probe is within ERROR inches of LOC (ROW or ROD). If
  ERROR is 0 then the default value of PROBE.ERR is used. The probe should be
  moving when this routine is called since it is assumed that the variable
  PROBE.PRB_POS is being updated.
}
VAR DELTA : REAL;

BEGIN

  WITH PROBE DO
  BEGIN
    IF ERROR = 0 THEN DELTA := ERR ELSE DELTA := ERROR;
    CASE DIR OF
      POS: NEAR_LOC := (PRB_POS > (DISP[LOC]-DELTA*DIR));
      NEG: NEAR_LOC := (PRB_POS < (DISP[LOC]-DELTA*DIR));
    END; { case }
  END; { with }
END; { near_loc }

{*****}

FUNCTION PROBE_AT_LOC(LOC,X,Y : INTEGER; ERROR : REAL): BOOLEAN;
{
  Checks to see if probe is within ERROR inches of LOC (ROW or ROD). If ERROR
  is 0 then the default value of LI is used. The probe should not be moving
  when this function is called since this routine uses the SONY gauges to find

```

```

    the position of the probe which takes a significant amount of time.
}
VAR DELTA : REAL;
    DISP : REAL;

BEGIN
    IF ERROR <> 0 THEN DELTA := ERROR
    ELSE IF X_Y = X THEN DELTA := ERRX ELSE DELTA := ERRY;

    IF X_Y = X THEN DISP := ROD_DISP[LOC] ELSE DISP := ROW_DISP[LOC];

    IF LOC <> 0 THEN
        BEGIN
            IF (SONY(X_Y) < (DISP+DELTA)) AND (SONY(X_Y) > (DISP-DELTA))
            THEN PROBE_AT_LOC := TRUE
            ELSE PROBE_AT_LOC := FALSE;
        END
    ELSE
        BEGIN
            IF (SONY(X_Y) < DISP+DELTA)
            THEN PROBE_AT_LOC := TRUE
            ELSE PROBE_AT_LOC := FALSE;
        END;
    END;

    {*****}

FUNCTION PROBE_PAST_LOC(LOC,X_Y : INTEGER; ERROR : REAL): BOOLEAN;
{
    Checks to see if probe is past ERROR inches of LOC (ROW or ROD). If ERROR
    is 0 then the default value of LI is used. The probe should not be moving
    when this function is called since this routine uses the SONY gauges to find
    the position of the probe which takes a significant amount of time.
}
VAR DISP : REAL;
    DELTA : REAL;

BEGIN
    IF ERROR <> 0 THEN DELTA := ERROR
    ELSE IF X_Y = X THEN DELTA := ERRX ELSE DELTA := ERRY;

    IF X_Y = X THEN DISP := ROD_DISP[LOC]
    ELSE DISP := ROW_DISP[LOC];

    IF (SONY(X_Y) > (DISP+DELTA))
    THEN PROBE_PAST_LOC := TRUE
    ELSE PROBE_PAST_LOC := FALSE;
END;

```

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

```
PROCEDURE STOP_PROBE;
```

```
{  
  Does a hard stop of the probe.  
}
```

```
BEGIN
```

```
  MOTOR(0); PROBE.STATUS := HARD_STOP;  
END;
```

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

```
PROCEDURE STOP_AT_LOC(LOC : INTEGER; ERROR : REAL);
```

```
{  
  Stops probe near the ROD or ROW indicated by LOC. ERROR determines the  
  maximum error (in inches) from the final value that the probe is allowed to  
  stop. If ERROR is set to 0 the probe is stopped within the default value  
  PROBE.ERR. If the probe takes too long to arrive at its final destination  
  the procedure stops trying to stop the probe and exits, updating the probe  
  status before it exits.  
}
```

```
VAR DONE : BOOLEAN;
```

```
  E : REAL;           { Position error. }  
  U,ABSU,U1,U2,U3 : REAL; { Control voltages. }
```

```
BEGIN
```

```
  DONE := FALSE;
```

```
  U1 := 0;
```

```
  U2 := 0;
```

```
  U3 := 0;
```

```
  WITH PROBE DO
```

```
    BEGIN
```

```
      IF ERROR = 0 THEN ERROR := ERR;
```

```
      { give probe 2.5 sec to stop then assume something is wrong. }
```

```
      LOAD_TIMER(TIMER_5,2500);
```

```
      LOAD_TIMER(TIMER_4,SAMPLE_PERIOD);
```

```
      REPEAT
```

```
        PRB_POS := SONY(AXIS); { Get new position. }
```

```
        { Calculate and output new control voltage. }
```

```
        E := (DISP[LOC] - PRB_POS);
```

```
        U := E*CONTROL_GAIN;
```

```
        ABSU := ABS(U);
```

```
        IF ABS(E) < ERROR THEN U := 0
```

```
        ELSE IF ABSU < STALL_V THEN U := STALL_V*U/ABSU
```

```
        ELSE IF ABSU > MAX_V THEN U := MAX_V*U/ABSU;
```

```
        DT_DA(ERR_FLG,AMP,U/MOTOR_CONTROL_GAIN);
```



```

{ check if probe has "landed" within error }
U3 := U2; U2 := U1; U1 := U;
IF (U=0) AND (U1=0) AND (U2=0) AND (U3=0) THEN
BEGIN
  STOP_PROBE;
  DONE := TRUE;
  STATUS := AT_LOC;
END;

{ see if sample time is too short, }
{ if it isn't wait for end of present sample period }
IF (NOT DONE) AND TIME_OUT(TIMER_4) THEN
BEGIN
  STOP_PROBE;
  DONE := TRUE;
  MSG('Sample time too short! Probe stopped, hit any key to continue. ');
  WAIT_FOR_KEY;
END
ELSE IF NOT DONE THEN REPEAT UNTIL TIME_OUT(TIMER_4);
UNTIL DONE OR TIME_OUT(TIMER_5);

IF NOT DONE THEN
BEGIN
  STOP_PROBE;
  STATUS := STUCK;
END;
END; { stop_at_loc }

{*****}

PROCEDURE GO_TO_LOC(LOC,X_Y : INTEGER; ERROR : REAL; SPEED : INTEGER);
{
  Stops probe near the ROD or ROW indicated by LOC if the probe is within
  ERROR inches of where it needs to be. Otherwise TO_LOC starts the motor in
  the correct direction and it is left up to other code to stop the probe.
  X_Y is to contain the axis the probe is to move along(X or Y) and LOC has
  the ROW or ROD where the probe is to stop. If ERROR is set to 0 then the
  probe is stopped within PROBE.ERR inches of LOC. SPEED is the motor speed
  at which the probe is set in motion. If SPEED is set to zero, 10 is used
  as the default. If the probe gets stuck or oscillates the procedure stops
  trying to stop the probe and exits but it does set PROBE.STATUS to STUCK.
}
VAR
  I : INTEGER;          { Counter. }
  NEW,OLD : REAL;       { Probe position indicators. }
  SPD : INTEGER;        { Speed at which the probe is started in motion. }

```

```

BEGIN
  WITH PROBE DO
  BEGIN
    AXIS := X_Y;
    CASE X_Y OF
      X :
        BEGIN
          FOR I := 0 TO LAST_ROW+1 DO DISP[I] := ROW_DISP[I];
          STOP_DIST := STOPX;      { Distance to stop in X. }
          CONTROL_GAIN := GAINX;   { gain of control in X. }
          ERR := ERRX;             { Maximum error in final position. }
          STALL_Y := STALLX;
        END;
      Y :
        BEGIN
          FOR I := 0 TO LAST_ROW+1 DO DISP[I] := ROW_DISP[I];
          STOP_DIST := STOPY;      { Distance to stop in Y. }
          CONTROL_GAIN := GAINY;   { gain of control in Y. }
          ERR := ERRY;             { Maximum error in final position. }
          STALL_Y := STALLY;
        END;
    END; { case }

    IF NOT PROBE_AT_LOC(LOC,AXIS,ERROR) THEN
    BEGIN
      PRB_POS := SONY(AXIS);      { Find the probes position. }

      { See if it needs to go forward or backward. }
      IF PRB_POS > DISP[LOC] THEN DIR:=NEG
      ELSE DIR := POS;

      { Determine how far probe is relative to where it needs to be. }
      IF ABS(DISP[LOC] - PRB_POS) > STOP_DIST THEN
      BEGIN { Probe is outside of its stopping distance. }
        TEXTCOLOR(WHITE+BLINK); MSG(TO_STOP_PRB); TEXTCOLOR(WHITE);
        IF SPEED = 0 THEN SPD := 10 ELSE SPD := SPEED;
        MOTOR(DIR*SPD);
      END
      ELSE STOP_AT_LOC(LOC,ERROR);
    END;
  END; { with }
END; { to_loc }

[*****]

PROCEDURE GO_HOME(RESET : BOOLEAN);
{
  Moves the probe to home position and asks the user to reset the SONY gauges
  if necessary and if RESET is true.

```

```

}
VAR
  PROBE_HOME : BOOLEAN; { Boolean indicating that the probe is home. }

BEGIN
  TEXTCOLOR(WHITE+BLINK);
  MSG('The probe is being moved to its home position.');
```

TEXTCOLOR(WHITE);

```

  PROBE_HOME := TRUE;    { Assume the probe is at its home position. }

  { See if the probe is actually at the X and Y stops. If the probe doesn't }
  { get stuck when the motor is reversed it must not at its stop.           }
  { Check X probe.                                                            }
  PROBE_AXIS := X;
  MOTOR(-10);
  REPEAT
    GET_PROBE_POSITION;
  UNTIL PRB_STK;
  IF SONY(X) > 0.001 THEN PROBE_HOME := FALSE;

  { Check Y stop. }
  PROBE_AXIS := Y;
  MOTOR(-10);
  REPEAT
    GET_PROBE_POSITION;
  UNTIL PRB_STK;
  IF SONY(Y) > 0.001 THEN PROBE_HOME := FALSE;

  IF (PROBE_HOME = FALSE) AND (RESET = TRUE) THEN
    BEGIN
      MSG('Reset SONY gauges then hit any key to continue.');
```

WAIT_FOR_KEY; { Wait for user. }

```

    END;
    MSG(MSG_13);      { Clear message window. }
    CLEAR_PROBE_STATUS;
  END; { go_home }

  (*****)

  PROCEDURE INIT_ROD_DISP;
  {
    Initializes the displacements of the rods from the stops.
  }
  VAR I : INTEGER;

  BEGIN
    ROD_DISP[0] := 0.0;
    ROD_DISP[DRY_ROD] := DRY_DISP;
```

```

ROD_DISP[WET_ROD] := WET_DISP;
ROD_DISP[3] := DISP_ROD1;
FOR I := 4 TO LAST_ROD DO ROD_DISP[I] := ROD_DISP[I-1] + DELTA_ROD;
ROD_DISP[LAST_ROD+1] := 10.0;
ROW_DISP[0] := 0.0;
ROW_DISP[1] := DISP_ROW1;
FOR I := 2 TO LAST_ROW DO ROW_DISP[I] := ROW_DISP[I-1] + DELTA_ROW;
ROW_DISP[LAST_ROW+1] := 10.0;
END;

{*****}

PROCEDURE INIT_PROBE;
{
  Initializes PROBE module.
}
VAR I : INTEGER;

BEGIN
  STCL_TIMER := TIMER_3;
  WITH PROBE DO
    BEGIN
      STOPPED := TRUE;
      AXIS := X;
      DIR := POS;
      PRE_POS := 0;
      STA_POS := 0;
      STATUS := CLEAR;
      STOP_DIST := STOPX;
      CONTROL_GAIN := GAINX;
      ERR := ERRX;
      STALL_V := STALLX;
      FOR I := 0 TO LAST_ROD+1 DO DISP[I] := ROD_DISP[I];
    END;
  { STCL_PERIOD and SAMPLE_PERIOD initialized from disk (see INIT.INC). }
END;

```

```

{*****}
{          SCRPRB.INC          }
{*****}
{

```

Description of this module:

This module contains screen probe movement routines.

The screen probe is a graphic means of showing the motion of the probe through the bundle. The cursor represents the probe and is moved through a graphic representaion of a bundle as the actual probe moves through bundle.

Procedures in this module:

```

PROCEDURE UPDATE_SCREEN_BUNDLE;
PROCEDURE PRINT_BUNDLE(X_OFF,Y_OFF : BYTE);
PROCEDURE INIT_SCRPRB;
}
{*****}
{          Declarations.          }
{*****}

```

CONST

```

{ Constants global to this module. }
X_SCRN_POSITIONS = 45;
Y_SCRN_POSITIONS = 8;

```

VAR

```

{ Variables global to this module. }
X_OLD,Y_OLD : INTEGER; { New and old positions of probe on screen. }
SCR_PROBE : CHAR; { Probe character. }

{ Offsets of screen bundle window. }
XOFF,YOFF : INTEGER;

{ Array holding all characters in the screen bundle. }
SCREEN_BUNDLE : ARRAY [1..X_SCRN_POSITIONS,1..Y_SCRN_POSITIONS] OF CHAR;

```

```

{*****}
{          The following routines maintain the screen probe position.          }
{*****}

{*****}

```

```

PROCEDURE UPDATE_SCREEN_PROBE;
{
    Updates the position of the screen so it follows the position of the probe.
}
VAR X_NEW, Y_NEW : INTEGER;

```



```

X_NEW := TRUNC(PROBE_POSITION/(7.420/X_SCRN_POSITIONS)) + 1;
IF X_NEW <> X_OLD THEN
BEGIN
  GOTOXY(X_OLD+XOFF-1,YOFF+8-Y_OLD);
  WRITE(SCREEN_BUNDLE[X_OLD,Y_OLD]);
  X_OLD := X_NEW;
END;
END; { x }
Y:
BEGIN
  Y_NEW := TRUNC(PROBE_POSITION/(4.860/Y_SCRN_POSITIONS)) + 1;
  IF Y_NEW <> Y_OLD THEN
  BEGIN
    GOTOXY(X_OLD+XOFF-1,YOFF+8-Y_OLD);
    WRITE(SCREEN_BUNDLE[X_OLD,Y_OLD]);
    Y_OLD := Y_NEW;
  END;
END; { y }
END; { case }
GOTOXY(X_OLD+XOFF-1,YOFF+8-Y_OLD);
TEXTCOLOR(WHITE+BLINK); WRITE(SCR_PROBE); TEXTCOLOR(WHITE);
END;
END; { if }
END; { update_screen_probe }

{*****}

PROCEDURE PRINT_BUNDLE(X_OFF,Y_OFF : BYTE);
{
  Prints a graphic representation of a bundle on the screen.
}
VAR
  I,J : BYTE;

BEGIN
  GOTOXY(X_OFF,Y_OFF);
  WRITELN(' DRY WET      1  2  3  4  5  6  7  8');
  GOTOXY(X_OFF,Y_OFF+1);
  FOR I := 1 TO 8 DO
  BEGIN
    GOTOXY(X_OFF,WHEREY+1);
    WRITE(CHR($49-I));
  END;
  HARKBOX(X_OFF+1,Y_OFF+1,
    X_OFF+X_SCRN_POSITIONS+2,Y_OFF+Y_SCRN_POSITIONS+2,1);
  XOFF := X_OFF+2;
  YOFF := Y_OFF+2;
  FOR I := 1 TO X_SCRN_POSITIONS DO
    FOR J := 1 TO Y_SCRN_POSITIONS DO

```

```

    BEGIN
        GOTOXY(1-1+XOFF,J-1+YOFF); WRITE(SCREEN_BUNDLE[1,J]);
    END;
    X_OLD := 1;
    Y_OLD := 1;
    SCR_PROBE := ' ';
END;

{*****}

PROCEDURE INIT_SCRPB;
[
    Initializes the variable screen bundle. Must be called before any other
    routines in this module.
]
VAR
    I,J : INTEGER;

BEGIN
    FOR I := 1 TO X_SCRN_POSITIONS DO
        FOR J := 1 TO Y_SCRN_POSITIONS DO SCREEN_BUNDLE[I,J] := ' ';

    { Fill in rods and standards. }
    FOR I := 1 TO Y_SCRN_POSITIONS DO
        BEGIN
            SCREEN_BUNDLE[2,I] := '0';
            SCREEN_BUNDLE[6,I] := '0';
            FOR J := 0 TO 7 DO SCREEN_BUNDLE[16+J*4,I] := '0';
        END;
    END;
END;

```



```

{*****}
{          DATA.INC          }
{*****}
{
Description of this module:

```

The procedures in this module take data during a run. They also keep track of the data buffers and files used to store the data.

When data is taken (using DATA_ACQ) it is stored in VOLT_DATA. VOLT_DATA contains MAX_DATA_PTS number of data points for each rod in a bundle. GET_RESULTS is called to generate a results file of the data in VOLT_DATA. The results file is named BNXXXXXX.DAT where XXXXXX is the bundle number of the current bundle stored in the global variable BUNDLE.

procedures in this module:

```

PROCEDURE CLR_DATA_BUF;
PROCEDURE DATA_ACQ(ROW, ROD : BYTE);
FUNCTION NOT_ENOUGH_DATA : BOOLEAN;
PROCEDURE GET_RESULTS;
FUNCTION SHOW_RESULTS(BUNDLE BUNDLE_TYPE; CMD : BYTE) BOOLEAN;
FUNCTION VIEW_DATA_FILE(BUNDLE : BUNDLE_TYPE) : BOOLEAN;
PROCEDURE DIRLIST(YPOS : INTEGER);
PROCEDURE INIT_DATA;

```

global variables used :

```

MIN_DRY_POWER, MAX_POWER_RATIO
PLANT, BUNDLE
}
{

```

```

}

{*****}
{               Declarations.               }
{*****}

CONST
{ Constants needed to use procedures in this module. }
  POWER = 0;           { Show reflected signal power on results screen. }
  RESULTS = 1;         { Show results on results screen. }

{ Constants global to this module. }
  MAX_DATA_PTS = 32;   { Number of data points to be taken for each rod. }
                      { Needs to be a multiple of 16. }

TYPE
{ TYPE declarations needed to use procedures in this module. }
{ TYPE declarations needed by procedures in this module. }

  { Data record. }
  DATA_TYPE = RECORD
    START_POS : REAL;
    VOLTAGE : ARRAY[1..MAX_DATA_PTS] OF REAL;
    END_POS : REAL;
  END;

VAR
{ Variables global to this module. }
  ECHO_1 : BYTE;      { A/D channel associated with first echo voltage. }
  ECHO_2 : BYTE;      { A/D channel associated with second echo voltage. }
  RAW_DATA_FILE : FILE OF DATA_TYPE; { Raw data file. }
  RES_SCR : SAVEBUFPTR; { Pointer to RES_SCR. }
  LD : REAL;          { Distance over which data is to be taken. }
  DATA_PERIOD : INTEGER; { Number of timer 5 counts between data points. }
  SAVE_RAW_DATA : CHAR; { If set to 'Y' then raw data is saved to disk. }

  { Echo voltage data buffer. Holds voltages directly from A/D converters. }
  VOLT_DATA : ARRAY[1..LAST_ROW,1..LAST_ROD] OF DATA_TYPE;
{

```

```

}

{*****}

PROCEDURE CLR_DATA_BUF;
{
  Clears the raw data buffer.
}
VAR
  I,J,K : INTEGER;

BEGIN
  { Clear temporary data storage array. }
  FOR K := 1 TO LAST_ROW DO
    FOR I := 1 TO LAST_ROD DO
      BEGIN
        VOLT_DATA[K][I].START_POS := 0.0;
        FOR J := 1 TO MAX_DATA_PTS DO VOLT_DATA[K][I].VOLTAGE[J] := -100.0;
        VOLT_DATA[K][I].END_POS := 0.0;
      END;
    END;
  END; { clear_data_buf }

{*****}

PROCEDURE DATA_ACQ(ROW,ROD : BYTE);
{
  Waits until the probe is within 50 mils of the rod and then takes
  MAX_DATA_PTS readings of the A/D converter associated with the first echo
  height and stores the data points in VOLT_DATA. If the probe gets stuck
  while it's over a rod the routine stops taking data and exits.
}
VAR
  I : INTEGER;          { Counter. }
  VOLTAGE : ARRAY[1..MAX_DATA_PTS] OF INTEGER;

BEGIN
  DT_DA(ERR_FLG,AMP,2.5);
  LOAD_TIMER(TIMER_5,2);
  REPEAT
    GET_PROBE_POSITION;
  UNTIL PRB_STK OR NEAR_LOC(ROD,LD/2);
  FOR I := 1 TO MAX_DATA_PTS DO
    BEGIN
      VOLTAGE[I] := DTAD_READ(ECHO_1,ERR_FLG);
      REPEAT UNTIL TIME_OUT(TIMER_5);
    END;
    VOLT_DATA[ROW][ROD].END_POS := SONY(I);
    VOLT_DATA[ROW][ROD].START_POS := PROBE_POSITION;
  END;

```

```

FOR I:= 1 TO MAX_DATA_PTS DO
  VOLT_DATA[ROW][BOD].VOLTAGE[I] := DTAD_CONV(VOLTAGE[I]);
  MOTOR(10);
END; { data_acq }

{*****}

FUNCTION NOT_ENOUGH_DISK : BOOLEAN;
{
  Returns true if there isn't enough disk space needed to make a run.
}
VAR BYTES_NEEDED : REAL;

BEGIN
  BYTES_NEEDED := 24*80; { bytes needed for results file: BHXXXXX.DAT }

  { bytes needed for raw data file: RWXXXXX.DAT }
  IF SAVE_RAW_DATA = 'Y' THEN
    BYTES_NEEDED := BYTES_NEEDED + (MAX_DATA_PTS*16*LAST_BOD*LAST_ROW);

  IF DISK_BYTES(DRIVE) < BYTES_NEEDED THEN NOT_ENOUGH_DISK := TRUE
  ELSE NOT_ENOUGH_DISK := FALSE;
END; { not_enough_disk }

{*****}

FUNCTION POWER(ROW, ROD : INTEGER): REAL;

TYPE VERY_LONG_VECTOR = ARRAY[0..MAX_DATA_PTS] OF REAL;

VAR I : INTEGER;
    XREAL, YIMAG : VERY_LONG_VECTOR;
    TEMP : REAL;

PROCEDURE fft(var xreal:  very_long_vector;
               var yimag:  very_long_vector;
               numdat:  integer);

VAR
  dc, maxpower, arg, q,
  cntr, pnt0, pnt1, i,
  j, a, b, k, m:          integer;
  prodreal, prodimag, harm, x, y:  real;
  cosary, sinary:         very_long_vector;

PROCEDURE swap(VAR s1,
               s2: real);

VAR
  temp: real;

```

```

BEGIN
  temp := s1;
  s1 := s2;
  s2 := temp;
END;

BEGIN
  j := 0;
  for i := 0 to numdat - 2 do
    begin
      if i < j then
        BEGIN
          swap(xreal[i], xreal[j]);
        END;
      k := numdat div 2;
      while k <= j do
        begin
          j := j - k;
          k := k div 2;
        end;
      j := j + k;
    end;
    maxpower := 0;
    i := numdat;
    while i <> 1 do
      begin
        maxpower := maxpower + 1;
        i := i div 2;
      end;
      harm := 2 * pi / numdat;
      FOR i:= 0 TO numdat - 1 DO
        BEGIN
          sinary[i] := -sin(harm * i);
          cosary[i] := cos(harm * i);
        END;
      a:=2 ; b := 1;
      FOR cntr:= 1 TO maxpower DO
        BEGIN
          pnt0 := numdat DIV a;
          pnt1 := 0;
          FOR k := 0 TO b-1 DO
            BEGIN
              i:= k;
              WHILE i<numdat DO
                BEGIN
                  arg := i + b;
                  IF k = 0 THEN
                    BEGIN
                      prodreal := xreal[arg];

```

```

        prodimag := yimag[arg];
    END
    ELSE
    BEGIN
        prodreal := xreal[arg] * cosary[pnt1] - yimag[arg] * sinary[pnt1];
        prodimag := xreal[arg] * sinary[pnt1] + yimag[arg] * cosary[pnt1];
    END;
    xreal[arg] := xreal[i] - prodreal;
    yimag[arg] := yimag[i] - prodimag;
    xreal[i] := xreal[i] + prodreal;
    yimag[i] := yimag[i] + prodimag;
    i := i + a;
    END;
    pnt1 := pnt1 + pnt0;
    END;
    a := 2 * a;
    b := b * 2;
    END;
END;

BEGIN
    IF VOLT_DATA[ROW][ROD].VOLTAGE[1] <> -100 THEN
    BEGIN
        FOR I := 1 TO MAX_DATA_PTS DO XREAL[I-1] := VOLT_DATA[ROW][ROD].VOLTAGE[I];
        FOR I := 0 TO (MAX_DATA_PTS-1) DO YIMAG[I] := 0;

        FFT(XREAL,YIMAG,MAX_DATA_PTS);

        TEMP := 0;
        FOR I := 1 TO ((MAX_DATA_PTS*8) DIV 32) DO
            TEMP := TEMP + XREAL[I]*XREAL[I]+YIMAG[I]*YIMAG[I];
        POWER := TEMP;
    END
    ELSE POWER := -100;
    END;

    {*****}

    PROCEDURE GET_RESULTS;
    {
        Creates a RESULTS_FILE called BWXXXXXX.DAT (where XXXXXX is the present
        bundle number) which contains the time, date, plant name (in global variable
        PLANT), bundle number (in global variable BUNDLE), and the results of the
        run for bundle number BUNDLE. Also creates a file called BWXXXXXX.DAT which
        contains all the raw data from a run if the variable SAVE_RAW_DATA is set to
        Y.
    }
    TYPE DATA_FILE_TYPE = FILE OF DATA_TYPE;

```

```

VAR
  I,J : INTEGER;           { Counters. }
  RESULT : STRING[4];       { Result for a given rod. }
  TIME_DATE : TIME_DATE_TYPE; { Holds time and date. }
  RESULTS_FILE : TEXT;      { File which is to hold the results. }
  DATA_FILE : DATA_FILE_TYPE; { File of raw data. }
  FILE_NAME : FILE_NAME_TYPE; { Generic file name. }
  POWER_DATA : ARRAY[1..LAST_ROW,1..LAST_ROD] OF REAL;

BEGIN
  { Calculate power for each rod. }
  FOR I := 1 TO LAST_ROW DO
    FOR J := 1 TO LAST_ROD DO POWER_DATA[I,J] := POWER(I,J);

  { Set up RESULTS_FILE so it can be written to. }
  FILE_NAME := CONCAT(DEF_DRV_DIR,'BN',BUNDLE,'.DAT');
  ASSIGN(RESULTS_FILE,FILE_NAME); REWRITE(RESULTS_FILE);

  { First six lines of RESULTS_FILE contain plant name, bundle number, }
  { time, date, minimum dry standard power, and maximum power ratio. }
  WRITELN(RESULTS_FILE,'Plant = ',PLANT);
  WRITELN(RESULTS_FILE,'Bundle = ',BUNDLE);
  GET_TIME(TIME_DATE);
  WITH TIME_DATE DO
  BEGIN
    WRITELN(RESULTS_FILE,'Date = ',MONTH:2,'/',DAY:2,'/',YEAR:4);
    WRITELN(RESULTS_FILE,'Time = ',HOUR:2,':',MINUTE:2,':',SECOND:2);
  END;
  WRITELN(RESULTS_FILE,'Minimum Dry Standard Power = ',MIN_DRY_POWER:7:0);
  WRITELN(RESULTS_FILE,'Maximum Power Ratio = ',MAX_POWER_RATIO:4:2);
  WRITELN(RESULTS_FILE);

  write(RESULTS_FILE,'    dry ');
  write(RESULTS_FILE,'    wet ');
  for i := 1 to 8 do
  begin
    write(RESULTS_FILE,'    ',i,' ');
  end;
  writeln(RESULTS_FILE);

  for i := 1 to LAST_ROW do
  begin
    { Put results of power calculations in results file. }
    WRITE(RESULTS_FILE,CHR(64+I),': ');
    FOR J := DRY_ROD TO LAST_ROD DO
    BEGIN
      IF POWER_DATA[I][J] < 0 THEN WRITE(RESULTS_FILE,' ? ');
      ELSE WRITE(RESULTS_FILE,POWER_DATA[I][J]:7:0);
    END;
  end;

```

```

WRITELN(RESULTS_FILE);
WRITE(RESULTS_FILE, ' ');
FOR J := DRY_ROD TO LAST_ROD DO
BEGIN
  IF POWER_DATA[I][J] >= MIN_DRY_POWER THEN
    WRITE(RESULTS_FILE, ' ')
  ELSE IF POWER_DATA[I][J] >= (MIN_DRY_POWER*MAX_POWER_RATIO) THEN
    WRITE(RESULTS_FILE, ' U ')
  ELSE IF POWER_DATA[I][J] >= 0 THEN
    WRITE(RESULTS_FILE, ' F ')
  ELSE WRITE(RESULTS_FILE, ' ? ')
END;
WRITELN(RESULTS_FILE);
end;
CLOSE(RESULTS_FILE);

IF SAVE_RAW_DATA = 'Y' THEN
BEGIN
  FILE_NAME := CONCAT(DEF_DRV_DIR, 'RW', BUNDLE, '.DAT');
  ASSIGN(DATA_FILE, FILE_NAME); REWRITE(DATA_FILE);
  I := 1;
  REPEAT
    J := 1;
    REPEAT
      IF VOLT_DATA[I][J].VOLTAGE[1] <> -100 THEN
        WRITE(DATA_FILE, VOLT_DATA[I][J])
      ELSE
        BEGIN
          J := LAST_ROD+1;
          I := LAST_ROW+1;
        END;
      J := J + 1;
    UNTIL J > LAST_ROD;
    I := I + 1;
  UNTIL I > LAST_ROW;
  CLOSE(DATA_FILE);
END;
END; { get_results }

{*****}

FUNCTION SHOW_RESULTS(BUNDLE : BUNDLE_TYPE; CMD : BYTE) : BOOLEAN;
{
  Fills the skeleton results screen with information from the results file
  BNXXXXX.DAT. Voltages or results are printed depending on CMD.
}
VAR
  LINE : STRING[255];      { Line read from BNXXXXX.DAT file. }
  X,Y,ROW,ROD : INTEGER;  { Counters. }

```



```

FNAME : FILE_NAME_TYPE; { Generic file name.          }
BNFILE : TEXT;          { Results file.                }
VALID : BOOLEAN;        { File name valid flag.        }
MESSAGE : MSG_TYPE;     { User error message.          }

{-----}
PROCEDURE H_LINE(X1,Y1,X2,LINE : INTEGER);
{ Draws a horizontal LINE at cursor position (X1,Y1) of length, X2-X1. }

VAR
  XPOS,YPOS : INTEGER;
  I : INTEGER;

BEGIN
  XPOS := WHEREX; YPOS := WHEREY;
  GOTOXY(X1,Y1);
  FOR I := 1 TO X2-X1+1 DO
    BEGIN
      WRITE(CHR(LINE));
    END;
  GOTOXY(XPOS,YPOS);
END; { h_line }

{-----}
PROCEDURE RESULTS_SCREEN;
{ Prints the skeleton of the results screen. }

VAR
  I,J : INTEGER;      { Counters. }
  ROW : BYTE;         { Row counter. }

{-----}
PROCEDURE F_L(LE,FC,MC,RE : BYTE);

VAR
  XPOS,YPOS : INTEGER;
  I : INTEGER;

BEGIN
  XPOS := WHEREX; YPOS := WHEREY;
  WRITE(CHR(LE));
  FOR I := 1 TO 47 DO
    IF I MOD 6 <> 0 THEN WRITE(CHR(FC))
    ELSE WRITE(CHR(MC));
  WRITE(CHR(RE));
  GOTOXY(XPOS,YPOS);
END; { f_l }

BEGIN

```

```

CLRSCR; TEXTCOLOR(YELLOW);

{ Draw lines under plant, bundle, time, and date. }
H_LINE(2,4,24,196);
H_LINE(2,7,24,196);
H_LINE(2,10,24,196);
H_LINE(2,13,24,196);

{ Draw bundle and standards. }
GOTOXY(32,2); F_L(218,196,194,191);
FOR I := 1 TO 15 DO
BEGIN
  GOTOXY(WHEREX,WHEREY+1);
  IF I MOD 2 <> 0 THEN F_L(179,32,179,179)
  ELSE F_L(195,196,197,180);
END;
GOTOXY(WHEREX,WHEREY+1); F_L(192,196,193,217);
GOTOXY(WHEREX,WHEREY+1); F_L(218,196,194,191);
GOTOXY(WHEREX,WHEREY+1); F_L(179,32,179,179);
GOTOXY(WHEREX,WHEREY+1); F_L(195,196,197,180);
GOTOXY(WHEREX,WHEREY+1); F_L(179,32,179,179);
GOTOXY(WHEREX,WHEREY+1); F_L(192,196,193,217);

GOTOXY(1,22); H_LINE(1,24,80,205);
TEXTCOLOR(WHITE); LOWVIDEO;

{ Label rods. }
I := 3;
J := 1;
REPEAT
  GOTOXY(30,I); WRITE(J);
  I := I + 2;
  J := J + 1;
UNTIL J > 8;
GOTOXY(29,20); WRITE('DRY');
GOTOXY(29,22); WRITE('WET');

{ Label rows. }
I := 35;
ROW := $41; { "A" }
REPEAT
  GOTOXY(I,1); WRITE(CHR(ROW));
  I := I + 6;
  ROW := ROW + 1;
UNTIL ROW > $48; { "H" }

GOTOXY(2,3); WRITE('PLANT = ');
GOTOXY(2,6); WRITE('BUNDLE = ');
GOTOXY(2,9); WRITE('DATE = ');

```

```

GOTOXY(2,12); WRITE('TIME = ');

NORMVIDEO;
END; { results_screen }

BEGIN
  FNAME := CONCAT(DEF_DRV_DIR,'BN',BUNDLE,'.DAT');
  ASSIGN(BNFILE,FNAME); {$I-} RESET(BNFILE); {$I+} VALID := (IORESULT=0);
  IF NOT VALID THEN
    BEGIN
      MESSAGE := CONCAT
        ('No results file (BN',BUNDLE,'.DAT) exists for bundle # ',BUNDLE,'. ');
      MSG(MESSAGE);
      SHOW_RESULTS := FALSE;
    END
  ELSE
    BEGIN
      SHOW_RESULTS := TRUE;
      WINDOW(1,1,80,25); { Set up window. }
      CLRSCR;

      { Print skeleton results screen. }
      IF RES_SCR = NIL THEN
        BEGIN
          RESULTS_SCREEN;
          SAVE_SCR(RES_SCR);
        END
      ELSE RECALL_SCR(RES_SCR);

      { Fill in screen with plant, bundle, time, and date. }
      TEXTCOLOR(WHITE);
      READLN(BNFILE,LINE); DELETE(LINE,1,8); GOTOXY(10,3); WRITE(LINE);
      READLN(BNFILE,LINE); DELETE(LINE,1,9); GOTOXY(11,6); WRITE(LINE);
      READLN(BNFILE,LINE); DELETE(LINE,1,7); GOTOXY(9,9); WRITE(LINE);
      READLN(BNFILE,LINE); DELETE(LINE,1,7); GOTOXY(9,12); WRITE(LINE);

      { Skip row and column labels in BNXXXXX.DAT file. }
      READLN(BNFILE); READLN(BNFILE); READLN(BNFILE);

      { Put result or voltage explanations on screen depending on whether }
      { voltages or results are to be viewed. }
      IF CHD = POWER THEN
        BEGIN
          GOTOXY(2,18); WRITE('MINIMUM DRY POWER = ',MIN_DRY_POWER:5:0);
          GOTOXY(2,19); WRITE('MAXIMUM POWER RATIO = ',MAX_POWER_RATIO:3:2);
          GOTOXY(2,21); WRITE('? = DATA NOT TAKEN');
        END
      ELSE
        BEGIN

```

```

GOTOXY(2,18); WRITE('F = FAILED ROD');
GOTOXY(2,19); WRITE('U = FAILURE UNCERTAIN');
GOTOXY(2,20); WRITE('? = DATA NOT TAKEN');
END;

{ Put results on screen. }
X := 33;
IF CHD <> POWR THEN
BEGIN
  X := X + 1;
  READLN(BNFILE);
END;
ROW := FIRST_ROW;
REPEAT
  READLN(BNFILE);
  READLN(BNFILE,LINE);
  Y := 20;
  ROD := 1;
  REPEAT
    GOTOXY(X,Y);
    IF CHD <> POWR THEN WRITE(COPY(LINE,(ROD-1)*7+6,3)) { Print result. }
    ELSE WRITE(COPY(LINE,(ROD-1)*7+6,5));
    ROD := ROD + 1;
    IF ROD = 3 THEN Y := 3
    ELSE Y := Y + 2;
  UNTIL ROD > LAST_ROD;
  X := X + 6;
  ROW := ROW + 1;
UNTIL ROW > LAST_ROW;

CLOSE(BNFILE);
ANY_KEY;          { Wait for user to hit any key to exit. }
END;
END; { show_results }

{*****}

FUNCTION VIEW_RAW_DATA(BUNDLE : BUNDLE_TYPE) : BOOLEAN;
{
  Prints out all raw data from RWXXXXXX.DAT file to the screen one screen at a
  time. BUNDLE refers to XXXXX in file name. If there is a problem with the
  file so it can not be printed VIEW_RAW_DATA is set to FALSE otherwise it is
  TRUE.
}
VAR
  FNAME : FILE_NAME_TYPE; { Generic file name.      }
  RAW_DATA_FILE : FILE OF DATA_TYPE; { Raw data file.      }
  VALID : BOOLEAN;        { File valid flag.          }
  I,J,K : INTEGER;        { Counters.                  }

```

```

DATA : DATA_TYPE;      { Data from raw data file. }
MESSAGE : MSG_TYPE;     { User error message.   }
RODS_PER_PAGE : INTEGER; { The data for the given number of rods can fit }
                        { on one screen. }

```

```

BEGIN

```

```

    FNAME := CONCAT(DEF_DRV_DIR,'RW',BUNDLE,'.DAT');
    ASSIGN(RAW_DATA_FILE,FNAME);
    {$I-} RESET(RAW_DATA_FILE); {$I+} VALID := (IORESULT=0);
    IF NOT VALID THEN
    BEGIN
        MESSAGE := CONCAT
            ('No raw data file (RW',BUNDLE,'.DAT) exists for bundle # ',BUNDLE,'. ');
        MSG(MESSAGE);
        VIEW_RAW_DATA := FALSE;
    END

```

```

END

```

```

ELSE

```

```

BEGIN

```

```

    VIEW_RAW_DATA := TRUE;
    RODS_PER_PAGE := 20 DIV ((MAX_DATA_PTS DIV 8) + 3);
    K := 0;

```

```

    { Set up screen. }

```

```

    MAKE_BORDER(0); Y1 := 2; WINDOW(X1,Y1,X2,Y2);

```

```

    MSG('Hit "Q" to Quit, any other key to continue. ');

```

```

    REPEAT

```

```

        GOTOXY(1,1); CLRSCR;

```

```

        I := 1;

```

```

        WHILE (I <= RODS_PER_PAGE) AND (NOT EOF(RAW_DATA_FILE)) DO

```

```

            BEGIN

```

```

                READ(RAW_DATA_FILE,DATA);

```

```

                K := K + 1;

```

```

                WRITE('ROW = ',CHR(65+((K-1) DIV 10)), ' MOD = ');

```

```

                IF (K MOD 10) = 1 THEN WRITE('DRY STANDARD')

```

```

                ELSE IF (K MOD 10) = 2 THEN WRITE('WET STANDARD')

```

```

                ELSE WRITE(CHR(47 + ((K-1) MOD 10)));

```

```

                WRITE('STARTED TAKING DATA AT ',DATA.START_POS:4:3,
                    ', ENDED AT ',DATA.END_POS:4:3);

```

```

            IF NOT EOF(RAW_DATA_FILE) THEN

```

```

                BEGIN

```

```

                    FOR J := 1 TO MAX_DATA_PTS DO

```

```

                        BEGIN

```

```

                            IF (J <> 1) AND ((J MOD 5) = 1) THEN WRITELN;

```

```

                            WRITE(DATA.VOLTAGE[J]:4:2, ' ');

```

```

                        END;

```

```

                    WRITELN;

```

```

                    WRITELN;

```

```

        END;
        I := I + 1;
    END;
    REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID;
    UNTIL (KBD_DATA = $1B) OR (KBD_DATA = $51) OR EOF(RAW_DATA_FILE);
    CLOSE(RAW_DATA_FILE);
END;
END;

{*****}

Procedure DirList(YPOS : INTEGER);
{
    This is a procedure to list out all bundles which have BNXXXXX.DAT files
    associated with them in the current data drive and directory.
}
TYPE
    STRING20 = STRING[20];
var
    MASK    : MASK_STRING_TYPE;
    NamR    : String20;
    i,j,
    Error   : Integer;

begin
    j := 1;
    FillChar(DTA,SizeOf(DTA),0);      { Initialize DTA.      }
    FillChar(NamR,SizeOf(NamR),0);    { Initialize the file name }

    GOTOXY(1,YPOS);
    WRITE('Data files exist for the following bundles : ');
    GOTOXY(1,YPOS+2);

    ERROR := 0;
    MASK := CONCAT(DEF_DRV_DIR,'BN*.DAT');
    FIRST_DIR(MASK,ERROR);    { Get first directory entry in DTA. }

    I := 3;                  { initialize 'I' to the first element }
    if (Error = 0) then
        repeat
            NamR[I] := Chr(Mem[Seg(DTA):Ofs(DTA)+29+I]);
            I := I + 1;
        until (NamR[I-1] = '.') OR (I>20);

        NamR[0] := Chr(I-2);    { set string length because assigning }
                                { by element does not set length }
        gotoxy(62-((7 - j mod 7)*10),whereY);  { Display it }
        write(NamR);
        j := j + 1;
    end if;
end;

```

```

while (Error = 0) do
begin
  ERROR := 0;
  NEXT_DIR(ERROR);

  I := 3;
  repeat
    NamR[I] := Chr(Mem[Seg(DTA):Ofs(DTA)+29+I]);
    I := I + 1;
  until (NamR[I-1] = '.' ) or (I > 20);
  NamR[0] := Chr(I-2);
  if (Error = 0) then
  begin
    if j mod 7 = 0 then
    begin
      gotoxy(62,whereY);
      WriteLn(NamR);
    end
    else
    begin
      gotoxy(62-((7 - j mod 7)*10),whereY);
      write(NamR);
    end;
    j := j + 1;
  end;
end;
IF (J - 1) MOD 7 <> 0 THEN WRITELN;
WRITELN; WRITE('Bytes left on data disk : ',DISK_BYTES(DRIVE):10:0,'. ');
end; { DirList }

{*****}

PROCEDURE INIT_DATA;
{
  Initializes DATA module.
}
BEGIN
  ECHO_1 := DT_AD_0; { A/D channel associated with first echo voltage. }
  ECHO_2 := DT_AD_1; { A/D channel associated with second echo voltage. }

  { Calculate initial guess at data period. Based on probe moving }
  { approximately 1 inch/sec and timer 5 counting at a rate of 1 MHz/10. }
  DATA_PERIOD := TRUNC(1E5*LD/MAX_DATA_PTS);
  { LD initialized from disk; see init.inc. }
END;

{

```

```

{*****}
{                                MENU.INC                                }
{*****}
{

```

Description of this module:

The procedures in this module "paint" their respective menu's on the screen when they are called.

Procedures in this module:

```

PROCEDURE STARTUP_MENU;
PROCEDURE MAIN_MENU;
PROCEDURE RUN_MENU;
PROCEDURE CAL_MENU;
PROCEDURE INIT_MENU;
PROCEDURE RESULT_MENU;
PROCEDURE MNL_MENU;

```

Global variables accessed by procedures in this module :

```

PLANT, BUNDLE
MOTOR_SPEED
SAMPLE_PERIOD, DISP_PERIOD, STCK_PERIOD, BRDW_PERIOD
MIN_DRY_POWER, MAX_POWER_RATIO
LX, LY
DEF_DRV, DEF_DIR
ROD, ROD_DISP
ROW, ROW_DISP

```

```

}
{*****}
{                                Declarations.                                }
{*****}

```

CONST

{ Constants needed to use procedures in this module. }

{ program menu's }

WAIT = 0;

RUN = 1;

CAL = 2;

INIT = 3;

SETUP = 4;

RESULT = 5;

MNL = 6;

{ Program control characters. }

QUIT_CHAR = \$51; { 'Q' }

ESC_CHAR = \$1B; { 'ESC' }

{ Control characters for WAIT_MENU. }


```

CAL_CHAR = $43;      { 'C' }
INIT_CHAR = $49;     { 'I' }
RUN_CHAR = $52;      { 'R' }
HNL_CHAR = $4D;      { 'H' }
EXAMINE_CHAR = $45;  { 'E' }

```

```

{ Control characters for RUN_MENU. }

```

```

{ Control characters for CAL_MENU. }

```

```

DOCAL_CHAR = $43;    { 'C' }
DRY_CHAR = $44;      { 'D' }
WET_CHAR = $57;      { 'W' }

```

```

{ Control characters for INIT_MENU. }

```

```

SAVE_CHAR = $53;     { 'S' }

```

```

{ Control characters for RESULT_MENU. }

```

```

POWR_CHAR = $50;     { 'P' }
RESULT_CHAR = $52;   { 'R' }
VIEW_CHAR = $44;     { 'D' }
LIST_CHAR = $4C;     { 'L' }

```

```

{ Control characters for HNL_MENU. }

```

```

POSI_CHAR = $58;     { 'X' }
POST_CHAR = $59;     { 'Y' }
NEGI_CHAR = $18;     { 'X' }
NEGJ_CHAR = $19;     { 'Y' }
BOD_CHAR = $44;      { 'D' }
ROW_CHAR = $57;      { 'W' }
SPEED_CHAR = $4D;    { 'M' }

```

```

TYPE

```

```

{ TYPE declarations needed to use procedures in this module. }

```

```

  COOR_TYPE = RECORD

```

```

    X : BYTE;  { Screen coordinates of parameter. }

```

```

    Y : BYTE;

```

```

    L : BYTE;  { Maximum length of parameter. }

```

```

  END;

```

```

  STRING76 = STRING[76];

```

```

CONST

```

```

{ Constants needed to use procedures in this module. }

```

```

  { Screen parameter coordinates for each menu. }

```

```

  MAIN_PAR : ARRAY[1..4] OF COOR_TYPE = (par: description
    ((X:10; Y:2; L:6),      { 1 : plant    }
     (X:47; Y:2; L:10),     { 2 : date   }
     (X:10; Y:3; L:6),      { 3 : bundle  }
     (X:47; Y:3; L:8));     { 4 : time   }

```

```

  RUN_PAR  : ARRAY[1..2] OF COOR_TYPE =

```

```

                ((X:1; Y:16; L:60),           { 1 : probe motion }
                (X:1; Y:17; L:60));           { 2 : message   }
CAL_PAR  : ARRAY[1..6] OF COOR_TYPE =
                ((X:22; Y:12; L:10),           { 1 : dry power   }
                (X:22; Y:13; L:10),           { 2 : wet power   }
                (X:25; Y:14; L:10),           { 3 : wet / dry   }
                (X:15; Y:15; L:10),           { 4 : echo height }
                (X:15; Y:16; L:10),           { 5 : probe speed }
                (X:1;  Y:17; L:60));           { 6 : probe motion }
MNL_PAR  : ARRAY[1..5] OF COOR_TYPE =
                ((X:16; Y:6; L:10),            { 1 : echo height }
                (X:1;  Y:8; L:25),            { 2 : probe motion }
                (X:10; Y:9; L:10),            { 3 : x-disp      }
                (X:10; Y:10; L:10),           { 4 : y-disp      }
                (X:1;  Y:12; L:70));           { 5 : message   }

```

VAR

```

{ Variables global to this module. }
{ pointers to various saved screens. }
MAIN_SCR  : SAVEBUFFPTR;
RUN_SCR   : SAVEBUFFPTR;
INIT_SCR  : SAVEBUFFPTR;
CAL_SCR   : SAVEBUFFPTR;
MNL_SCR   : SAVEBUFFPTR;
RESULT_SCR : SAVEBUFFPTR;

```

{*****}

PROCEDURE GOTOPAR(PAR : INTEGER);

VAR X,Y : INTEGER;

BEGIN

CASE MENU OF

WAIT : BEGIN X:=MAIN_PAR[PAR].X; Y:=MAIN_PAR[PAR].Y; END;

RUN : BEGIN X:=RUN_PAR[PAR].X; Y:=RUN_PAR[PAR].Y; END;

CAL : BEGIN X:=CAL_PAR[PAR].X; Y:=CAL_PAR[PAR].Y; END;

INIT : BEGIN END;

RESULT : BEGIN END;

MNL : BEGIN X:=MNL_PAR[PAR].X; Y:=MNL_PAR[PAR].Y; END;

END; { case }

GOTOXY(X,Y);

END;

{*****}

PROCEDURE PRINT_PAR_STRING(PAR : INTEGER; PAR_STRING : STRING76);

VAR X,Y,L,I : INTEGER;

```

BEGIN
CASE MENU OF
  WAIT : BEGIN
    X:=MAIN_PAR[PAR].X; Y:=MAIN_PAR[PAR].Y; L:=MAIN_PAR[PAR].L;
  END;
  RUN : BEGIN
    X:=RUN_PAR[PAR].X; Y:=RUN_PAR[PAR].Y; L:=RUN_PAR[PAR].L;
  END;
  CAL : BEGIN
    X:=CAL_PAR[PAR].X; Y:=CAL_PAR[PAR].Y; L:=CAL_PAR[PAR].L;
  END;
  INIT : BEGIN END;
  RESULT : BEGIN END;
  MNL : BEGIN
    X:=MNL_PAR[PAR].X; Y:=MNL_PAR[PAR].Y; L:=MNL_PAR[PAR].L;
  END;
END; { case }

GOTOXY(X,Y);
I := 1;
WHILE I <= L DO
BEGIN
  IF I <= LENGTH(PAR_STRING) THEN WRITE(PAR_STRING[I]) ELSE WRITE(' ');
  I := I + 1;
END;
END;

{*****}

PROCEDURE STARTUP_MENU;
{
  This menu is called up while the program is initializing variables and
  hardware.
}
BEGIN
  ClrScr;
  MakeBox(1,1,80,24,2);
  CursorOff;
  MakeBox(15,4,65,14,3);
  gotoxy(36,5); write(' Welcome ');
  gotoxy(38,7); write(' to ');
  gotoxy(40 - length(Prog_Name) div 2,9); write(Prog_Name);
  gotoxy(40 - length(Version) div 2,11); write(Version);
  MakeBox(2,16,78,23,3);
  Y1 := 17; WINDOW (X1,Y1,X2,Y2); GOTOXY(1,1);
END; { startup_menu }

{*****}

```

```

PROCEDURE MAIN_MENU;
{
  The main menu, which allows the user to enter all the other sub-menu's.
}
BEGIN
  Y1 := 7;
  IF HAIN_SCR = HIL THEN
    BEGIN
      MAKE_BORDER(Y1-1);
      GOTOXY(34,1); WRITE(' MAIN MENU '); WINDOW(3,2,78,22); GOTOXY(1,1);
      WRITELN; LOWVIDEO;
      WRITELN
      (' ' to do Calibration.          ' ' to Initialize data.);
      WRITELN
      (' ' to start a Run.              ' ' to move probe Manually.);
      WRITELN
      (' ' to Examine results of a run.  ' ' to Quit program.);
      NORMVIDEO;
      TEXTCOLOR(YELLOW);
      GOTOXY(2,2); WRITE(CHR(CAL_CHAR)); GOTOXY(39,2); WRITE(CHR(INIT_CHAR));
      GOTOXY(2,3); WRITE(CHR(RUN_CHAR)); GOTOXY(39,3); WRITE(CHR(MNL_CHAR));
      GOTOXY(2,4); WRITE(CHR(EXAMINE_CHAR)); GOTOXY(39,4); WRITE(CHR(QUIT_CHAR));
      TEXTCOLOR(WHITE);
      WINDOW (X1,Y1,X2,Y2);
      IF NOT HRDM_FAULT THEN
        BEGIN
          LOAD_TIMER(TIMER_5,750); { Update time and date every 750 ms. }
          LOWVIDEO;
          GOTOXY(MAIN_PAR[2].X-7,MAIN_PAR[2].Y); WRITE('DATE = ');
          GOTOXY(MAIN_PAR[4].X-7,MAIN_PAR[4].Y); WRITE('TIME = ');
          NORMVIDEO;
        END;
      GOTOXY(1,5); WRITELN('INSTRUCTIONS:'); WRITELN;
      LOWVIDEO;
      WRITELN('1) Calibrate the system. ');
      WRITELN('2) If necessary, initialize the Plant name. ');
      WRITELN('3) Start run. ');
      WRITELN('4) Examine results of the run. ');
      GOTOXY(MAIN_PAR[1].X-9,MAIN_PAR[1].Y); WRITE('PLANT = ');
      GOTOXY(MAIN_PAR[3].X-9,MAIN_PAR[3].Y); WRITE('BUNDLE = ');
      NORMVIDEO;
      SAVE_SCR(MAIN_SCR);
    END
  ELSE
    BEGIN
      RECALL_SCR(MAIN_SCR);
      WINDOW(X1,Y1,X2,Y2);
    END;
  END;

```

```

NORMVIDEO; TEXTCOLOR(WHITE);
GOTOXY(MAIN_PAR[1].X,MAIN_PAR[1].Y); WRITE(PLANT);
GOTOXY(MAIN_PAR[3].X,MAIN_PAR[3].Y); WRITE(BUNDLE);
END; { main_menu }

{*****}

PROCEDURE RUN_MENU;
{
  This menu is called up when the user wants to make a run.
}
BEGIN
  Y1 := 5;
  IF RUN_SCR = NIL THEN
    BEGIN
      MAKE_BORDER(Y1-1);
      GOTOXY(34,1); WRITE( ' RUN MENU ' ); WINDOW(3,2,78,22); GOTOXY(1,1);
      WRITELN; LOWVIDEO;
      WRITELN
      ('Hit any key to abort run and quit to main menu. ');
      NORMVIDEO;
      WINDOW(X1,Y1,X2,Y2);

      { Print plant, bundle, and probe motion on the screen. }
      LOWVIDEO;
      GOTOXY(1,6); WRITE('PLANT = ');
      GOTOXY(1,7); WRITE('BUNDLE = ');
      NORMVIDEO;
      SAVE_SCR(RUN_SCR);
    END
  ELSE
    BEGIN
      RECALL_SCR(RUN_SCR);
      WINDOW(X1,Y1,X2,Y2);
    END;
  GOTOXY(9,6); WRITE(PLANT);
  GOTOXY(10,7); WRITE(BUNDLE);
  PRINT_BUNDLE(26,4);
  TEXTCOLOR(WHITE+BLINK); MSG('Hit any key to end run. ');
  TEXTCOLOR(WHITE);
END; { run_menu }

{*****}

PROCEDURE CAL_MENU;
{
  The calibration menu is called to set up the equipment.  By moving the probe
  to the wet and dry rods the user can calibrate the USIP 12.  The SONY gauges
  can also be zeroed in this menu.

```

```

)
BEGIN
  Y1 := 6;
  IF CAL_SCR = NIL THEN
    BEGIN
      MAKE_BORDER(Y1-1);
      GOTOXY(32,1); WRITE(' CALIBRATION MENU '); WINDOW(3,2,78,22); GOTOXY(1,1);
      WRITELN; LOWVIDEO;
      WRITELN
      ('' ' to move probe to Dry standard. ' ' to move probe to Wet standard. ');
      WRITELN
      ('' ' to do Calibration. ' ' to Quit to main menu. ');
      NORMVIDEO;
      TEXTCOLOR(YELLOW);
      GOTOXY(2,2); WRITE(CHR(DRY_CHAR)); GOTOXY(39,2); WRITE(CHR(WET_CHAR));
      GOTOXY(2,3); WRITE(CHR(DOCAL_CHAR)); GOTOXY(39,3); WRITE(CHR(QUIT_CHAR));
      TEXTCOLOR(WHITE);
      WINDOW(X1,Y1,X2,Y2); GOTOXY(1,2);
      WRITELN('CALIBRATION PROCEDURE : '); WRITELN; LOWVIDEO;
      WRITELN('1) Use the keys described above to move the probe over the ');
      WRITELN('   desired rods. ');
      WRITELN('2) Follow the directions in the users manual for adjusting the ');
      WRITELN('   echo heights on the USIP 12. ');
      GOTOXY(1,10); WRITE('MINIMUM DRY STANDARD POWER = ');
      GOTOXY(1,11); WRITE('MAXIMUM (WET POWER / DRY POWER) = ');
      GOTOXY(CAL_PAR[1].X-21,CAL_PAR[1].Y); WRITE('DRY STANDARD POWER = 0');
      GOTOXY(CAL_PAR[2].X-21,CAL_PAR[2].Y); WRITE('WET STANDARD POWER = 0');
      GOTOXY(CAL_PAR[3].X-24,CAL_PAR[3].Y); WRITE('WET POWER / DRY POWER = 0.00');
      GOTOXY(CAL_PAR[4].X-14,CAL_PAR[4].Y); WRITE('ECHO HEIGHT = ');
      GOTOXY(CAL_PAR[5].X-14,CAL_PAR[5].Y);
      WRITE('DATA PERIOD = ' x 1E-5 sec');
      SAVE_SCR(CAL_SCR);
    END
  ELSE
    BEGIN
      RECALL_SCR(CAL_SCR);
      WINDOW(X1,Y1,X2,Y2);
    END;
    NORMVIDEO;
    GOTOXY(30,10); WRITE(MIN_DRY_POWER:7:0);
    GOTOXY(35,11); WRITE(MAX_POWER_RATIO:3:2);
    GOTOPAR(5); WRITE(DATA_PERIOD);
  END; { cal_menu }

```

```

{*****}

```

```

PROCEDURE INIT_MENU;

```

```

{
  The initialization menu allows the user to change certain program defaults

```

which are retrieved from disk file INIT.VAR. The new defaults can be permanently saved to disk if the user so chooses.

```
}
{-----}
PROCEDURE PRINT_DEFAULTS;
{ This procedure prints the values of variables which can be changed }
{ in the INITIALIZE MENU. }
```

```
BEGIN
  GOTOXY(17,2); WRITE(PLANT);
  GOTOXY(20,3); WRITE(BUNDLE);
  GOTOXY(27,4); WRITE(SAVE_RAW_DATA);

  GOTOXY(23,6); WRITE(DEF_DRV);
  GOTOXY(26,7); WRITE(DEF_DIR);

  GOTOXY(29,9); WRITE(MIN_DRY_POWER:6:0);
  GOTOXY(23,10); WRITE(MAX_POWER_RATIO:3:2);

  GOTOXY(31,12); WRITE(DISP_ROW1:4:3);
  GOTOXY(31,13); WRITE(DISP_ROD1:4:3);

  GOTOXY(31,15); WRITE(DISP_PERIOD);
  GOTOXY(29,16); WRITE(STCK_PERIOD);
  GOTOXY(31,17); WRITE(SAMPLE_PERIOD);

  GOTOXY(68,2); WRITE(LD:4:3);
  GOTOXY(71,3); WRITE(MIN_PRB_MOV:4:3);

  GOTOXY(67,5); WRITE(STOPX:4:3);
  GOTOXY(66,6); WRITE(ERRR:4:3);
  GOTOXY(71,7); WRITE(GAINX:4:1);
  GOTOXY(68,8); WRITE(STALLX:4:1);

  GOTOXY(67,10); WRITE(STOPY:4:3);
  GOTOXY(66,11); WRITE(ERRR:4:3);
  GOTOXY(71,12); WRITE(GAINY:4:1);
  GOTOXY(68,13); WRITE(STALLY:4:1);
END;
```

```
BEGIN
  Y1 := 6;
  IF INIT_SCR = NIL THEN
    BEGIN
      MAKE_BORDER(Y1-1);
      GOTOXY(30,1); WRITE(' INITIALIZATION MENU ');
      WINDOW(3,2,78,22); GOTOXY(1,1);
      WRITELN; LOWVIDEO;
      WRITELN
```

```

('Press letter associated with default you wish to redefine. ');
WRITELN
(' ' to save defaults to disk.      ' ' to Quit to main menu. ');
NORMVIDEO;
TEXTCOLOR(YELLOW);
GOTOXY(2,3); WRITE(CHR(SAVE_CHAR)); GOTOXY(39,3); WRITE(CHR(QUIT_CHAR));
TEXTCOLOR(WHITE);
WINDOW(X1,Y1,X2,Y2);
LOWVIDEO;
GOTOXY(1,2); WRITE('A) Plant Name = ');
GOTOXY(1,3); WRITE('B) Bundle Number = ');
GOTOXY(1,4); WRITE('C) Save raw data to file? ');
GOTOXY(1,6); WRITE('D) Data file drive = ');
GOTOXY(1,7); WRITE(' Data file directory = ');
GOTOXY(1,9); WRITE('E) Min. dry standard power = ');
GOTOXY(1,10); WRITE(' Max. power ratio = ');
GOTOXY(1,12); WRITE('F) Disp. of first row (in.) = ');
GOTOXY(1,13); WRITE(' Disp. of first rod (in.) = ');
GOTOXY(1,15); WRITE('G) Display Period (mil sec) = ');
GOTOXY(1,16); WRITE(' Probe Period (mil sec) = ');
GOTOXY(1,17); WRITE(' Sample Period (mil sec) = ');
GOTOXY(40,2); WRITE('H) Data acq. window (in.) = ');
GOTOXY(40,3); WRITE(' Min. probe movement (in.) = ');
GOTOXY(40,5); WRITE('I) Stop dist. in X (in.) = ');
GOTOXY(40,6); WRITE(' Max error in X (in.) = ');
GOTOXY(40,7); WRITE(' Control gain in X (V/in.) = ');
GOTOXY(40,8); WRITE(' Stall voltage in X (V) = ');
GOTOXY(40,10); WRITE('J) Stop dist. in Y (in.) = ');
GOTOXY(40,11); WRITE(' Max error in Y (in.) = ');
GOTOXY(40,12); WRITE(' Control gain in Y (V/in.) = ');
GOTOXY(40,13); WRITE(' Stall voltage in Y (V) = ');
NORMVIDEO; TEXTCOLOR(YELLOW);
GOTOXY(1,2); WRITE('A');
GOTOXY(1,3); WRITE('B');
GOTOXY(1,4); WRITE('C');
GOTOXY(1,6); WRITE('D');
GOTOXY(1,9); WRITE('E');
GOTOXY(1,12); WRITE('F');
GOTOXY(1,15); WRITE('G');
GOTOXY(40,2); WRITE('H');
GOTOXY(40,5); WRITE('I');
GOTOXY(40,10); WRITE('J');
TEXTCOLOR(WHITE);
SAVE_SCR(INIT_SCR);
END
ELSE
BEGIN
RECALL_SCR(INIT_SCR);
WINDOW(X1,Y1,X2,Y2);

```



```

END;
PRINT_DEFAULTS;
END; { init_menu }

{*****}

PROCEDURE RESULT_MENU;
{
  The results menu is used to view the results of runs. The user can view
  both voltages and pass/fail information for each bundle for which a run was
  made and data files exist. A text file of raw data can also be created from
  raw data files in the data drive and directory.
}
BEGIN
  Y1 := 7;
  IF RESULT_SCR = NIL THEN
    BEGIN
      MAKE_BORDER(Y1-1);
      GOTOXY(34,1); WRITE(' RESULTS MENU '); WINDOW(3,2,78,22); GOTOXY(1,1);
      WRITELN; LOWVIDEO;
      WRITELN
      (' ' to see Results of a run.      ' to List bundles on data disk. ');
      WRITELN
      (' ' to see Power from each rod.   ' to view raw Data. ');
      WRITELN
      (' ' to Quit to main menu. ');
      NORMVIDEO;
      TEXTCOLOR(YELLOW);
      GOTOXY(2,2); WRITE(CHR(RESULT_CHAR));
      GOTOXY(2,3); WRITE(CHR(PWRF_CHAR));
      GOTOXY(39,3); WRITE(CHR(VIEW_CHAR));
      GOTOXY(39,2); WRITE(CHR(LIST_CHAR));
      GOTOXY(2,4); WRITE(CHR(QUIT_CHAR));
      TEXTCOLOR(WHITE);
      WINDOW(X1,Y1,X2,Y2);
      SAVE_SCR(RESULT_SCR);
    END
  ELSE
    BEGIN
      RECALL_SCR(RESULT_SCR);
      WINDOW(X1,Y1,X2,Y2);
    END;
END; { result_menu }

{*****}

PROCEDURE MNL_MENU;
{
  The manual menu allows the user to manually move the probe. It should be

```

used with caution since it can damage the probe if used improperly.

```
}
VAR
  I,XPOS,YPOS : INTEGER;

BEGIN
  Y1 := 7;
  IF MNL_SCR = NIL THEN
    BEGIN
      MAKE_BORDER(Y1-1);
      GOTOXY(34,1); WRITE(' MANUAL MENU '); WINDOW(3,2,78,22); GOTOXY(1,1);
      WRITELN; LOWVIDEO;
      WRITELN
      ('X' / 'Y' to move probe 'Z' / 'Y' / 'X');
      WRITELN
      (' ' to move probe to roW.      ' ' to move probe to roD.);
      WRITELN
      (' ' to adjust Motor speed.      ' ' to Quit to main menu.);
      NORMVIDEO;
      TEXTCOLOR(YELLOW);
      GOTOXY(2,2); WRITE('X'); GOTOXY(WHEREX+5,WHEREY); WRITE('X');
      GOTOXY(39,2); WRITE('Y'); GOTOXY(WHEREX+5,WHEREY); WRITE('Y');
      GOTOXY(2,3); WRITE(CHR(ROW_CHAR)); GOTOXY(39,3); WRITE(CHR(ROD_CHAR));
      GOTOXY(2,4); WRITE(CHR(SPEED_CHAR)); GOTOXY(39,4); WRITE(CHR(QUIT_CHAR));
      TEXTCOLOR(WHITE);
      WINDOW(X1,Y1,X2,Y2); LOWVIDEO;
      GOTOXY(MNL_PAR[3].X-9,MNL_PAR[3].Y); WRITE('X-DISP = ');
      GOTOXY(MNL_PAR[4].X-9,MNL_PAR[4].Y); WRITE('Y-DISP = ');
      GOTOXY(1,3); WRITE('PLANT = ');
      GOTOXY(1,4); WRITE('BUNDLE = ');
      GOTOXY(MNL_PAR[1].X-15,MNL_PAR[1].Y); WRITE('ECHO VOLTAGE = ');
      SAVE_SCR(MNL_SCR);
    END
  ELSE
    BEGIN
      RECALL_SCR(MNL_SCR);
      WINDOW(X1,Y1,X2,Y2);
    END;
    NORMVIDEO;
    GOTOXY(1,13); WRITE('ROD : ');
    FOR I := 1 TO 10 DO
      BEGIN
        IF I = 1 THEN WRITE(' DRY')
        ELSE IF I = 2 THEN
          BEGIN
            WRITE(' WET'); GOTOXY(WHEREX-1,WHEREY);
          END
        ELSE
          END
      END
    END
```

```

BEGIN
  GOTOXY(WHEREX+6,WHEREY); WRITE(1-2);
END;
END;
GOTOXY(1,14); WRITE('X : ');
FOR I := 1 TO 10 DO WRITE(ROD_DISP[I]:4:3, ' ');

GOTOXY(1,15); WRITE('ROW : ');
FOR I := 1 TO 8 DO WRITE(' ',CHR($40+I));
GOTOXY(1,16); WRITE('Y : ');
FOR I := 1 TO 8 DO WRITE(ROW_DISP[I]:4:3, ' ');
GOTOXY(MNL_PAR[3].X,MNL_PAR[3].Y); WRITE(SONY(X):4:3);
GOTOXY(MNL_PAR[4].X,MNL_PAR[4].Y); WRITE(SONY(Y):4:3);
GOTOXY(10,3); WRITE(PLANT);
GOTOXY(10,4); WRITE(BUNDLE);
GOTOXY(MNL_PAR[1].X,MNL_PAR[1].Y);
WRITE(DTAD_CONV(DTAD_READ(ECHO_1,ERR_FLG)):3:2);
PRINT_BUNDLE(28,1);
END; { mnl_menu }

{*****}

PROCEDURE INITL_MENU;
{
  Initializes MENU module.
}
BEGIN
  MAIN_SCR := PTR(0,0);
  INIT_SCR := PTR(0,0);
  RUN_SCR := PTR(0,0);
  CAL_SCR := PTR(0,0);
  MNL_SCR := PTR(0,0);
  RESULT_SCR := PTR(0,0);

  X1 := 3;           { Constant screen boundaries. }
  X2 := 78;
  Y2 := 22;
END;

```

```

{*****}
{          KBD.INC          }
{*****}
{

```

Description of this module:

The following routines do user command interpretation.

Each procedure in this module is called when a character has been entered by the user while he was in the menu associated with the procedure called. The character entered is then acted on according to the rules of the menu.

Procedures in this module:

```

PROCEDURE WAIT_KBD_CMD(KBD_DATA : BYTE);
PROCEDURE RUN_KBD_CMD(KBD_DATA : BYTE);
PROCEDURE CAL_KBD_CMD(KBD_DATA : BYTE);
PROCEDURE INIT_KBD_CMD(KBD_DATA : BYTE);
PROCEDURE RESULT_KBD_CMD(KBD_DATA : BYTE);
PROCEDURE MNL_KBD_CMD(KBD_DATA : BYTE);
}
{*****}

```

```

PROCEDURE WAIT_KBD_CMD(KBD_DATA : BYTE);
{
  Executed when MENU = WAIT.
}
CONST
  HDWR_PFB = 'Problem with hardware; probe cannot be moved.';

```

```

VAR
  TBUNDLE : BUNDLE_TYPE;
  TPLANT : PLANT_TYPE;
  FILE_NAME : FILE_NAME_TYPE;
  MESSAGE : MSG_TYPE;

```

```

BEGIN
  CASE KBD_DATA OF
    QUIT_CHAR,ESC_CHAR:
      BEGIN
        MSG('Do you really want to quit program (Y/N) ?');
        IF YES THEN
          { Set TERMINATE flag to TRUE, reset window size to full screen. }
          BEGIN
            TERMINATE := TRUE; WINDOW(1,1,80,25); CURSORON;
          END;
        END;
    RUN_CHAR:
      { Before starting a run make sure the user has set up for a run. }

```

```

    { If he has, ask for a bundle number, and use the }
    { bundle number as the name of the raw data file which is opened. }
    { Initialize variables, start motor Y, and enter RUN menu. }
BEGIN
    IF HRDW_FAULT THEN MSG(HDWR_PRB)
    ELSE
    BEGIN
        MESSAGE := CONCAT
('Put data disk into ',DEF_DRV,':', hit any key to continue (ESC to abort).');
        MSG(MESSAGE);
        REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID;
        IF KBD_DATA = $1B THEN MSG(MSG_13)
        ELSE
        BEGIN
            IF NOT_ENOUGH_DISK THEN MSG
('Not enough space on data disk, insert new data disk and try again.')
            ELSE
            BEGIN
                GO_HOME(TRUE);    { Move probe to home position. }

                { Ask for bundle name. }
                CLRESCR; TBUNDLE := ''; GOTOXY(1,5);
                WRITELN('What is the bundle # of the bundle you are examining ? ');
                WRITELN;
                WRITE(''',BUNDLE, '' will be used if no bundle # is given... ');
                READLN(TBUNDLE); IF TBUNDLE <> '' THEN BUNDLE := TBUNDLE;

                CLR_KBD_BUF;    { Clear keyboard buffer of any characters. }
                CLR_DATA_BUF;   { Clear data buffer. }
                MENU := RUN;    { Change MENU to RUN. }
                RUN_MENU;       { Put the run menu on the screen. }

                { Initialize variables. }
                ROW := 1; ROD := 1;
                GO_TO_LOC(ROW,Y,0,0);
            END; { if }
        END; { if }
    END;
END;
MNL_CHAR: { Go to the manual probe movement menu. }
BEGIN
    IF HRDW_FAULT THEN MSG(HDWR_PRB)
    ELSE
    BEGIN
        GO_HOME(TRUE);
        MENU := MNL; MNL_MENU;
    END; { if }
END;
EXAMINE_CHAR: { Go to the "examine results" menu. }

```

```

BEGIN
  MENU := RESULT; RESULT_MENU;
END;
CAL_CHAR: { Go to the Calibration menu. }
BEGIN
  IF HDW_FAULT THEN MSG(HDW_PPB)
  ELSE
    BEGIN
      GO_HOME(TRUE);
      MENU := CAL; CAL_MENU; { Put up calibration menu. }
    END;
  END;
INIT_CHAR:          { Go to initialization menu. }
BEGIN
  MENU := INIT; INIT_MENU;
END;
ELSE ILLEGAL_KEY;   { An illegal key has been entered. }
END; { case }
END; { wait_kbd_cmd }

{*****}

PROCEDURE RUN_KBD_CMD(KBD_DATA : BYTE);
{
  Executed when MENU = RUN.
}
VAR I : INTEGER;

BEGIN
  STOP_PROBE;      { Stop motor. }
END; { run_kbd_cmd }

{*****}

PROCEDURE CAL_KBD_CMD(KBD_DATA : BYTE);
{
  Executed when MENU = CAL.
}
BEGIN
  IF NOT PROBE_STOPPED THEN STOP_PROBE
  ELSE
    BEGIN
      CASE KBD_DATA OF
        QUIT_CHAR,ESC_CHAR:      { Quit calibration menu. }
        BEGIN
          MENU := WAIT; MAIN_MENU; { Return to main menu. }
        END;
        DOCAL_CHAR:
        BEGIN

```

```

IF PROBE_PAST_LOC(0,X,0.010) THEN
BEGIN
    ROW := 3; ROD := 0;
END
ELSE
BEGIN
    ROW := 1; ROD := DRY_ROD;
END;
GO_TO_LOC(ROD,X,0,0);
TEXTCOLOR(WHITE+BLINK); MSG('Hit any key to stop probe.');
```

TEXTCOLOR(WHITE);

```

END;
DRY_CHAR,WET_CHAR:
BEGIN
    ROW := 2;
    IF KBD_DATA = DRY_CHAR THEN ROD := DRY_ROD
    ELSE ROD := WET_ROD;

    { If the probe is already at the desired rod do nothing, otherwise }
    { start the probe moving in the proper direction and show the }
    { probe motion on the screen. }
    GO_TO_LOC(ROD,X,0,0);
    TEXTCOLOR(WHITE+BLINK);
    MSG('Hit any key to stop probe.');
```

TEXTCOLOR(WHITE);

```

END;
ELSE ILLEGAL_KEY;    { An illegal key has been entered. }
END; { case }
END; { if }
END; { cal_kbd_cmd }
```

{*****}

```

PROCEDURE SAVE_TO_DISK; FORWARD;

PROCEDURE INIT_KBD_CMD(KBD_DATA : BYTE);
{
    Executed when MENU = INIT.
}
VAR
    ERROR : BOOLEAN;
    TSTRING : STRING[255]; { Temporary string buffer. }
```

```

BEGIN
CASE KBD_DATA OF
    QUIT_CHAR,ESC_CHAR: { Go to main menu. }
    BEGIN
        MENU := WAIT; MAIN_MENU;
    END;
    SAVE_CHAR: SAVE_TO_DISK;
```

```

$41:          { Get new plant name from user. }
BEGIN
  CLRSCR;
  WRITELN; WRITELN('The present plant name is ',PLANT);
  WRITELN; WRITE('Enter new plant name (CR to skip): ');
  READ(TSTRING); IF TSTRING <> '' THEN PLANT := TSTRING;
  INIT_MENU;
END;

$42:          { Get new bundle number from user. }
BEGIN
  CLRSCR;
  WRITELN; WRITELN('The present bundle number is ',BUNDLE);
  WRITELN; WRITE('Enter new bundle # (CR to skip): ');
  READ(TSTRING); IF TSTRING <> '' THEN BUNDLE := TSTRING;
  INIT_MENU;
END;

$43:          { Save raw data ? }
BEGIN
  CLRSCR;
  WRITELN; IF SAVE_RAW_DATA = 'Y' THEN
    WRITELN('The data is presently being saved.')
  ELSE WRITELN('The data is presently not being saved. ');
  WRITELN; WRITE('Save raw data (Y or N) ? ');
  REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID = TRUE;
  CASE KBD_DATA OF
    $59,$4E: SAVE_RAW_DATA := CHR(KBD_DATA);
    $0D: { Default not changed. }
    ELSE MSG('Y or N. ');
  END;
  INIT_MENU;
END;

$44:          { Get new data drive path from user. }
BEGIN
  CLRSCR;
  WRITELN; WRITELN('The present data drive is ',DEF_DRV);
  WRITELN('The present data directory is ',DEF_DIR);
  WRITELN; WRITE('Enter new drive (CR to skip): ');
  REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID = TRUE;
  CASE KBD_DATA OF
    $41:
      BEGIN
        DRIVE := 1; DEF_DRV := CHR(KBD_DATA); WRITE(DEF_DRV);
      END;
    $42:
      BEGIN
        DRIVE := 2; DEF_DRV := CHR(KBD_DATA); WRITE(DEF_DRV);
      END;
    $43:
      BEGIN

```



```

        DRIVE := 3; DEF_DRV := CHR(KBD_DATA); WRITE(DEF_DRV);
    END;
    $OD: WRITE(DEF_DRV) { Default drive not changed. }
    ELSE MSG('Incorrect drive name entered.');
```

END;

```

WRITELN; WRITE('Enter new directory (CR to skip): ');
READ(TSTRING);
IF TSTRING <> '' THEN DEF_DIR := TSTRING;
DEF_DRV_DIR := CONCAT(DEF_DRV, ':', DEF_DIR);
INIT_MENU;
```

END;

\$45:

```

BEGIN
    CLRSCR;
    WRITELN;
    WRITELN('The present minimum dry standard power is ', MIN_DRY_POWER:8:0);
    WRITELN('The present maximum power ratio is ', MAX_POWER_RATIO:3:2);
    WRITELN;
    WRITE
        ('Enter new minimum dry standard power (CR to skip): ');
    READLN(MIN_DRY_POWER);
    WRITE
        ('Enter new maximum power ratio (CR to skip): ');
    READLN(MAX_POWER_RATIO);
    INIT_MENU;
```

END;

\$46: { Get new row and rod displacements. }

```

BEGIN
    CLRSCR;
    WRITELN;
    WRITELN('Present displacement of first row = ', DISP_ROW1:4:3);
    WRITELN('Present displacement of first rod = ', DISP_ROD1:4:3);
    WRITELN;
    WRITE('Enter new first row displacement (CR to skip): ');
    READLN(DISP_ROW1);
    WRITE('Enter new first rod displacement (CR to skip): ');
    READLN(DISP_ROD1);
    INIT_ROD_DISP;
    INIT_SCRPRB;
    INIT_MENU;
```

END;

\$47:

```

BEGIN
    CLRSCR;
    WRITELN;
    WRITELN('The present display period is ', DISP_PERIOD, ' ms.');
```

```

WRITE('Enter new display period (in milli sec) (CR to skip): ');
READLN(DISP_PERIOD);
LOAD_TIMER(DISP_TIMER,DISP_PERIOD);
WRITE('Enter new probe check period (in milli sec) (CR to skip): ');
READLN(STCK_PERIOD);
LOAD_TIMER(STCK_TIMER,STCK_PERIOD);
WRITE('Enter new sample period (in milli sec) (CR to skip): ');
READLN(SAMPLE_PERIOD);
INIT_MENU;
END;
$48:
BEGIN
  CLRSCR;
  WRITELN;
  WRITELN('LD: Data aquisition window (in.) = ',LD:4:3);
  WRITELN('MIN_PRB_MOV = ',MIN_PRB_MOV:4:3);
  WRITELN;
  WRITE('Enter new LD (CR to skip): '); READLN(LD);
  WRITE('Enter new MIN_PRB_MOV (CR to skip): '); READLN(MIN_PRB_MOV);
  INIT_MENU;
END;
$49:
BEGIN
  CLRSCR;
  WRITELN;
  WRITELN('STOPX: Stop dist. in X (in.) = ',STOPX:4:3);
  WRITELN('ERRX: Max. error in X (in.) = ',ERRX:4:3);
  WRITELN('GAINX: Control gain in X (V/in) = ',GAINX:4:1);
  WRITELN('STALLX: Stall voltage in X (V) = ',STALLX:4:1);
  WRITELN;
  WRITE('Enter new STOPX (CR to skip): '); READLN(STOPX);
  WRITE('Enter new ERRX (CR to skip): '); READLN(ERRX);
  WRITE('Enter new GAINX (CR to skip): '); READLN(GAINX);
  WRITE('Enter new STALLX (CR to skip): '); READLN(STALLX);
  INIT_MENU;
END;
$4A:
BEGIN
  CLRSCR;
  WRITELN;
  WRITELN('STOPY: Stop dist. in Y (in.) = ',STOPY:4:3);
  WRITELN('ERRY: Max. error in Y (in.) = ',ERRY:4:3);
  WRITELN('GAINY: Control gain in Y = ',GAINY:4:1);
  WRITELN('STALLY: Stall voltage in Y (V) = ',STALLY:4:1);
  WRITELN;
  WRITE('Enter new STOPY (CR to skip): '); READLN(STOPY);
  WRITE('Enter new ERRY (CR to skip): '); READLN(ERRY);
  WRITE('Enter new GAINY (CR to skip): '); READLN(GAINY);
  WRITE('Enter new STALLY (CR to skip): '); READLN(STALLY);

```

```

        INIT_MENU;
    END;
    ELSE ILLEGAL_KEY;    { An illegal key has been entered. }
END; { case }
END; { init_kbd_cmd }

{*****}

PROCEDURE RESULT_KBD_CMD(KBD_DATA : BYTE);
{
    Executed when MENU = RESULT.
}
VAR CMD : BYTE;          { Command for SHOW_RESULTS procedure. }
    TBUNDLE : BUNDLE_TYPE;

BEGIN
    CASE KBD_DATA OF
        QUIT_CHAR,ESC_CHAR: { Quit to main menu. }
        BEGIN
            MENU := WAIT; MAIN_MENU;
        END;
        POWR_CHAR, RESULT_CHAR: { Show results. }
        { Ask user which bundle data he wants to examine and open the }
        { corresponding file if it exists and show the results,      }
        { otherwise give an error message.                            }
        BEGIN
            TBUNDLE := '';
            GOTOXY(1,1); WRITE('Which bundle data do you want to examine ? ');
            GOTOXY(1,2);
            WRITE('','BUNDLE',' will be used if no bundle # is given... ');
            READLN(TBUNDLE); IF TBUNDLE <> '' THEN BUNDLE := TBUNDLE;
            IF KBD_DATA = POWR_CHAR THEN CMD := POWR
            ELSE CMD := RESULTS;
            IF SHOW_RESULTS(BUNDLE,CMD) THEN RESULT_MENU
            ELSE
                BEGIN
                    GOTOXY(1,1); CLREOL;
                    GOTOXY(1,2); CLREOL;
                END;
            END;
        VIEW_CHAR :          { View a text file of raw data. }
        { Ask user which bundle data he wants to examine and open the }
        { corresponding BXXXXXXXX.DAT file and print its data information }
        { on screen.                                                    }
        BEGIN
            TBUNDLE := '';
            GOTOXY(1,1); WRITELN('Which raw bundle data do you want to examine ? ');
            GOTOXY(1,2);
            WRITE('','BUNDLE',' will be used if no bundle # is given... ');

```

```

    READ(TBUNDLE); IF TBUNDLE <> '' THEN BUNDLE := TBUNDLE;
    IF VIEW_RAW_DATA(BUNDLE) THEN RESULT_MENU
    ELSE
    BEGIN
        GOTOXY(1,1); CLREOL;
        GOTOXY(1,2); CLREOL;
    END
END;
LIST_CHAR:
BEGIN
    CLRSCR; DIRLIST(4);
END;
ELSE ILLEGAL_KEY; { An illegal key has been entered. }
END; { case }
END; { result_kbd_cmd }

{*****}

PROCEDURE HNL_KBD_CMD(KBD_DATA : BYTE);
{
    Executed when MENU = RUN.
}
VAR GO : BOOLEAN;

BEGIN
    IF NOT PROBE_STOPPED THEN
    BEGIN
        STOP_PROBE;
        MSG(MSG_13);
    END
    ELSE
    BEGIN
        CASE KBD_DATA OF
            QUIT_CHAR,ESC_CHAR:      { Exit to main menu. }
            BEGIN
                MENU := WAIT; MAIN_MENU;
            END;
            POSX_CHAR,NEGX_CHAR:
            BEGIN
                IF KBD_DATA = POSX_CHAR THEN ROD := LAST_ROD+1 ELSE ROD := 0;
                GO_TO_LOC(ROD,X,0,HNL_MOTOR_SPEED);
            END;
            POSY_CHAR,NEGY_CHAR:
            BEGIN
                GO := TRUE;
                IF PROBE_PAST_LOC(WRT_ROD,X,0.5) THEN
                BEGIN
                    MSG('Probe is inside bundle, do you still want to move it (Y/N)?');
                    GO := YES;
                END
            END
        END
    END

```

```

END;
IF GO THEN
BEGIN
  IF KBD_DATA = POSY_CHAR THEN ROW := LAST_ROW+1 ELSE ROW := 0;
  GO_TO_LOC(ROW,Y,0,HNL_MOTOR_SPEED);
END;
END;
ROW_CHAR:
BEGIN
  MSG('Enter rod : "Z"ero, "D"ry, "W"et, "1"-"8", ESC to abort. ');
  REPEAT
    REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID;
    CASE KBD_DATA OF
      $31..$38: BEGIN ROD := KBD_DATA - $30 + 2; MSG(MSG_13); END;
      $5A: BEGIN ROD := 0; MSG(MSG_13); END;
      $44: BEGIN ROD := DRY_ROD; MSG(MSG_13); END;
      $57: BEGIN ROD := WET_ROD; MSG(MSG_13); END;
      $1B: MSG(MSG_13);
      ELSE BEGIN WRITE('G'); KBD_VALID := FALSE; END;
    END; { case }
  UNTIL KBD_VALID;

  IF KBD_DATA <> $1B THEN GO_TO_LOC(ROD,X,0,0);
END;
ROW_CHAR:
BEGIN
  IF PROBE_PAST_LOC(WET_ROD,X,0.5) THEN
    MSG('Cannot move to another row until probe is removed from bundle. ')
  ELSE
    BEGIN
      MSG('Enter row : "Z"ero, "A" - "H", ESC to abort. ');
      REPEAT
        REPEAT KBD_INP(KBD_DATA,KBD_VALID); UNTIL KBD_VALID;
        CASE KBD_DATA OF
          $41..$48: BEGIN ROW := KBD_DATA - $40; MSG(MSG_13); END;
          $5A: BEGIN ROW := 0; MSG(MSG_13); END;
          $1B: MSG(MSG_13);
          ELSE BEGIN WRITE('G'); KBD_VALID := FALSE; END;
        END; { case }
      UNTIL KBD_VALID;

      IF KBD_DATA <> $1B THEN GO_TO_LOC(ROW,Y,0,0);
    END;
  END;
SPEED_CHAR:
BEGIN
  GOTOPAR(5);
  WRITE('Present motor speed is ',hnl_motor_speed,
    '. Enter new motor speed (1-10) : ');

```

```

        READ(MNL_MOTOR_SPEED); GOTOPAR(5); CLREOL;
    END;
    ELSE ILLEGAL_KEY; { An illegal key has been entered. }
END; { case }
END;
END; { mnl_kbd_cmd }

{*****}

PROCEDURE INIT_KBD;

BEGIN
    BES_SCR := PTR(0,0);
END;

```

```

{*****}
{                               EXEC.INC                               }
{*****}
{

```

Description of this module:

This module contains actual execution routines.

The routine associated with a menu is called continuously while the user remains in a given menu. The routine does the repetitive tasks and/or logic functionally needed in the menu.

Procedures in this module:

```

PROCEDURE EXEC_WAIT;
PROCEDURE EXEC_RUN;
PROCEDURE EXEC_CAL;
PROCEDURE EXEC_INIT;
PROCEDURE EXEC_RESULT;
PROCEDURE EXEC_MNL;
PROCEDURE INIT_EXEC;

```

Global variables accessed by procedures in this module :

```

    ROD, ROD_DISP
    ROW, ROW_DISP
}
{*****}

```

```

PROCEDURE EXEC_WAIT;
{
    Executed when MENU = WAIT.
}
VAR
    TIME_DATE : TIME_DATE_TYPE;

```

```

BEGIN
    { If timer 5 has timed out and there is no hardware fault which would }
    { keep the timer from functioning, then print time and date.           }
    IF (TIME_OUT(TIMER_4)) AND (NOT HRDW_FAULT) THEN
    BEGIN
        GET_TIME(TIME_DATE);
        WITH TIME_DATE DO
        BEGIN
            GOTOPAR(2); WRITE(MONTH:2,'/',DAY:2,'/',YEAR:4);
            GOTOPAR(4); WRITE(HOUR:2,':',MINUTE:2,':',SECOND:2);
        END;
    END;
END; { exec_wait }

```

{*****}

PROCEDURE EXEC_RUN;

{
 Executed when MENU = RUN.
}

BEGIN

 IF NOT PROBE_STOPPED THEN

 BEGIN

 CASE PROBE_AXIS OF

 X:

 BEGIN

 IF NEAR_LOC(ROD,2*LD) THEN

 BEGIN

 IF ROD <> 0 THEN

 BEGIN

 DATA_ACQ(ROW,ROD); { Take data. }

 ROD := ROD + 1; { Increment rod number. }

 IF ROD > LAST_ROD THEN

 BEGIN

 STOP_PROBE; { Stop probe. }

 ROD := 0; { Update probe motion variables. }

 GO_TO_LOC(ROD,X,0,0);

 END;

 END

 ELSE

 BEGIN

 STOP_AT_LOC(ROD,0); { Stop probe at stop. }

 ROW := ROW + 1; { Update probe motion variables. }

 IF ROW <= LAST_ROW THEN

 begin

 GO_TO_LOC(ROW,Y,0,0);

 end

 ELSE

 BEGIN { Start probe moving home. }

 ROW := 0;

 GO_TO_LOC(ROW,Y,0,0);

 END;

 END;

 END;

 END; { x }

 Y:

 BEGIN

 IF NEAR_LOC(ROW,0) THEN

 BEGIN

 STOP_AT_LOC(ROW,0);

 IF ROW <> 0 THEN


```

        BEGIN
            ROD := 1;
            GO_TO_LOC(ROD,X,0,0);
        END;
    END;
END; { y-dir }
END; { case }
END
ELSE
BEGIN
    { Update screen information. }
    CASE PROBE_STATUS OF
        STUCK: PRINT_PAR_STRING(1,'PROBE STUCK. ');
        HARD_STOP: PRINT_PAR_STRING(1,'PROBE STOPPED. ');
        ELSE PRINT_PAR_STRING(1,'RUN FINISHED. ');
    END; { case }
    PRINT_PAR_STRING(2,'DATA ACQUISITION STOPPED. ');
    UPDATE_SCREEN_PROBE;

    { Generate results from the run, and exit to the main menu. }
    TEXTCOLOR(WHITE+BLINK); MSG('Data taken is being processed. ');
    TEXTCOLOR(WHITE);
    GET_RESULTS; WRITE('G'); ANY_KEY;

    { Return to main menu. }
    CURSOROFF; MENU := WAIT; MAIN_MENU;
END; { if }

IF TIME_OUT(DISP_TIMER) THEN UPDATE_SCREEN_PROBE;
END; { exec_run }

{*****}

PROCEDURE EXEC_CAL;
{
    Executed when MENU = CAL.
}
VAR DRY_POWER,WET_POWER : REAL;
    START_POS,END_POS : REAL;

BEGIN
    IF NOT PROBE_STOPPED THEN
    BEGIN
        { Check to see if probe is near a rod. }
        IF NEAR_LOC(ROD,2*LD) THEN
        BEGIN
            CASE ROW OF
                1 :
            BEGIN

```

```

DATA_ACQ(ROW,ROD);
ROD := ROD + 1;
IF ROD > WET_ROD THEN
BEGIN { Stop motor and update screen parameters. }
  START_POS := SONY(X);
  LOAD_TIMER(TIMER_5,2);
  REPEAT UNTIL TIME_OUT(TIMER_5);
  END_POS := SONY(X);
  STOP_PROBE;
  DATA_PERIOD:=TRUNC((LD/(ABS(START_POS-END_POS)/2000))/MAX_DATA_PTS);
  DRY_POWER := POWER(FIRST_ROW,DRY_ROD);
  WET_POWER := POWER(FIRST_ROW,WET_ROD);
  GOTOPAR(1); CLREOL; WRITE(DRY_POWER:7:0);
  GOTOPAR(2); CLREOL; WRITE(WET_POWER:7:0);
  GOTOPAR(3); CLREOL; WRITE((WET_POWER/DRY_POWER):3:2);
  GOTOPAR(5); WRITE(DATA_PERIOD);
  PRINT_PAR_STRING(6,'CALIBRATION FINISHED. ');
  MSG(MSG_13); WRITE('G');
  CLEAR_PROBE_STATUS;
END;
END; { 1 }
2 :
BEGIN
  STOP_AT_LOC(ROD,0);
END; { 2 }
3 :
BEGIN
  STOP_AT_LOC(ROD,0);
  ROW := 1; ROD := DRY_ROD;
  GO_TO_LOC(ROD,X,0,0);
END; { 3 }
END; { case }
END; { if }
END
ELSE IF PROBE_STATUS <> CLEAR THEN
BEGIN { Update screen parameters. }
  CASE PROBE_STATUS OF
    STUCK: PRINT_PAR_STRING(6,'PROBE STUCK. ');
    HARD_STOP: PRINT_PAR_STRING(6,'PROBE STOPPED. ');
    AT_LOC:
      IF ROD = DRY_ROD THEN PRINT_PAR_STRING(6,'PROBE OVER DRY STANDARD. ')
      ELSE PRINT_PAR_STRING(6,'PROBE OVER WET STANDARD. ');
  END; { case }
  MSG(MSG_13); WRITE('G');
  CLEAR_PROBE_STATUS;
END; { if }

{ Put echo voltage on screen every DISP_PERIOD so the
{ screen isn't updated so often that it becomes difficult to read. }

```

```

IF TIME_OUT(DISF_TIMER) THEN
BEGIN
  GOTOPAR(4); WRITE(ABS(DTAD_CONV(DTAD_READ(ECHO_1,ERR_FLG))):3:1);
END;
END; { exec_cal }

{*****}

PROCEDURE EXEC_INIT;
{
  Executed when MENU = INIT.
}
BEGIN
END; { exec_init }

{*****}

PROCEDURE EXEC_RESULT;
{
  Executed when MENU = RESULT.
}
BEGIN
END; { exec_result }

{*****}

PROCEDURE EXEC_MNL;
{
  Executed when MENU = MNL.
}
VAR
  LOC : INTEGER; { X and Y positions of the cursor. }

BEGIN
  IF NOT PROBE_STOPPED THEN
  BEGIN
    CASE PROBE_AXIS OF
      X: LOC := ROD;           { The probe is moving, in search of a rod. }
      Y: LOC := ROW;
    END;

    { Check to see if probe is near LOC. }
    IF NEAR_LOC(LOC,0) THEN
    BEGIN { Probe has reached it's destination. }
      STOP_AT_LOC(LOC,0);
      MSG(MSG_13); WRITE('G');
    END;
  END
  ELSE IF PROBE_STATUS <> CLEAR THEN

```

```

BEGIN
CASE PROBE_STATUS OF
STUCK: PRINT_PAR_STRING(2,'PROBE STUCK.');
```

HARD_STOP: PRINT_PAR_STRING(2,'PROBE STOPPED.');

AT_LOC:

```

BEGIN
  IF PROBE_AXIS = X THEN
    BEGIN
      IF ROD = 0 THEN PRINT_PAR_STRING(2,'PROBE AT ZERO X STOP.')
```

ELSE IF ROD = LAST_ROD+1 THEN

PRINT_PAR_STRING(2,'PROBE AT END X STOP.')

ELSE PRINT_PAR_STRING(2,'PROBE AT ROD.');

```

    END
  ELSE
    BEGIN
      IF ROW = 0 THEN PRINT_PAR_STRING(2,'PROBE AT HOME Y STOP.')
```

ELSE IF ROW = LAST_ROW+1 THEN

PRINT_PAR_STRING(2,'PROBE AT END Y STOP.')

ELSE PRINT_PAR_STRING(2,'PROBE AT ROW.');

```

    END;
  END;
END;
MSG(MSG_13);
CLEAR_PROBE_STATUS;
END;

{ Put echo voltage and displacement on screen every DISP_PERIOD. }
IF TIME_OUT(DISP_TIMER) THEN
BEGIN
  GOTOPAR(1); WRITE(ABS(DTAD_CONV(DTAD_READ(ECHO_1,ERR_FLG))):3:1);
  GOTOPAR(3); WRITE(SONY(X):4:3);
  GOTOPAR(4); WRITE(SONY(Y):4:3);
  UPDATE_SCREEN_PROBE;
END;
END; { exec_mnl }

{*****}

PROCEDURE INIT_EXEC;

BEGIN
END;

```

```

{*****}
{                               INIT.INC                               }
{*****}
{

```

Description of this module:

This module contains variable initialization routines.

Each module has its own variable/hardware initialization routine which is called by INITIALIZE. INITIALIZE should be called at the very top of the main program. INIT_FROM_DISK is used to initialize certain variables from disk file INIT.VAR. SAVE_TO_DISK is used to save present variable default values to disk file INIT.VAR.

Procedures in this module:

```

PROCEDURE INITIALIZE;
PROCEDURE INIT_FROM_DISK;
PROCEDURE SAVE_TO_DISK;

```

Global variables used : many

```

}
{*****}

```

PROCEDURE INITIALIZE;

```

{
  Initializes all global variables as well as calling routines which
  initialize variables global to modules only.
}

```

BEGIN

```

  MWL_MOTOR_SPEED := 3;
  MENU := WAIT;      { Start off in WAIT menu. }

```

```

  DISP_TIMER := TIMER_2;

```

```

  TEXTMODE(C80); CURSOROFF;

```

```

  INIT_DT2801;
  INIT_TECHAR;
  INIT_SONY;
  INIT_PROBE;
  INIT_MENU;
  INIT_UTILITY;
  INIT_SCRPBB;
  INIT_DATA;
  INIT_KBD;
  INIT_EXEC;

```

```

END;

```

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

```
PROCEDURE INIT_FROM_DISK(VAR ERR_FLG : BOOLEAN);
```

```
{  
  Initializes all variables whose default values are stored on disc. If a  
  problem with the file INIT.VAR is encountered ERR_FLG is set to true.  
}
```

```
VAR
```

```
  VALID : BOOLEAN;
```

```
  INFILE : TEXT;
```

```
BEGIN
```

```
  ASSIGN(INFILE,'INIT.VAR');
```

```
  {$I-} RESET(INFILE); {$I+} VALID := (IORESULT = 0);
```

```
  IF NOT VALID THEN
```

```
    BEGIN
```

```
      WRITELN;
```

```
      WRITELN('No "INIT.VAR" file found.');
```

```
      WRITELN('This file is needed to initialize the data.');
```

```
      WRITELN('The program will not run without it.');
```

```
      ANY_KEY; CLRSCR;
```

```
      ERR_FLG := TRUE;
```

```
    END
```

```
  ELSE
```

```
    BEGIN
```

```
      READLN(INFILE,PLANT);
```

```
      READLN(INFILE,BUNDLE);
```

```
      READLN(INFILE,STOPT);
```

```
      READLN(INFILE,STOPY);
```

```
      READLN(INFILE,ERRI);
```

```
      READLN(INFILE,ERRY);
```

```
      READLN(INFILE,GAINX);
```

```
      READLN(INFILE,GAINY);
```

```
      READLN(INFILE,STALLX);
```

```
      READLN(INFILE,STALLY);
```

```
      READLN(INFILE,LD);
```

```
      READLN(INFILE,DISP_ROW1);
```

```
      READLN(INFILE,DISP_ROD1);
```

```
      READLN(INFILE,MIN_PRB_MOV);
```

```
      READLN(INFILE,MIN_DRY_POWER);
```

```
      READLN(INFILE,MAX_POWER_RATIO);
```

```
      READLN(INFILE,DISP_PERIOD);
```

```
      READLN(INFILE,STCL_PERIOD);
```

```
      READLN(INFILE,SAMPLE_PERIOD);
```

```
      READLN(INFILE,DEF_DRV);
```

```
      READLN(INFILE,DEF_DIR);
```

```
      READLN(INFILE,SAVE_RAW_DATA);
```

```
      CASE DEF_DRV OF
```

```

        'A': DRIVE := 1;
        'B': DRIVE := 2;
        'C': DRIVE := 3;
    END;
    DEF_DRV_DIR := CONCAT(DEF_DRV, ': ', DEF_DIR);
    ERR_FLG := FALSE;
    INIT_ROD_DISP;
END;
CLOSE(INFILE);
END;

{*****}

PROCEDURE SAVE_TO_DISK;
{
    Saves to disc all variables whose default values are stored on disc.
}
VAR
    INFILE : TEXT;

BEGIN
    ASSIGN(INFILE, 'INIT.VAR'); REWRITE(INFILE);
    WRITELN(INFILE, PLANT);
    WRITELN(INFILE, BUNDLE);
    WRITELN(INFILE, STOPX);
    WRITELN(INFILE, STOPY);
    WRITELN(INFILE, EPRX);
    WRITELN(INFILE, EPRY);
    WRITELN(INFILE, GAINX);
    WRITELN(INFILE, GAINY);
    WRITELN(INFILE, STALLX);
    WRITELN(INFILE, STALLY);
    WRITELN(INFILE, LD);
    WRITELN(INFILE, DISP_ROW1);
    WRITELN(INFILE, DISP_ROD1);
    WRITELN(INFILE, MIN_PRR_MOV);
    WRITELN(INFILE, MIN_DRY_POWER);
    WRITELN(INFILE, MAX_POWER_RATIO);
    WRITELN(INFILE, DISP_PERIOD);
    WRITELN(INFILE, STCK_PERIOD);
    WRITELN(INFILE, SAMPLE_PERIOD);
    WRITELN(INFILE, DEF_DRV);
    WRITELN(INFILE, DEF_DIR);
    WRITELN(INFILE, SAVE_RAW_DATA);
    CLOSE(INFILE);
END;

```

Appendix C
Listing of Manufacturers

List of Manufacturers

Compaq Computer :

Compaq Computer Corporation
P. O. Box 30
19515 FM 149
Houston, Texas 77070

Data Translation DT2801 board :

Data Translation, Inc.
100 Locke Drive
Marlborough, Ma. 01752-1192

Krautkramer Branson USIP-12 flaw detector :

Krautkramer Branson
Mifflin County Industrial Park
P.O. Box 350
Lewistown, Pa. 17044

Scientific Solutions TECMAR board :

Scientific Solutions
6225 Cochran Road
Solon, Ohio 44139

Sony Magnascale :

Dimensional Control Corporation
1403 Industrial Road
San Carlos, Ca. 94070

Turbo Pascal :

Borland International
4585 Scotts Valley Drive
Scotts Valley, Ca. 95066

124-A Motor Control :

Powr-Ups Corporation
50 Orville Drive
Bohemia, N. Y. 11716

Bibliography

- Light, Glenn M., et al. "Development of a Failed Fuel Pin Detection System." Technical paper, Society of Manufacturing Engineers, 1983. DIALOG file 8, item 13.
- Ogawa, S. Y., Lyons, M. F., Soulis, R. E. "Techniques and Equipment for Fuel Examinations at Power Reactor Sites." Conference paper, Conference on Remote System Technology, ANS winter meeting, November 1973. DIALOG file 8, item 28.
- Walton, Lewis A., et al. "Locating Leaking Fuel Rods in Light Water Reactors." Conference paper, Proceedings of the American Power Conference, 1985. DIALOG file 8, item 12.