STATE OF CALIFORNIA • BUSINESS, TRANSPORTATION & HOUSING AGENCY
DEPARTMENT OF TRANSPORTATION

TECHNICAL SERVICES BRANCH

HIGH SPEED PROFILING DEVICE

FINAL REPORT
DECEMBER 1994

FINAL REPORT #FHWA/CA/TL-95-05

CALTRANS STUDY #F90TL14

Prepared in cooperation with the United States Department of Transportation, Federal Highway Administration

STATE OF CALIFORNIA
DEPARTMENT OF TRANSPORTATION

ENGINEERING SERVICE CENTER
OFFICE OF MATERIALS ENGINEERING & TESTING SERVICES
TECHNICAL SERVICES BRANCH

HIGH SPEED PROFILING DEVICE

Report Number FHWA/CA/TL-95-05
Caltrans Study Number F90TL14

Supervised by ..... Rich Howell, P.E.

Principal Investigator ..... Walter A. Winter, P.E.

Co-Investigator ..... Greg A. Larson, P.E.

Report Prepared by ..... Greg A. Larson, P.E.

Walter A Winter
WALTER A. WINTER
Senior Materials and Research Engineer

Richard Howell
RICH HOWELL, Chief
Technical Services Branch

REGISTERED PROFESSIONAL ENGINEER
WALTER A. WINTER
No. 19272
EXPIRE
9/30/97
CIVIL
STATE OF CALIFORNIA

Technical Report Documentation Page

1. Report No. FHWA/CA/TL-95-05	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A High Speed Profiling Device		5. Report Date November 1994
		6. Performing Organization Code
7. Authors Greg A. Larson		8. Performing Organization Report No. 631155
9. Performing Organization Name and Address Division of New Technology, Materials and Research California Department of Transportation Sacramento, CA 95819		10. Work Unit No.
		11. Contract or Grant No. F90TL14
12. Sponsoring Agency Name and Address California Department of Transportation Sacramento, CA 95819		13. Type of Report and Period Covered Final
		14. Sponsoring Agency Code

15. Supplementary Notes

This project was performed in cooperation with the U.S. Department of Transportation, Federal Highway Administration.

16. Abstract

A profiling device (profilometer) capable of measuring pavement profiles at freeway traffic speeds was developed. This device can be used either to supplement or to replace the Response Type Road Roughness Meters (RTRRMs) that are typically part of a pavement management system. Since profilometers determine road roughness by measuring the pavement profile directly, they are not subject to the same stringent calibration requirements as RTRRMs. In addition, a profilometer's measurements are collected independent of vehicle speed, while an RTRRM is only accurate at the speeds for which it is calibrated.

The profilometer in this project uses laser height sensors to measure the distance from the vehicle to the pavement, while some other profilometers use ultra-sonic sensors or broad-band light sensors. Data from these laser sensors is collected by a VMEbus computer running MS-DOS. Software written in the C programming language reduces and processes the collected data into a pavement profile and an International Roughness Index (IRI) value. Additional navigation sensors provide vehicle location information during data acquisition.

17. Key Words Profilometer Response Type Road Roughness Meter International Roughness Index Pavement Profile Laser Height Sensor		18. Distribution Statement No restrictions. This document is available to the public through the National Technical Information Service, Springfield, VA 22161	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 200	22. Price

Form DOT F 1700.7 (8-72)

NOTICE

The contents of this report reflect the views of the Division of New Technology, Materials and Research which is responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California or the Federal Highway Administration. This report does not constitute a standard, specification, or regulation.

Neither the State of California nor the United States Government endorse products or manufacturers. Trade or manufacturers' names appear herein only because they are considered essential to the object of this document.

ACKNOWLEDGEMENTS

This research was sponsored by the United States Department of Transportation, Federal Highway Administration.

The Office of Electrical and Electronics Engineering, under the Division of New Technology, Materials and Research, would like to express their gratitude for the assistance and cooperation of the following people:

Office of Electrical and Electronics Engineering
- Edmund Ung, Associate Electronics Engineer
- Del Gans, Materials & Research Engineering Associate
- Bob Cullen, Materials & Research Engineering Associate
- Les Ballinger, Electrical Engineering Technician II
- Eric Jacobsen, Electrical Engineering Technician II
- Bruce Morehead, Student Assistant

Office of Structural Materials
- Bill Poroshin, Machine and Instrument Maker

Division of Maintenance, Office of Roadway Maintenance
- John Poppe, Pavement Condition Survey Supervisor
- Mark Woerner, Pavement Management Research Analyst
- Chuck Hoffman, Transportation Engineering Technician

GLOSSARY

3M	Minnesota Mining and Manufacturing Corporation
AC	Alternating Current
AC	Asphaltic Concrete
ADC	Analog-to-Digital Converter
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
BIOS	Basic Input/Output System
CPU	Central Processing Unit
CRT	Cathode Ray Tube
DC	Direct Current
DIO	Digital Input/Output
DOS	Disk Operating System
EL	Electro-Luminescent
FHWA	Federal Highway Administration
GPS	Global Positioning System
HPMS	Highway Performance Monitoring System
IBM	International Business Machines
IDE	Integrated Drive Electronics
IRI	International Roughness Index
LCD	Liquid Crystal Display
LED	Light Emitting Diode
MS	MicroSoft Corporation
NMEA	National Marine Electronics Association
PC	Personal Computer
PCC	Portland Cement Concrete
PVC	Poly Vinyl Chloride
RAM	Random Access Memory
RMS	Root Mean Square
RTRRM	Response Type Road Roughness Meter
TTL	Transistor-Transistor Logic
VGA	Video Graphics Array

```
        case 1 :                                    // check for second column
          gotoxy(21,linenum);                       // move cursor to second column
          break;                                    // break
        case 2 :                                    // check for third column
          gotoxy(41,linenum);                       // move cursor to third column
          break;                                    // break
        case 3 :                                    // check for fourth column
          gotoxy(61,linenum);                       // move cursor to fourth column
        }                                           // end of switch
    strcpy(buffer, path);                           // install path in buffer
    strcat(buffer, file_name[i]);                   // add file name to path
    fileptr = fopen(buffer, "rb");                  // open file for reading
    fread(&processed, 1, 1, fileptr);               // get file processed flag
    fclose(fileptr);                                // close file
    if(processed)                                   // check for file processed
        printf("* %s\n", file_name[i]);             // print processed file name
    else                                            // un-processed file
        printf("  %s\n", file_name[i]);             // print un-processed file name
    }                                               // end of for
gotoxy(1,25);                                       // move cursor to last line
printf("Move the highlight bar to the desired file name and press return...");
//
// highlight the first file name
//
gotoxy(3,3);                                        // move cursor to line 3
textattr(INVERSE);                                  // set to inverse video
cprintf("%s", file_name[0]);                        // re-print the first file name
gotoxy(3,3);                                        // move cursor to line 3
linenum = 3;                                        // initialize line number
//
// highlight each file name as the cursor moves over it
//
i = 0;                                              // initialize file name index
while((ch = getch()) != 0x0D) {                     // wait for a <CR>
    if(ch == 0x1B) {                                // check for escape
        textattr(NORMAL);                           // set to normal video
        return(1);                                  // say escape pressed
        }                                           // end of if
    if(ch == 0) {                                   // check for arrow key
        ch = getch();                               // get second character
        if((ch == 0x48) || (ch == 0x50)) {          // check for up or down arrow
          textattr(NORMAL);                         // set to normal video
          cprintf("%s", file_name[i]);              // re-print current line

          if(ch == 0x48) {                          // check for up arrow
              if(linenum == 3) linenum = 23;        // top line, wrap around
              else linenum--;                       // otherwise, move up one line

              if(i == 0) {                          // check for first file
                i = filecnt - 1;                    // go to last file
                linenum = filecnt%21 + 2;           // calculate line number
                if(filecnt%21 == 0)                 // check for multiple of 21
                      linenum = 23;                 // move cursor to last line
                }                                   // end of if
```

3.3.2 Tape Backup Unit 9

3.3.3 Computer Chassis 10

3.3.3.1 Central Processing Unit Circuit Card 10

3.3.3.2 Disk Drive Module 10

3.3.3.3 Digital Input/Output Circuit Card 10

3.3.3.4 Analog-to-Digital Converter Circuit Card 11

3.3.3.5 Signal Conditioning Circuit Card 11

3.3.3.6 Optocator Interface Circuit Cards 11

3.3.3.7 Direct Current to Direct Current Power Supply 13

3.3.3.8 Alternating Current to Direct Current Power Supply 13

3.3.3.9 Power Source Selection Circuitry 13

3.4 Computer Keyboard 13

3.5 Video Display 13

3.6 Navigation Sensors 14

3.6.1 Wheel Pulse Sensors 14

3.6.2 Magnetic Flux Gate Compass 14

3.6.3 Solid-State Angular Rate Sensor 14

3.6.4 Loop Excitation Power Monitor 14

3.7 Power System 15

4. SOFTWARE DESCRIPTION 17

4.1 Profile Van Computer Software 17

4.1.1 Background Routines 17

4.1.1.1 DISTISR.ASM 17

4.1.1.2 TIMEISR.ASM 19

4.1.2 Foreground Routines 19

4.1.2.1 EXEC.C 19

4.1.2.2 DATA_ACQ.C 20

4.1.2.3 DISP_VEL.C 20

4.1.2.4 DATA_PRO.C 20

4.1.2.4.1 RAW.C 21

4.1.2.4.2 SLOPE.C 21

4.1.2.4.2.1 INIT_STM.C 21

4.1.2.4.3 CNVNAV.C 21

4.1.2.5 SYS_TEST.C 21

4.1.2.5.1 BOUNCE.C 22

4.1.2.5.2 WHEEL.C 22

4.1.2.5.3 STEP.C 23

4.1.2.5.4 HEIGHT.C 23

4.1.2.5.5 ADC.C 23

4.1.3 Utility Routines 23

4.1.3.1 GPS.C 24

4.1.3.2 RS232.C 24

4.1.3.3 MOV_CURS.C 24

4.1.3.4 GETMSG.C 24

4.2 Laptop Computer Software 25

4.2.1 SERIAL.C 25

4.2.2 PROCESS.C 25

5. DATA FORMAT 29

5.1 Program and Data Files 29

5.2 RAW Sub-directory 29

5.2.1 TEST.SET 30

5.2.2 TEST.RAW 30

5.2.3 TEST.NAV 30

5.2.4 TEST.GPS 31

5.2.5 TEST.KEY 31

5.3 DATA Sub-directory 32

5.4 SLOPE Sub-directory 32

5.5 ELEV Sub-directory 33

5.6 IRI Sub-directory 33

6. FIELD TESTING 35

6.1 Yolo County Test Site 35

6.2 Nevada Test Sites 36

6.2.1 Site One 37

6.2.2 Site Two 37

6.2.3 Site Three 38

6.2.4 Site Four 38

6.2.5 Site Five 39

6.2.6 Site Six 39

7. CONCLUSIONS 41

8. RECOMMENDATIONS 43

9. IMPLEMENTATION 45

10. REFERENCES 47

APPENDICES

A. Engineering Data A-1

Photographs A-1

Signal Conditioning Circuit Card Schematics A-8

Power Source Selector A-12

Interconnect Wiring Diagram A-13

B. Software Listings B-1

DISTISR.ASM B-2

TIMEISR.ASM B-7

EXEC.C B-11

DATA_ACQ.C B-17

DISP_VEL.C B-29

DATA_PRO.C B-33

RAW.C B-42

SLOPE.C B-49

INIT_STM.C B-54

CNVNAV.C B-57

SYS_TEST.C B-59

BOUNCE.C B-61

WHEEL.C B-65

STEP.C B-70

HEIGHT.C B-72

ADC.C B-74

GPS.C B-76

RS232.C B-82

MOV_CURS.C B-85

GETMSG.C B-86

SERIAL.C B-88

PROCESS.C B-95

C. User's Manual C-1

D. Data Record Formats D-1

Setup Data File (TEST.SET) D-1

Keyboard Data File (TEST.KEY) D-2

Partial GPS Data File (TEST.GPS) D-3

Partial Navigation Data File (TEST.NAV) D-4

Partial Processed Navigation Data File (TEST.PRN) D-5

Partial Raw Data File (TEST.RAW) D-6

Converted Data Files (TEST.VEL, TESTL.ACC, TESTL.HGT) D-7

Partial Slope and Elevation Profile Data (TESTL.SLP, TESTL.ELV) D-8

Partial IRI Data File (TESTL.IRI) D-9

E. Vendor Addresses E-1

FIGURES

Figure	Title	Page
1	Hardware Block Diagram	8
2	Signal Conditioning Circuit Card Block Diagram	12
3	Power System Block Diagram	16
4	Van Computer Software Block Diagram	18
5	Laptop Computer Software Block Diagram	26
A1	Profile Van, front view	A-1
A2	Profile Van, rear view	A-2
A3	Bumper, right wheel path with cover plate removed	A-2
A4	Laser Probe and Accelerometer	A-3
A5	Probe Processing Unit	A-3
A6	Equipment Console	A-4
A7	Computer Chassis, front cover open	A-4
A8	Laptop Computer and Keyboard	A-5
A9	Color VGA Monitor	A-5
A10	Five Channel GPS Receiver with Antenna	A-6
A11	Magnetic Flux Gate Compass	A-6
A12	Loop Excitation Power Monitor, installed	A-7
A13	Loop Excitation Power Monitor with cover removed	A-7
A14	Signal Conditioner Schematic Diagram	A-8

A15 Power Source Selector Schematic Diagram A-12

A16 Interconnect Wiring Diagram A-13

1. INTRODUCTION

1.1 Problem

The state of California's Department of Transportation (Caltrans) currently uses a Response Type Road Roughness Meter (RTRRM) to obtain roughness data on its highways. Some of the roughness data is reported to the Federal Highway Administration (FHWA) as part of their Highway Performance Monitoring System (HPMS) program. The FHWA uses this data to determine how federal highway funds are to be distributed to the states. They have recently revised the calibration and data collection procedures for the equipment that collects HPMS data. The new procedures mandate a rigorous set of calibration requirements for RTRRM devices. Satisfying these requirements would be prohibitively expensive for Caltrans.

An alternative is to use a high speed profilometer, which is a device that directly measures the longitudinal pavement profile at highway speeds and uses it to determine road roughness by one of several methods. Since it measures the profile directly, a profilometer is not subject to the same stringent calibration requirements that are necessary with an RTRRM. This project was undertaken to develop a profilometer that would comply with the FHWA requirements.

1.2 Objective

The objective of this project was to develop a profilometer based on the PRORUT system that was built for the FHWA by the University of Michigan Transportation Research Institute (1). The PRORUT system consists of a specially equipped van that is capable of measuring longitudinal pavement profile and rut depth while traveling at highway speeds. The van's data acquisition and control system is based on an IBM Personal Computer with a 3M cartridge tape drive. Software aids in the collection, processing, and viewing of the data.

The PRORUT system was designed in the early 1980s using equipment that was then the state-of-the-art. For this project, however, the system was re-designed to take advantage of recent improvements in computer and sensor technology. In addition, the PRORUT software, which was written in the FORTRAN programming language, was re-written using the currently more popular C programming language. Navigational sensors and software were also added to the design to reduce the amount of work performed by the operator. The system developed under this project will be used by Caltrans to collect roughness data from HPMS test locations and to measure ride quality for the remainder of the state highway system.

1.3 Background and Significance of Work

The RTRRMs produce roughness data by accumulating measurements of the movement of the rear axle of an automobile relative to its body. The measurements are heavily affected by variables such as fuel level, wheel alignment, tire balance and tire wear, shock absorber characteristics and leaf spring performance, and disturbances caused by gusty winds. All of these variables are beyond the control of the operator, so it is difficult to obtain repeatability in the measurements from RTRRMs.

The RTRRMs report road roughness as a "ride score", which is a weighted accumulation of axle displacements resulting in a single numeric value in the range between 0 (perfectly smooth) and 70 (impassable). Before being used for data collection, the RTRRMs are calibrated at test sites of known ride score to correlate their output with the ride score table. Using this method, Caltrans has built a historical data base of ride scores dating back to 1978 that covers approximately 77,000 lane-kilometers (48,000 lane-miles) of pavement.

To determine the status of the pavement on its highways, Caltrans performs a "pavement condition survey" every two years (2). The first part of the survey consists of a visual inspection by trained pavement raters who look for and document signs of pavement distress. The second part consists of collecting ride scores for every lane-mile of highway. The manual ratings and ride scores are incorporated into a data base that tracks the condition of the pavement sections as they age. Although this survey is performed independently of the HPMS data collection, there are many locations where the two overlap. In the past, Caltrans has reported the ride scores collected during the survey to the FHWA for these overlapping sites. Ride scores for the remaining HPMS sites were then collected separately.

Now, however, the FHWA is requiring that all HPMS data be reported in the form of an International Roughness Index, or IRI. The IRI is computed by solving the mathematical equations for a quarter car simulation using the measured longitudinal profile as the input to the simulation (3). A quarter car simulation models a single wheel of an automobile using a system of masses, springs, and dampers. The vertical movement of the sprung mass, which simulates the driver and passenger seating area, is accumulated over a distance traveled and is reported as meters of displacement per traveled kilometer (inches/mile), with higher IRI values indicating rougher roads.

Since RTRRM devices do not measure longitudinal profile, the only way that they can generate IRI values is to somehow correlate the ride score to an IRI. This correlation can be accomplished by following the calibration and testing procedures for RTRRM devices that are outlined in Appendix J of the HPMS Field Manual (FHWA Order M 5600.1A "Roughness Equipment, Calibration and Data Collection"). The appendix was revised in December 1987 to include the new procedures for using RTRRMs. The calibration

require three days per RTRRM. In a state as large as California, where the RTRRMs might be far away from the calibration sites, the logistics problems associated with returning the equipment to be re-calibrated for several days every month are formidable.

To avoid these calibration problems, Caltrans chose to replace the RTRRMs with a profilometer. Profilometers are available from several commercial sources, but the cost to purchase one with the necessary specifications was found to be too high. The engineering data and software listings for the PRORUT system were available, so Caltrans decided to build its own profilometer based on the PRORUT.

1.4 Scope

This project was conceptualized to develop a profilometer that would generate IRI values in real time as the system traveled down the highway. However, initial investigation revealed that it would be very difficult to accomplish this goal within the time and funding constraints of the project. Therefore, the decision was made to simply collect the raw data in real time and process it off-line later (post-mission processing).

The software was initially written to collect one sample of raw profile data for every 25.4 mm (one inch) of vehicle travel, which resulted in huge amounts of data having to be stored, even for very short runs. The problem of handling this data became an issue early in the system field test phase of the project. It was then decided to complete the 25.4 mm (one inch) sampling software first and then develop another version that would collect one sample for every 305 mm (one foot) of travel. A 305 mm (one foot) sampling interval is common among commercially available profilometers since it enables them to sense roughness in the band of frequencies that affect ride quality without having to process excessive amounts of data.

In addition to collecting profile data, the profilometer acquires data from a variety of navigation sensors. Building on the work that was performed for another FHWA project, the profilometer collects data from a flux gate magnetic compass and a solid-state angular rate sensor (4). It also receives accurate vehicle location messages from a five channel Global Positioning System (GPS) receiver. A loop excitation power monitor detects the presence of inductive loops as the van passes over them. All this data can be post-mission processed to provide a "track" of where the vehicle was when the IRIs were collected.

2. TECHNOLOGY OPTIONS

The basic principle behind longitudinal pavement profiling is to establish an inertial reference coordinate system in the vehicle and make height measurements relative to it. To create the inertial reference, an accelerometer, whose sensitive axis is in the vertical direction, is mounted near the height sensor. The up and down acceleration of the vehicle is measured, processed, and used to correct the height sensor data and give true profile. Vehicle velocity is also needed by the profile processing algorithms.

The choice of height measuring sensor technology is the key to making reliable profile measurements. Several alternatives are available for height sensing depending on accuracy, sample rate, ease of maintenance, and sensitivity to surface type. Early in the project, a contract was awarded to the Department of Electrical & Electronic Engineering at the California State University, Sacramento, to investigate the best height sensor technology to use in this application. Their recommendation was to use the laser-based Optocator height sensor manufactured by Selcom AB of Sweden (5). The four most common methods for making height measurements in profilometers are presented below.

2.1 Mechanical Follower-Wheels

The early General Motors profilometers measured height using a spring loaded wheel which rolled along the surface of the road and was attached to a potentiometer. As the wheel moved up and down with the changes in elevation the resistance from the potentiometer would change linearly to provide the height measurement. This method suffered from wheel bounce and required excessive maintenance (6). Contact type sensors have now been replaced by the non-contact technologies discussed below.

2.2 Ultrasonic Height Sensors

Another method for determining height in profilometers is to measure the length of time it takes for an ultrasound pulse to travel from the vehicle to the pavement and back again. Since the speed of sound is relatively constant in air, the distance traveled by the pulse can be calculated from the elapsed time. Ultrasonic height sensing experiences difficulties on some pavement surfaces that do not reflect sound adequately. In addition, the speed of sound in air does vary with temperature, pressure, and humidity, which can create errors in the height measurements (6). Air turbulence is another source of error.

2.3 Optical Height Sensors

Two primary optical height sensing methods are used in profilometers. The K. J. Law Model 690-DNC profilometer uses a sensor with two incandescent light sources. One source generates a light spot on the pavement and one generates a reference signal. The reflected light spot and the reference signal strike a rotating mirror and are detected by a

receiver. The position in time of the reflected light spot between two consecutive reference signals is proportional to the distance from the sensor to the pavement. This system is popular and works well for a variety of surface textures and colors.

The second optical height sensing method uses an infrared light source. While under contract with the FHWA, Southwest Research Institute designed and built a height sensor that has an infrared light emitting diode as its light source. During laboratory and field testing, it was discovered that this sensor suffered from sensitivity to changes in surface reflectivity (6).

2.4 Laser Height Sensors

The Selcom Optocator height sensors are used almost exclusively by the profilometers that have laser light sources. They offer excellent resolution and accuracy, low maintenance, and high reliability. Selcom manufactures a wide variety of laser sensors for different applications, but the sensors used on this project were specifically designed for making road roughness measurements. They have several options added to improve their performance, such as smaller spot size, increased immunity to ambient light, and faster sampling rate. They also have special control circuitry that adjusts the output power of the laser to maintain a constant received light intensity regardless of changes in the surface reflectivity. This feature is essential to a profilometer since it must be capable of collecting data on different pavement types, compositions, textures, and colors.

3. HARDWARE DESCRIPTION

The hardware for the profile van is primarily commercial, off-the-shelf equipment. The laser height sensors and accelerometers are required to meet military standards for shock, vibration, and temperature. The remainder of the hardware is rated for industrial-type environmental conditions.

A block diagram of the profile van hardware is shown in Figure 1. The interconnect wiring diagram for the system and schematics for the signal conditioning circuit card and the power source selector are shown in Appendix A.

3.1 Vehicle

The profile equipment is installed in a 1989 Chevrolet 3/4 ton cargo van. Air bags were installed under the rear axle to reduce the sway during driving. Independent left and right air shock absorbers were added to the front end to enable the operator to adjust the height of the laser sensors to the center of their measurement range.

3.2 Bumper

The existing van bumper has been cut in half lengthwise and re-installed. A sealed, hollow, rectangular bumper fabricated from 6.4 mm (1/4") thick 6061T6 aluminum was designed and mounted beneath the original bumper. The laser height sensors for the left and right wheelpath are mounted in each end of the bumper so that their measurement points are 1.65 m (65") apart. The bumper has a pair of small holes in the bottom surface of each end for the laser light to exit and the reflected light to re-enter. Accelerometers for the left and right wheelpath are installed next to the laser sensors with their sensitive axis perpendicular to the pavement.

3.2.1 Laser Height Sensors

The laser height sensors are Selcom model number 2008-501-I Optocators with a 390 mm (15.4") standoff height and a 128 mm (about 5") measurement range. The standoff height is the distance from the face of the laser source to the center of the measurement range. The sensors are mounted so that their nominal height above the pavement is at the center of their measurement range. The operator can adjust the front air shock absorbers to maximize the dynamic range of the sensors.

The sensor consists of two major assemblies: a probe and a probe processing unit. The probe contains an infrared laser diode transmitter and a linear position sensitive photodetector array. The array is offset from the transmitter and tilted at an angle pointing toward the spot on the pavement where the laser light will be reflected. The

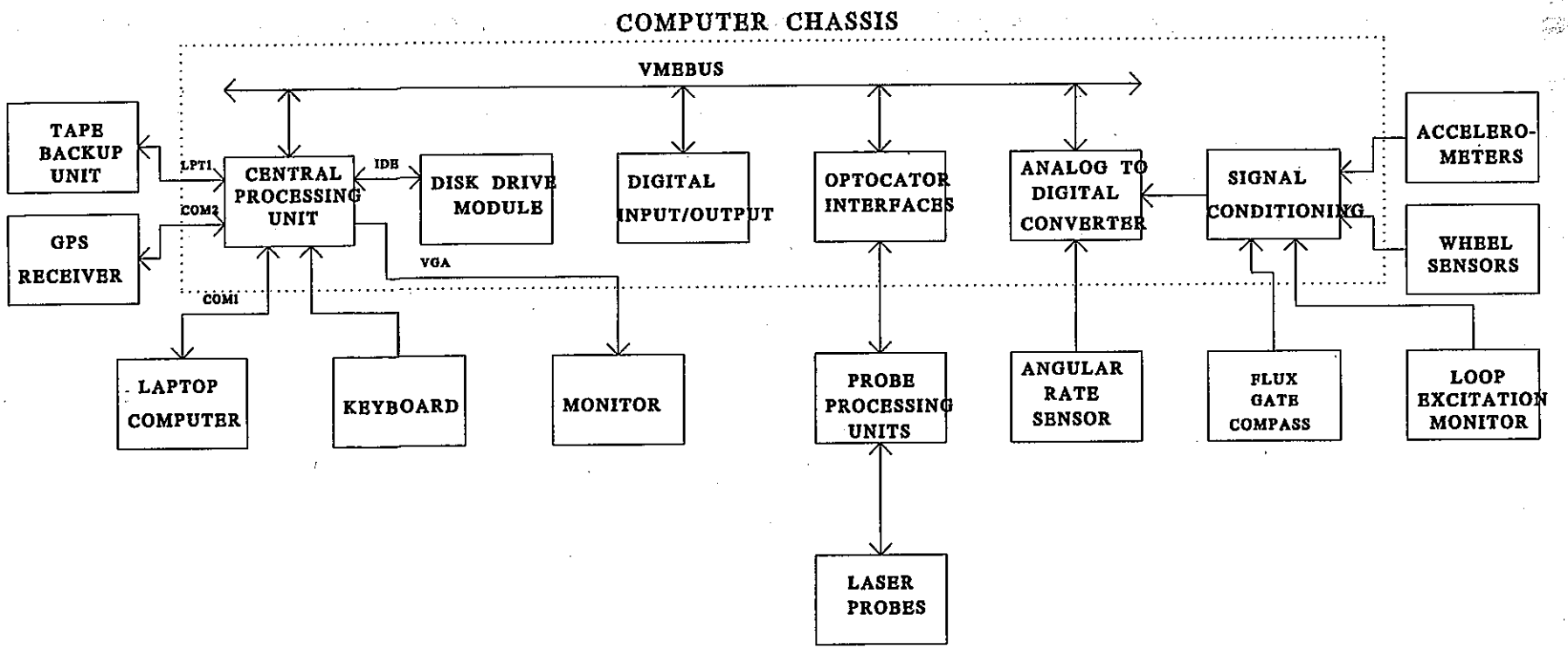


Figure 1 Hardware Block Diagram

height is determined by triangulation, with the element in the array where the reflected light falls being proportional to the distance from the probe to the pavement.

The probe is connected by a cable to the probe processing unit. The probe processing unit contains the analog and digital circuitry for processing and converting the signal from the probe into a digital height value. This height value is sent to the Optocator interface circuit card in the computer chassis as a fifteen bit serial data stream. Twelve of the data bits contain height information, giving a resolution of 31.25 micrometers. The three remaining bits indicate if an error was detected in the measurement. An error is detected if the reflected light is outside the field of view of the receiver array, such as when the light spot falls into a pavement crack.

3.2.2 Accelerometers

The accelerometers are Lucas Schaevitz model LSBC-2 with a ± 2 g range. They have a one g bias option installed to compensate for the Earth's static gravity force. They produce 2.5 volts of output per 1 g of input. The output signal from each accelerometer passes through a lowpass filter with a cut-off frequency of 20 Hz on the signal conditioning circuit card.

3.3 Equipment Console

The equipment console consists of a Type 38 cabinet made by Elma Electronics. The dimensions of the console are 530 mm wide by 700 mm high by 520 mm deep. It is shock mounted to the floor of the van just behind the driver's seat with Aeroflex cable-type shock and vibration isolators. The computer chassis is slide rack mounted in the top half of the console. The tape backup unit is installed just below the computer chassis and the GPS receiver module is mounted on the bottom plate of the console.

3.3.1 GPS Receiver Module

The GPS receiver module is a five channel original equipment manufacturer unit that was built by Magellan Systems Corporation. The receiver sends formatted position messages to the computer through an RS-232C link that is connected to a serial port (COM2) on the computer chassis. It provides absolute vehicle location with an accuracy of about ± 25 meters. The antenna is mounted on the roof of the van.

3.3.2 Tape Backup Unit

The tape backup unit is a Colorado Memory Systems Jumbo Trakker with up to 250 Megabytes of storage per tape cartridge. The tape unit plugs into the centronics parallel port (LPT1) on the computer chassis and can transfer up to one Megabyte of data per second.

3.3.3 Computer Chassis

The computer chassis is a Type 12 VMEbus rack mount kit manufactured by Elma Electronics. It contains a 6U high VMEbus computer backplane capable of handling up to ten VMEbus form factor circuit cards. Behind the backplane is the electrical power source selection circuitry that automatically chooses which power source to use for the computer. A power supply for converting the 12 volt power in the van to the 24 volts that is required by the Optocator interface circuit cards is located there as well. The back panel of the chassis has connectors mounted on it for routing the sensor signals to the computer. When all the sensor signal cables are disconnected from the back panel, the entire chassis can be slid out of the equipment console for troubleshooting.

3.3.3.1 Central Processing Unit (CPU) Circuit Card

The CPU circuit card is a model XVME-686 manufactured by Xycom. It is the equivalent of a desktop IBM PC/AT compatible computer but is built to a VMEbus form factor. The microprocessor is a 20 megahertz 80386SX along with an 80387SX math coprocessor. It has built-in support for Video Graphics Array (VGA) displays and Integrated Drive Electronics (IDE) disk drives. The card also has a keyboard input port, two RS-232C serial ports (COM1 and COM2), and a centronics parallel port (LPT1). It uses the MS-DOS version 6.20 operating system and has four megabytes of memory. A VMEbus controller interface is built into the card which enables it to communicate with the other circuit cards that are plugged into the VMEbus backplane.

3.3.3.2 Disk Drive Module

The disk drive module is a model XVME-955 manufactured by Xycom. It contains a Teac floppy disk drive with a capacity of 1.44 megabytes and a Quantum model LPS525A hard disk drive with a capacity of 525 megabytes. It occupies two slots of the VMEbus backplane and is plugged in next to the CPU card. Cables are connected between the module and the IDE interface on the CPU card. The profile software and all the data collected during a data acquisition run are stored on the hard disk drive.

3.3.3.3 Digital Input/Output (DIO) Circuit Card

The DIO circuit card is a model XVME-290 from Xycom. It contains two sixteen bit parallel ports that can be programmed to be either inputs or outputs. One of these ports is programmed to output a signal to the signal conditioning circuit card that resets the interrupt request from the wheel pulse sensors. The other port is used to input the status of some digital signal lines such as the wheel pulse sensors. The card also has two 24 bit timers, one of which is programmed to generate an interrupt to the CPU card every millisecond. This interrupt is used to keep track of elapsed time and to control the

acquisition of the navigation data. The other timer is setup to measure the van's velocity. The CPU card can read data from or write data to the DIO card through the VMEbus interface.

3.3.3.4 Analog-to-Digital Converter (ADC) Circuit Card

The ADC circuit card is a model XVME-590/3 from Xycom. It contains a twelve bit analog-to-digital converter chip with a ten microsecond conversion time. A multiplexer routes any one of sixteen single-ended analog inputs to the chip through a programmable gain amplifier. The full-scale input voltage range at a gain of one is ± 10 volts. The CPU card can read data from the ADC card through the VMEbus interface. Eight channels on the ADC card are used by the profile system, as are shown below:

Channel 0	left wheelpath accelerometer signal
1	right wheelpath accelerometer signal
2	magnetic flux gate compass x-axis signal
3	magnetic flux gate compass y-axis signal
4	loop excitation monitor power signal
5	solid state angular rate sensor signal
6	zero volts (for diagnostic purposes)
7	five volts (for diagnostic purposes)

3.3.3.5 Signal Conditioning Circuit Card

The signal conditioning circuit card was designed and built specifically for this project. It serves as an interface between most of the sensors and the ADC card. It converts the wheel pulse sensor signals to TTL level so that the CPU card can read them. The card also has frequency-to-voltage converter chips for changing the wheel pulse signals into a DC voltage proportional to the frequency, which is used to detect a change of heading. Lowpass filters eliminate noise from the x-axis and y-axis signal lines in the magnetic flux gate compass and converts the inductive loop excitation monitor signal into a voltage that is proportional to its peak power. It contains five-pole lowpass Bessel filters with 20 hertz cutoff frequencies for the left and right wheelpath accelerometers. This card gets its electrical power and ground signals from the VMEbus backplane, but does not have a VMEbus interface. A block diagram of the signal conditioning circuit card is shown in Figure 2.

3.3.3.6 Optocator Interface Circuit Cards

The two (one for each wheelpath laser sensor) Optocator interface circuit cards are manufactured by Selcom AB. They read in the serial data streams from the probe processing units and convert them to a parallel format. These cards also have the ability to average a number of samples from the laser sensors to produce a height value that is

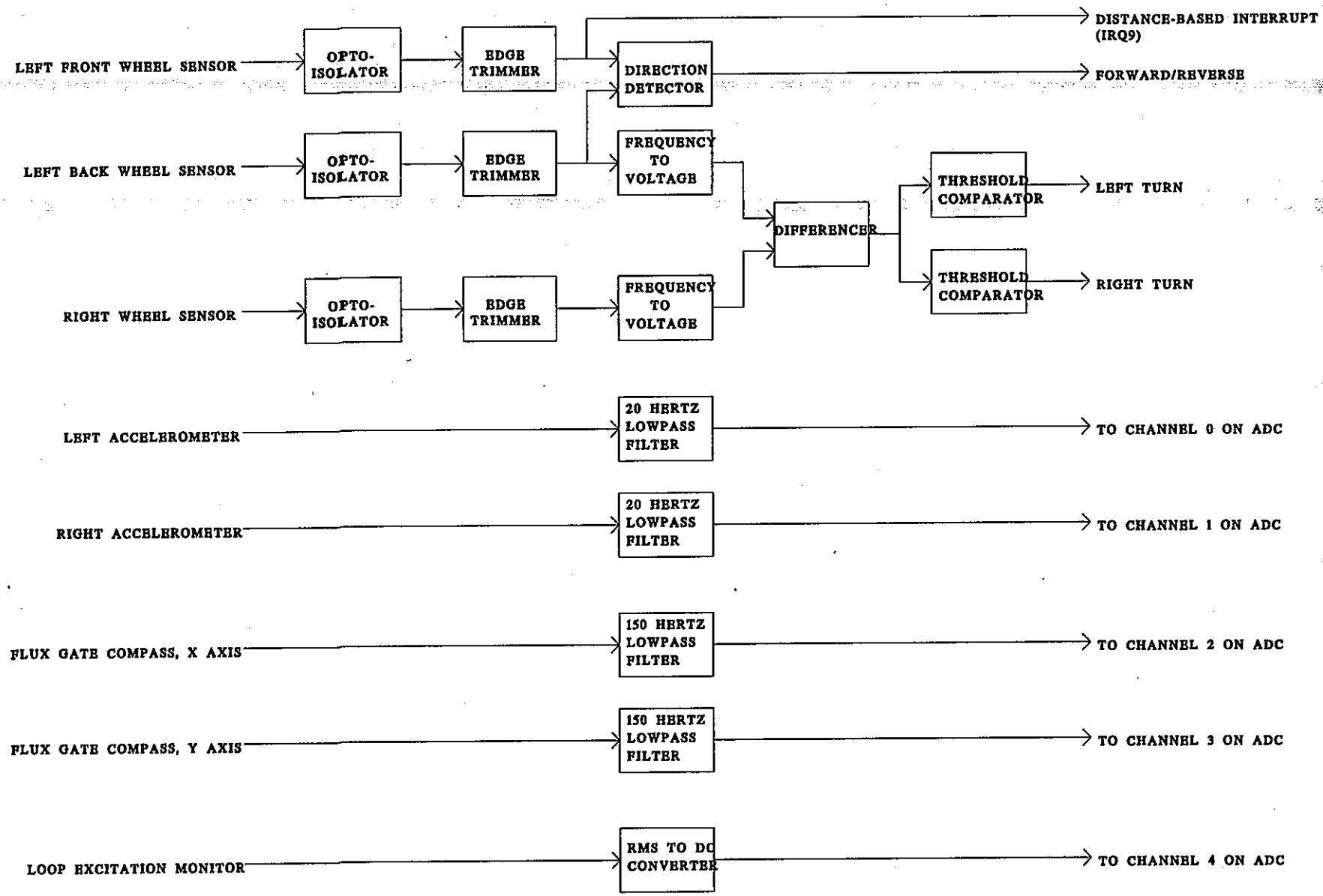


Figure 2 Signal Conditioning Circuit Card Block Diagram

less noisy and less sensitive to errors caused by the light beam falling into pavement cracks. They are programmed to average 32 samples, which results in an updated height value every one millisecond. The CPU card can read the height data from the Optocator interface cards through the VMEbus interface.

3.3.3.7 Direct Current to Direct Current (DC/DC) Power Supply

The DC/DC power supply is a model GK 120 manufactured by BICC-VERO Electronics. It converts the nominal 12 volt DC power from the computer battery and charging system to the + 5 volts and ± 12 volts used by the VMEbus backplane. It has input and output current limiting and overvoltage protection.

3.3.3.8 Alternating Current to Direct Current Power Supply

The alternating current (AC) to direct current power supply is a model PK 200 manufactured by BICC-VERO Electronics. When the van is parked and has access to a standard electrical power outlet (nominally 120 Volts AC), an extension cord can be connected between the AC socket on the side of the van and the outlet to power the profile equipment, which saves the charge on the computer battery. This power supply converts the AC power to 12 volt DC power used by the DC/DC power supply, thereby replacing the auxiliary battery and charging system.

3.3.3.9 Power Source Selection Circuitry

The power source selection circuitry was designed and built specifically for this project and contains high current relays and switches. It automatically selects AC power for the profile equipment if it is available, disconnecting the computer battery. If AC power is not available, it automatically selects the computer battery power. It was designed to enable the operator to switch from battery power to AC power without affecting the operation of the computer, but the transfer of power sources tends to reset the computer.

3.4 Computer Keyboard

The computer keyboard is a Marquardt MiniBoard Model 7065 with 82 keys and overlays to simulate a 101 key enhanced keyboard.

3.5 Video Display

The video display is a Magnavox CM2089 color monitor with a 356 mm (14") diagonal screen and VGA resolution. It is mounted on a swiveling pedestal just behind the front passenger seat.

3.6 Navigation Sensors

The navigation sensors provide vehicle track data based on the principle of dead reckoning. The magnetic flux gate compass and the solid state angular rate sensor give vehicle heading data at one second intervals. The wheel pulse sensors indicate the distance traveled along that heading.

3.6.1 Wheel Pulse Sensors

The wheel pulse sensors are Model DZ350SLE magnetic zero speed sensors manufactured by Electro Corporation. The sensors are mounted to the disk rotor guard plate on both front wheels, with two on the left and one the right. An 86 tooth gear is press-fit into the inside rim of each front wheel. The sensors detect the presence of teeth as they pass and generate pulses. The gear tooth spacing is designed to create a pulse for every 25.4 mm (1″) of travel with a duty cycle of approximately 50%. The spacing between the two left wheel sensors is adjusted to achieve a 90 degree phase difference (quadrature) between the output pulse trains. Circuitry on the signal conditioning card can determine the direction of travel (forward or reverse) by comparing these two pulse trains. In addition, the pulse trains from the left and right wheel are compared to sense turning maneuvers.

3.6.2 Magnetic Flux Gate Compass

The magnetic flux gate compass is a Model 02-0022 made by Etak. It is mounted in the center of the van above the headliner. It senses vehicle heading by measuring the component of the Earth's magnetic field in two perpendicular axes. The x-axis and y-axis outputs are lowpass filtered by the signal conditioning card and sent to channels 2 and 3, respectively, on the ADC card.

3.6.3 Solid State Angular Rate Sensor

The solid state angular rate sensor is a Systron Donner model QRS11-00050-200. It is installed on a cross beam underneath the floor in the center of the van. It has a measurement range of ± 50 degrees per second. The sensor is mounted so that its sensitive axis is in the yaw plane of the vehicle. Its output is connected to channel 5 on the ADC card.

3.6.4 Loop Excitation Power Monitor

The loop excitation power monitor is a model 631150-310 designed and built by Caltrans. It was developed for use in a research project investigating vehicle navigation. The monitor is mounted below the rear bumper and senses the loop excitation signal as the van passes over the vehicle detector loop. This AC signal is sent through a Root-

Mean-Squared (RMS) to DC converter on the signal conditioning card to generate an output voltage which is proportional to the peak amplitude of the excitation signal. The output of the converter is routed to channel 4 on the ADC card.

3.7 Power System

The power system was designed and built specifically for this project. It consists of a heavy-duty alternator connected to both the van battery in the engine compartment and the auxiliary battery used by the computer in the cargo area of the van. A continuous duty relay isolates the auxiliary battery from the rest of the van electrical system when the driver starts the engine. This isolation protects the computer and the sensors from the voltage drop caused by the engine starting. A block diagram of the power system is shown in Figure 3.

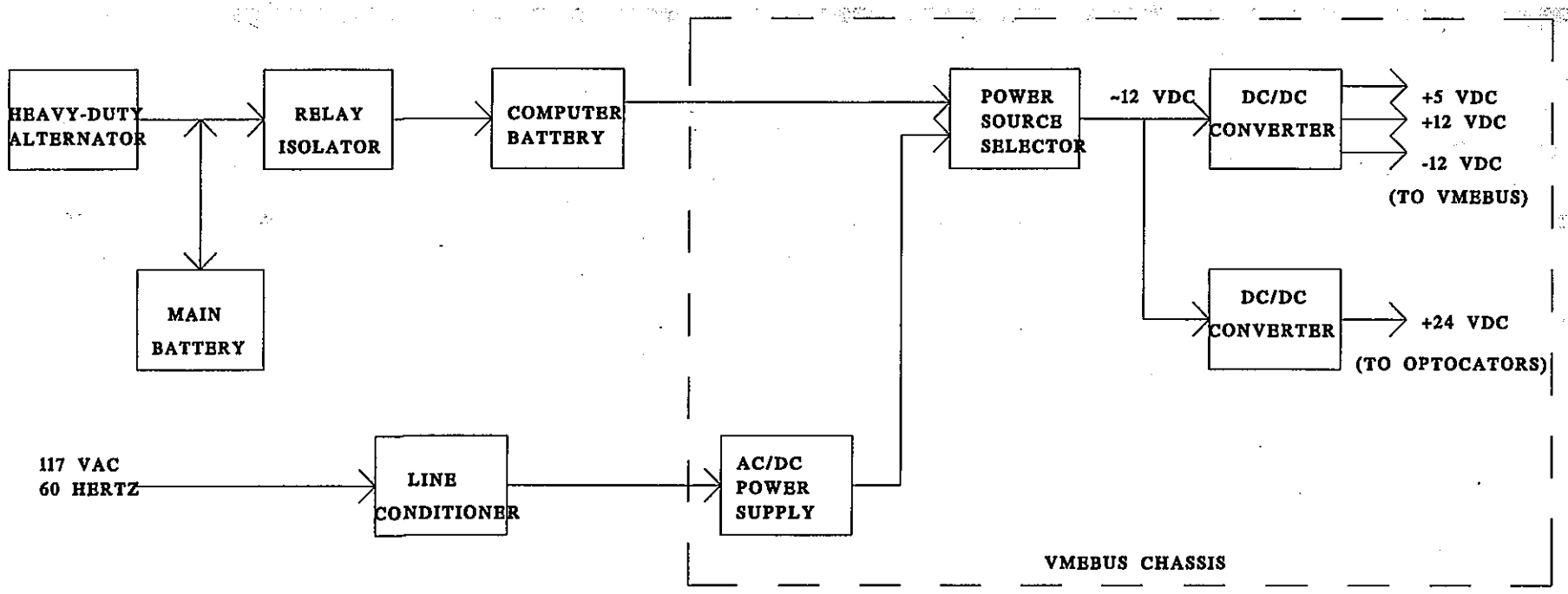


Figure 3 Power System Block Diagram

4. SOFTWARE DESCRIPTION

4.1 Profile Van Computer Software

The profile van computer software consists of three parts: foreground routines, background routines, and utilities. The foreground routines are modular programs which contain the operator interface and the data storage and processing software. The background routines perform the data acquisition functions and are triggered by an interrupt request. When an interrupt request occurs, the foreground processing is suspended and the background software takes over the operation of the computer. After quickly performing its tasks, the background routine returns control to the foreground. The utility routines perform general functions such as cursor movement and character display, which are needed in many of the foreground routines.

Since speed is very important in the background routines, they are written in assembly language using Borland's Turbo Assembler version 3.0. All other software in the system is written in the C programming language using Borland's C++ Compiler version 3.1.

In addition to these three types of routines, two stand-alone programs were written for the system. The serial communication program enables a laptop computer to be used as the operator interface in place of the keyboard and display. A separate data processing program was also written to allow the operator to process the raw profile data on computers outside of the van.

A block diagram of the profile van computer software is shown in Figure 4.

4.1.1 Background Routines

The two background routines perform the data acquisition functions in the profile van. One routine collects profile data from the sensors every time a pulse comes into the computer from the wheel pulse sensors. The other routine acquires navigation data every millisecond and stores it in a buffer. The background routines are only enabled during a data acquisition run.

4.1.1.1 DISTISR.ASM (Distance-based Interrupt Service Routine)

The pulse train generated by the left front wheel pulse sensor is connected to circuitry on the signal conditioning circuit card that issues an interrupt request (IRQ9) for every rising edge. The computer then executes this routine to service the request. The first task performed is to reset the interrupt request line. The routine then gets vehicle velocity data from the 24 bit timer on the DIO card, which measures the time between interrupts (the time for the van to travel 25.4 mm, 1″). Next, it reads data from the left and right wheelpath laser sensors and accelerometers. It accumulates the laser sensor and

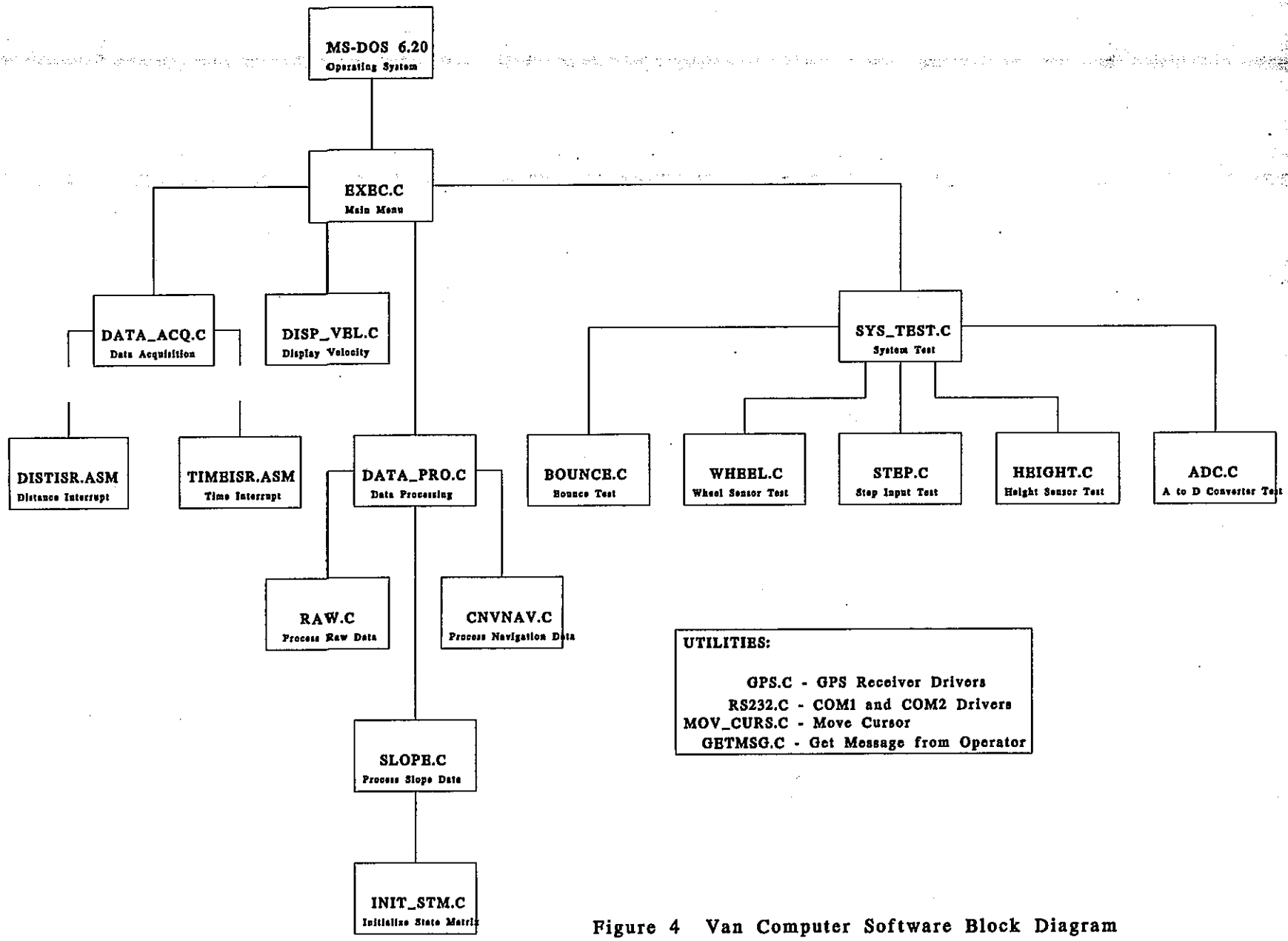


Figure 4 Van Computer Software Block Diagram

accelerometer data over a 305 mm (1′) interval and then calculates the average of each signal for that interval. The averages are stored in a buffer that has space for 528 samples (161 m or 0.1 miles). When the buffer becomes full, a flag is set to tell the foreground routine (DATA_ACQ.C) to write the data into the raw data file.

4.1.1.2 TIMEISR.ASM (Time-based Interrupt Service Routine)

The second 24 bit timer on the DIO circuit card is programmed to generate an interrupt every millisecond using interrupt request line 10 (IRQ10). This routine is then executed to service the interrupt request. The first task is to reset the interrupt request line. The next task is to check if a character has been received from the GPS receiver. If a character is available, it is read from the COM2 port and stored in a message buffer. If the character is a carriage return (signaling a complete message), a flag is set telling the foreground (DATA_ACQ.C) to process the message. Next, the data from the navigation sensors is acquired through the ADC circuit card. The navigation data is saved in a linear buffer that can store 1000 samples (one second), and a flag is set to notify the foreground when the buffer becomes full. If the loop excitation monitor signal is above the threshold (3.5 volts), a flag is set to tell the foreground that an inductive loop has been detected. The last task performed is to check if another character is available from the GPS receiver and process the character using the method already described.

4.1.2 Foreground Routines

The foreground routines are programs that interact with the operator during the normal operation of the system. The program may be set up to run automatically from the AUTOEXEC.BAT file when the computer is first turned on, or it may be started manually from the DOS prompt by entering "profile" (which triggers a batch file to run). Entering "profile laptop" causes the input and output to be re-directed to a laptop computer through a serial connection.

4.1.2.1 EXEC.C (Main Menu)

This module is the entry point for the profilometer software. It checks for command line arguments to determine whether or not a laptop computer will be used for the operator interface. Then it initializes all the global variables and sets up the computer hardware (the CPU, DIO, and ADC circuit cards). Next, it displays a header screen showing the Caltrans logo and the software version number. Finally, it prints the main menu showing the five operations that can be selected by the operator. These five selections include the following:

1. Data Acquisition
2. Display Van Speed and GPS Data
3. Data Reduction and Processing

4. System Test
5. Exit to DOS

If the exit to DOS selection is chosen, the program returns control of the computer to DOS. The remaining entries are described below.

4.1.2.2 DATA_ACQ.C (Data Acquisition)

This module controls the collection of profile data during a data acquisition run. It prompts the operator for a file name to store the data under and requests other run-specific information. It opens files to store the data in and initializes the run variables, serial ports, timer chips, and interrupt hardware. It then checks the status of the GPS receiver and prompts the operator to begin the run. During the data acquisition run, it displays elapsed time and distance in real time while the background routines store the raw profile and navigation data. At the end of the run it closes all the data files and returns to the main menu.

4.1.2.3 DISP_VEL.C (Display Van Speed and GPS Data)

This module is used as a confidence check for the wheel pulse sensors and the GPS receiver. After initializing the serial ports, timer chips, and interrupt hardware, it displays the elapsed time and distance in real time as the van travels down the road. It also prints out the position messages from the GPS receiver as they are received at the rate of one per second. The operator can use this routine to verify that the system timer, wheel pulse sensors, and GPS receiver are working properly.

4.1.2.4 DATA_PRO.C (Data Reduction and Processing)

Selecting this routine causes the data reduction and processing menu to be displayed. The following five choices are available from the menu:

1. Process Raw Sensor Data
2. Process Slope Profile Data
3. Batch Process Raw Sensor Data
4. Convert Navigation Binary Data File to Text
5. Return to the Main Menu

Choosing the return to the main menu selection causes the screen to be erased and the main menu to be displayed. If option 1 or 3 is selected, the screen is erased and a list of the available raw profile sensor data files is displayed. The operator is then prompted to select a file to be processed. In batch processing, the operator can select a group of files to process and can then leave the computer unattended while it completes the data reduction. A log file is generated during the batch processing that identifies any errors

encountered. The operator can examine this file later to ensure that the processing was successfully completed. If option 2 is selected, a list of the available slope profile data files is displayed and the operator is asked to choose a file to process. If option 4 is selected, the operator will be prompted to choose a file to process from the list of available navigation files. The routines that perform the actual processing are described below.

4.1.2.4.1 RAW.C (Process Raw Sensor Data)

This module processes the raw profile sensor data files in two steps. First, it separates the raw data file into five different files, one for the van velocity, one for each of the left and right accelerometers, and one for each of the left and right laser height sensors. At the same time it converts the raw data into engineering units for each file. The second step is to use the converted velocity, accelerometer, and height value for each wheelpath to calculate the slope profile for that wheelpath.

4.1.2.4.2 SLOPE.C (Process Slope Profile Data)

This routine processes the slope profile into an elevation profile and an IRI file. The elevation profile is calculated by integrating the slope profile data point-by-point with respect to distance. The IRI is calculated by solving the equations of motion for the quarter car simulation using matrix algebra and with the slope profile as the input. The modeled vertical motion of the sprung mass is accumulated over one tenth of a mile and is then multiplied by ten to get the IRI in units of inches per mile. The matrices used in the calculations are initialized in the INIT_STM.C routine described below.

4.1.2.4.2.1 INIT_STM.C (Initialize State Transition Matrix)

The state transition matrix and particular response matrix used in processing the slope profile into an IRI are dependent upon the data acquisition sample distance (one foot). This module initializes the two matrices prior to the IRI calculations in the SLOPE.C module.

4.1.2.4.3 CNVNAV.C (Convert Navigation Binary Data File to Text)

The navigation data collected during a data acquisition run is saved in binary form in the navigation file. This routine converts the binary data to ASCII text so that it can be imported into a spreadsheet for further processing.

4.1.2.5 SYS_TEST.C (System Test)

Selecting this routine causes the system test menu to be displayed. The following six choices are available from the menu:

1. Bounce Test
2. Wheel Pulse Sensor Test
3. Step Input Test (Height Sensor)
4. Height Sensor Test
5. Analog-to-Digital Converter Test
6. Return to the Main Menu

Choosing the return to the main menu entry causes the screen to be erased and the main menu to be displayed. The remaining tests are described below.

4.1.2.5.1 BOUNCE.C (Bounce Test)

The bounce test is another confidence check for the profilometer system. First, the van is parked on a level surface with the parking brake set. Flat metal plates are then placed under each laser sensor so that the laser light spot reflects off of them. After the operator begins the test, the van bumper is "bounced" up and down by an amplitude of about one inch. The program collects profile data at one millisecond intervals, simulating a vehicle velocity of 25.4 meters per second. After about 30 seconds the operator stops the data acquisition. The raw data file that was created from this test (BOUNCE.RAW) can then be processed like any other raw data file. If the accelerometers and laser height sensors are functioning correctly, their signals will cancel, resulting in a relatively flat elevation profile and a small IRI value.

4.1.2.5.2 WHEEL.C (Wheel Pulse Sensor Test)

This test enables the operator to check the accuracy of the odometer and the vehicle velocity sensor and to calibrate the odometer, if necessary. It should be used on a low traffic volume road and should cover a known distance. The program prompts the operator for the beginning and ending post miles of the test site. After the front wheel has been centered on the starting mark, the operator begins the test and the driver accelerates the van to the desired velocity and drives over the course. The driver then slows down at the end of the course and stops the van with the front wheel centered on the ending mark, at which point the operator stops the test. The program will then display the known and measured distance traveled and the percent error between them. It will also show the known and measured average velocity and the percent error between them. Finally, it will display the number of elapsed odometer counts over the traveled distance. The elapsed odometer counts can be used to calibrate the odometer by calculating the number of counts per 1.6 km (1 mile) and editing the ODOMETER.TXT file to place the new value in it. The nominal value for the number of counts per 1.6 km (1 mile) is 63360, or one count per 25.4 mm (1″).

4.1.2.5.3 STEP.C (Step Input Test)

This test checks the accuracy of the laser height sensors in making static height measurements. With the van parked on a level surface and the parking brake on, the operator places the metal step blocks under each laser height sensor so that the laser light spot reflects off of them. Either the lowest or the highest face of the step block is used for the starting point and all subsequent measurements are made relative to that point. After the operator begins the test, the program displays the starting value, the current value, and the difference between the two (all values are in millimeters). As the different faces of the step block are slid under the light spot, the current value and the difference will change. If the lowest face is used for the starting point, the difference values should read 0, 1, 3, 7, and 15. If the highest face is used for the starting point, the readings should be 0, -8, -12, -14, and -15. The tolerance for these measurements is ± 0.250 mm. No one should be sitting in the van during this test.

4.1.2.5.4 HEIGHT.C (Height Sensor Test)

This test simply displays the data from the left and right laser height sensors on the screen in real time. It shows the raw data in hexadecimal, the height in meters, the height in inches, and the number of samples that were averaged in order to calculate these height values. The Optocator interface circuit card is programmed to average 32 samples for each height value that it sends to the computer. If the value for the number of samples averaged is not 32, then the laser sensor has detected an invalid reading. When the operator rocks the van back and forth, the height values displayed on the screen should change, indicating that the sensors are working correctly.

4.1.2.5.5 ADC.C (Analog-to-Digital Converter Test)

This test displays in real time the raw data and voltages from the first eight channels on the ADC circuit card. Channels 0 and 1 are connected to the left and right accelerometers, respectively. Channels 2 through 5 are connected to the navigation sensors. Channels 6 and 7 are not connected to sensors, but are used for diagnostic purposes. Channel 6 is tied to ground on the ADC card and should always read within ±0.01 volts of ground (0.00 volts) during this test. Channel 7 is tied to +5 volts on the ADC card and should always read within ±0.05 volts of that value. The operator can use this test to verify that the accelerometers are working by rocking the van back and forth and seeing that the voltages on channels 0 and 1 are changing.

4.1.3 Utility Routines

The utility routines are general purpose programs that are called by more than one of the foreground routines.

4.1.3.1 GPS.C (GPS Receiver Drivers)

This module contains the driver routines for the GPS receiver. The check_GPS routine checks if the GPS receiver is responding to commands, and prompts the operator if no response is detected. GPS_status commands the receiver to output a status message to the van computer, which is decoded and displayed to the operator. The PMGLH function is called by GPS_status to decode the status message. The GPS_messages_off routine commands the receiver to stop transmitting all the messages that it is currently sending. The read_message function removes GPS position messages from a buffer that is filled by the background routines and writes them into a file. The checksum routine detects errors by calculating the checksum characters for either a message coming in from the receiver or for a command going out to the receiver.

4.1.3.2 RS232.C (COM1 and COM2 Drivers)

This module contains the routines that handle communication between the GPS receiver, laptop computer, and the van computer. The serial_init routine uses the BIOS to initialize the COM2 port that is connected to the GPS receiver. The init_COM1 routine initializes the COM1 port that is connected to the laptop computer. The serial_in routine reads characters from the laptop computer and provides them to the GETMSG.C routines to simulate input from the keyboard. The serial_out routine is used to transmit commands to the GPS receiver through the COM2 port.

4.1.3.3 MOV_CURS.C (Move Cursor)

This routine uses ANSI escape sequences to move the cursor to any row and column on the screen. The ANSI.SYS driver must be loaded into memory in order for this routine to work.

4.1.3.4 GETMSG.C (Get Message from Operator)

These functions control the source of operator input and the destination of screen output. If the laptop computer is selected for the operator interface, input to the van computer comes from the COM1 port and screen output goes to the COM1 port. Otherwise, the operator interface is through the keyboard and local display. The chkkey function checks for input from either the keyboard or the COM1 port. The laptop_in function reads a single character from the COM1 port. The getkey routine gets a character from either the keyboard or the laptop_in function. The getmsg routine gets a string of characters from either the keyboard or the COM1 port, terminating on a received carriage return.

4.2 Laptop Computer Software

The laptop computer software can operate in two modes. During normal operation of the system, the serial communication software links the van computer to the laptop computer in half duplex mode. The physical interface between the computers is a serial cable (RS-232C) connecting the COM1 ports on each machine. The output of the van computer is re-directed from the display to the COM1 port, and the operator input is taken from the COM1 port instead of from the keyboard.

The second mode of operation is data processing. After the data files have been acquired, they can be more rapidly processed on the laptop computer using the stand-alone processing software.

A block diagram of the laptop computer software is shown in Figure 5.

4.2.1 SERIAL.C (COM1 Drivers for the Laptop Computer)

This stand-alone program is executed on the laptop computer to enable it to be used as the interface between the operator and the van computer. It reads data from the van computer sent through the COM1 port and displays it on the screen. The operator can press keys on the laptop keyboard which are echoed locally to the screen and sent back to the van computer through the COM1 port.

4.2.2 PROCESS.C (Data Reduction and Processing Software)

At the time when this project was started, the 20 megahertz 80386SX microprocessor CPU card that was purchased was the fastest computer available in a VMEbus form factor. Since that time, computer speed and processing power have increased significantly. For example, the laptop computer used for the operator interface in the van contains a 33 megahertz 80486DX microprocessor, which is almost an order of magnitude faster than the van computer in terms of processing power. Therefore, the decision was made to transfer the raw profile data from the van computer to the laptop computer in order to reduce the amount of time necessary for processing the data. This data transfer is accomplished using the existing serial link and an off-the-shelf communication program called LapLink V. Although the van computer still possesses the ability to process the data itself, typically the raw data is reduced using the laptop computer and the processing software.

This stand-alone program consists of the same RAW.C, SLOPE.C, INIT_STM.C, and CNVNAV.C routines described above along with a slightly modified version of the DATA_PRO.C routine described above. It enables the operator to reduce the profile data on any PC/AT compatible computer that has a math coprocessor. Therefore, after the

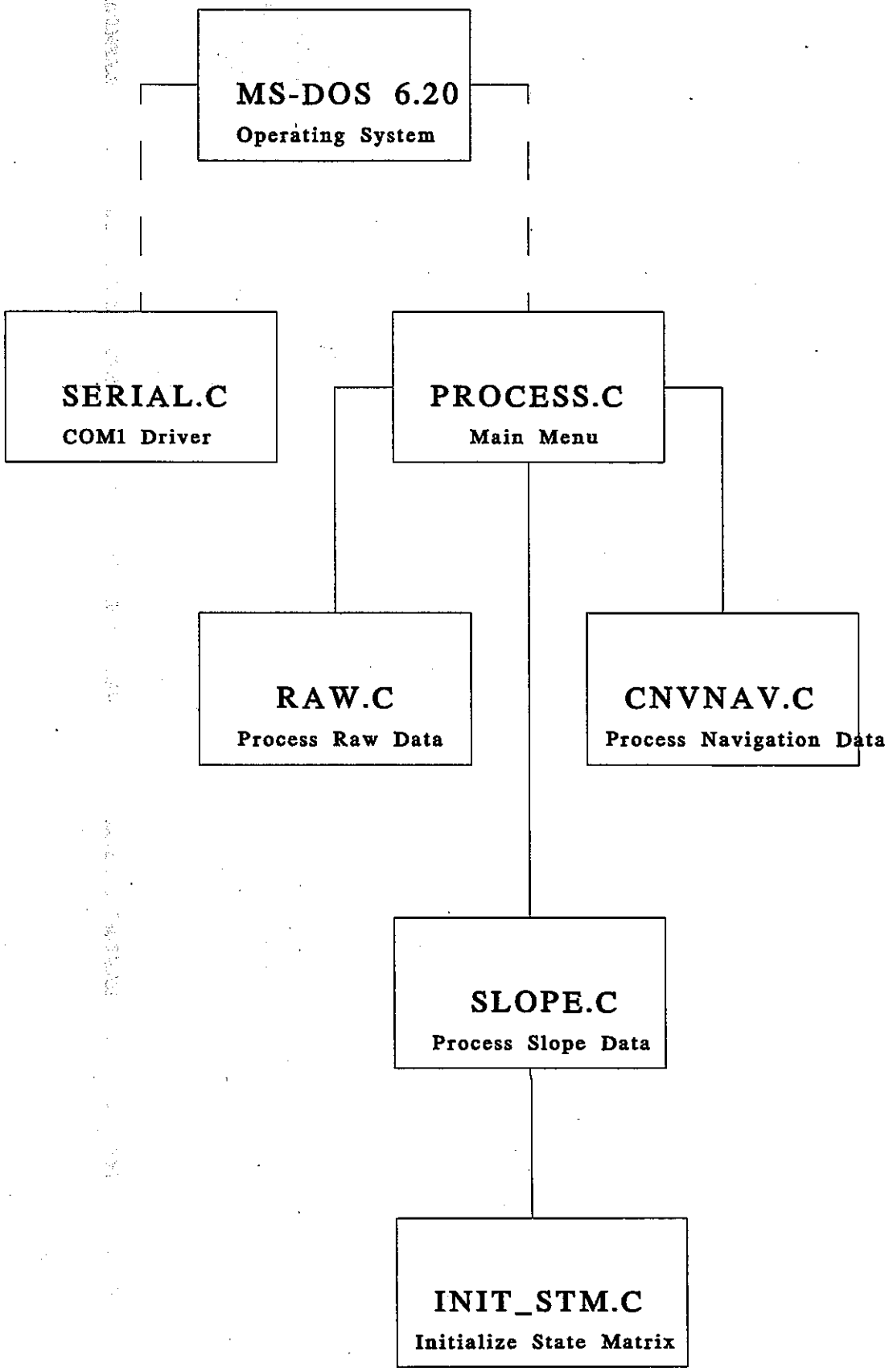


Figure 5 Laptop Computer Software Block Diagram

raw data is transferred to the laptop computer, the operator can process it in the comfort of an office or motel room rather than while sitting in the van.

5. DATA FORMAT

5.1 Program and Data Files

The profile program (PROFILE.EXE) is stored on the hard disk drive (C:) in a sub-directory called PROFILE. The Borland Graphics Interface library (EGAVGA.BGI) that is required by the profile program is also stored in this sub-directory. The ODOMETER.TXT file containing the number of odometer counts per 1.6 km (1 mile) is located there as well. The PROFILE sub-directory has 5 sub-directories of its own, including the following:

RAW
DATA
SLOPE
ELEV
IRI

The raw sensor data files that are collected during a data acquisition run are placed in the RAW sub-directory. During the data reduction process, other files are created from these raw data files and are stored in the remaining sub-directories according to their contents. The files that are located in each of the sub-directories are described below.

5.2 RAW Sub-directory

Prior to a data acquisition run, the program prompts the operator for a data file name and some specific information regarding the run, such as location, length, and test conditions. The software then uses that file name to open five files that have the same name but different extensions. All of these files are placed in the RAW sub-directory. For example, if the operator entered a file name of TEST, the profile program would open the following files in the RAW sub-directory:

TEST.SET
TEST.RAW
TEST.NAV
TEST.GPS
TEST.KEY

In addition, the program would open a TEST.NAV, TEST.GPS, and TEST.KEY file on the RAMdrive (D:). During a data acquisition run, the navigation, GPS, and keyboard data is stored in these files on the RAMdrive. After the run, the program copies the contents of each of these files into their counterpart files on the hard disk drive. The purpose for using the RAMdrive is to reduce the amount of time that the computer spends on writing data to the hard disk, which is much slower than a RAMdrive. Only the

profile sensor data is written to the hard disk during the run, since the amount of profile data collected during a run is typically too large to fit on a RAMdrive. The contents of the five example files are described below.

5.2.1 TEST.SET

TEST.SET is a text file containing the setup information for the data acquisition run. The program prompts the operator for the test location (district, county, route, lane number, and direction of travel), the beginning and ending post miles, and comments for the run. This information is saved in the setup file and then the file is closed before the run begins. An example of a setup file is shown in Appendix D.

5.2.2 TEST.RAW

The TEST.RAW file is a binary file which contains the raw data from the profile sensors. Before the run starts, a header consisting of the date, time, and a flag indicating whether or not the file has been processed is written into the file. During the run, the distance-based background routine saves a sample of raw sensor data after every 305 mm (1′) of travel. The samples are stored in two memory buffers that have a capacity of 528 samples (161 m or 0.1 miles). When the first buffer becomes full, the background routine sets a flag telling the foreground to write the contents of the buffer into the TEST.RAW file. While the first buffer is being emptied, the raw sensor data is stored in the second buffer. The data storage then alternates between these two buffers for the length of the run. At the end of the run the program closes the file.

Each sample of raw profile sensor data consists of five sixteen- bit words. The first word is the vehicle velocity data, which comes from the velocity sensor. The next two words are the left accelerometer and height sensor data, respectively. The last two words are the right accelerometer and height sensor data. An example of a raw sensor data file is shown in Appendix D.

5.2.3 TEST.NAV

The TEST.NAV file is a binary file which contains the processed data from the navigation sensors. Before the data acquisition run, a header consisting of the date and time is written into the file. During the run, the time-based background routine saves a sample of raw sensor data every millisecond. The samples are stored in two memory buffers that have a capacity of 1000 samples (one second). When the first buffer becomes full, the background routine sets a flag telling the foreground to process the data. While the first buffer is being processed, the raw sensor data is stored in the second buffer. The data storage then alternates between these two buffers for the length of the run.

Each sample of raw navigation sensor data consists of three sixteen-bit words. The first two words are the x and y axis signals from the magnetic flux gate compass, respectively. The last word is the signal from the solid-state angular rate sensor. The foreground routine processes a buffer of samples every second by taking the average of 1000 samples for each of the three signals. It then saves the current elapsed odometer count, which is a 32-bit word, followed by the three averaged data values, into the TEST.NAV file on the RAMdrive. An example of a navigation data file is shown in Appendix D.

During post-mission data processing, the data in the navigation file is converted from a binary format to ASCII text and stored in a file with a .PRN extension. This file can then be imported into a spreadsheet (such as LOTUS 123, Excel, or Quattro Pro) that produces a plot of the vehicle "track". An example of an ASCII text navigation file is shown in Appendix D.

5.2.4 TEST.GPS

The TEST.GPS file is an ASCII text file which contains the vehicle position messages sent by the GPS receiver. Prior to the data acquisition run, a header consisting of the date and time is written into the file. During the run, the time-based background routine checks the COM2 port for a character from the GPS every millisecond. Any characters that are detected are stored in a memory buffer until a line feed character is received, indicating that a complete message (60 characters) is available. The background then sets a flag telling the foreground routine to write the message to the TEST.GPS file on the RAMdrive.

The GPS receiver transmits a position message at the rate of about one per second. These messages conform to the NMEA 183 standard. The format for the GPS position message is shown in Appendix D.

5.2.5 TEST.KEY

The TEST.KEY file is an ASCII text file which contains the messages that the operator generates from the keyboard and also has the location of any inductive loops that were detected during a run. Before the run, a header consisting of the date and time is written into the file by the profile program. During the run, the operator can press any of the function keys in the range from F1 to F10 to generate a programmed message to the TEST.KEY file. The following messages are available:

F1 : Speed below 20mph
F2 : Speed above 20mph
F3 : Flex to Rigid
F4 : Rigid to Flex
F5 : Railroad Crossing

F6 : Lane Closed
F7 : Back in Lane
F8 : Unusual Occurrence
F9 : Bridge Approach
F10: Bridge Departure

When the program detects that a function key has been pressed, it writes the current elapsed time and elapsed odometer count, along with the corresponding message, into the TEST.KEY file on the RAMdrive.

In addition, the background routine checks every millisecond to see if an inductive loop has been detected. If one is detected, it sets a flag telling the foreground to process the loop. The foreground then writes the current elapsed time and elapsed odometer count, along with the string "Loop", into the TEST.KEY file.

5.3 DATA Sub-directory

The raw profile data files in the RAW sub-directory contain five channels of sensor data that are interlaced with each other. When a raw data file is processed, an intermediate step is to separate these five channels of data, convert them into engineering units, and save each one in its own file. The separated data files can then be used to calculate the slope profile for each wheelpath. These files can also be used as a diagnostic tool by viewing the contents to determine whether or not a sensor is giving bad data.

For example, if the file TEST.RAW is selected for processing, five binary data files will be created and stored in the DATA sub-directory. The new files and their contents are as follows:

TEST.VEL - the vehicle velocity in meters per second (meters/second)
TESTL.ACC - vertical acceleration in the left wheelpath (meters/second^2)
TESTL.HGT - vertical height in the left wheelpath (meters)
TESTR.ACC - vertical acceleration in the right wheelpath (meters/second^2)
TESTR.HGT - vertical height in the right wheelpath (meters)

Note that an L and an R are appended to some file names to differentiate the left wheelpath files from the right wheelpath files.

5.4 SLOPE Sub-directory

After the raw sensor data has been separated and converted, the processing software uses it to calculate the slope profile. The slope profile calculating routines take the vehicle velocity file, the accelerometer file, and the height file as inputs and create a slope profile file for each wheelpath. For example, if the input files were TEST.VEL, TESTL.ACC,

and TESTL.HGT, an output file called TESTL.SLP would be created and stored in the SLOPE sub-directory. Similarly, TEST.VEL, TESTR.ACC, TESTR.HGT would be processed to generate a TESTR.SLP file for the right wheelpath.

5.5 ELEV Sub-directory

After a slope profile has been calculated, it can be used to produce an elevation profile. The processing software takes a slope profile as input and numerically integrates it point by point to generate the elevation profile. For example, if a file named TESTL.SLP is used as input, an output file called TESTL.ELV would be created and stored in the ELEV sub-directory. Similarly, a TESTR.ELV file would be generated for the right wheelpath elevation profile.

5.6 IRI Sub-directory

The IRI is calculated from the slope profile also. The processing routines run the slope profile through the quarter car simulation to compute the IRI. For example, if a file called TESTL.SLP is used as input, an output file called TESTL.IRI would be created and stored in the IRI sub-directory. Similarly, a TESTR.IRI file would be generated for the right wheelpath IRI. The procedure for calculating the IRI from the slope profile is well documented in the World Bank Technical Paper Number 45 (7).

6. FIELD TESTING

The profile system was field tested at seven sites with different pavement surface types and different roughnesses. The comparison standard was provided by a dipstick profiling device manufactured by FACE , which measured the profile and calculated the roughness at these sites.

6.1 Yolo County Test Site

The primary field test site was on a frontage road north of Interstate 80 just west of the Yolo Causeway. The surface was rigid pavement (Portland Cement Concrete) with moderate roughness in the range between 2.37 and 3.16 m/km (150 and 200 inches per mile), as measured by the dipstick. A straight, 1609 m (one mile) length of the eastbound lane was marked at 161 m (tenth of a mile) intervals for use in calibrating the odometer. The profile of the last 805 m (half mile) of the section was measured with the dipstick and processed into IRI values for each 161 m (tenth of a mile) in the left and right wheelpath. The profilometer made six data collection runs over each 161 m (tenth of a mile) section and the results are shown below:

805 - 966 m (0.5 - 0.6 miles)

	Left		Right	
Run	m/km	inches/mile	m/km	inches/mile
1	2.68	170	2.79	177
2	2.71	172	2.81	178
3	2.75	174	2.78	176
4	2.73	173	2.78	176
5	2.73	173	2.84	180
6	2.75	174	2.78	176
average:	2.73	173	2.79	177
dipstick:	3.01	191	2.92	185
difference:	-0.28	-18	-0.13	-8

966 - 1127 m (0.6 - 0.7 miles)

	Left		Right	
Run	m/km	inches/mile	m/km	inches/mile
1	2.70	171	2.41	153
2	2.65	168	2.41	153
3	2.62	166	2.40	152
4	2.62	166	2.38	151
5	2.60	165	2.38	151
6	2.64	167	2.41	152
average:	2.64	167	2.41	152
dipstick:	2.73	173	2.35	149
difference:	-0.09	-6	+0.06	+3

1127 - 1287 m (0.7 - 0.8 miles)

Run	Left m/km	Left inches/mile	Right m/km	Right inches/mile
1	2.87	182	2.81	178
2	2.89	183	2.75	174
3	2.97	188	2.81	178
4	2.89	183	2.86	181
5	2.89	183	2.86	181
6	2.89	183	2.94	186
average:	2.90	184	2.84	180
dipstick:	2.97	188	2.70	171
difference:	-0.07	-4	+0.14	+9

1287 - 1448 m (0.8 - 0.9 miles)

Run	Left m/km	Left inches/mile	Right m/km	Right inches/mile
1	2.75	174	3.35	212
2	2.76	175	3.42	217
3	2.73	173	3.52	223
4	2.73	173	3.42	217
5	2.76	175	3.41	216
6	2.79	177	3.42	217
average:	2.76	175	3.42	217
dipstick:	2.90	184	2.84	180
difference:	-0.14	-9	+0.58	+37

1448 - 1609 m (0.9 - 1.0 miles)

Run	Left m/km	Left inches/mile	Right m/km	Right inches/mile
1	2.84	180	2.75	174
2	2.84	180	2.76	175
3	2.92	185	2.71	172
4	2.90	184	2.78	176
5	2.86	181	2.73	173
6	2.95	187	2.79	177
average:	2.89	183	2.76	175
dipstick:	2.79	177	2.45	155
difference:	+0.10	+6	+0.31	+20

Table 1 A Comparison of Dipstick and Profile Van IRIs for the Yolo Test Site

6.2 Nevada Test Sites

The state of Nevada's Department of Transportation has six surveyed test sites in the Carson City area that it uses to verify the operation of its profiling equipment. These test sites are one tenth of a mile long and cover a wide range of roughnesses. Sites 1, 2, and

3 were constructed with flexible pavement (Asphaltic Concrete), while sites 4, 5, and 6 were built with rigid pavement. The sites were surveyed using a dipstick and the profile data was processed to generate IRI values.

The dipstick survey was performed using left and right wheelpaths that were separated by 1.75 m (69 inches). The profilometer has height sensors that are spaced 1.65 m (65 inches) apart, so during the tests it was driven such that the left sensor was directly over the surveyed left wheelpath and the right sensor was 0.1 m (4 inches) to the inside of the surveyed right wheelpath. When viewing the data from each site be aware that the right wheelpath is not the same for the dipstick and the profilometer.

Six data acquisition runs were made at each of the sites with a vehicle velocity of about 80 kilometers per hour (50 miles per hour). The following sections describe the location of the sites and compare the results from the dipstick with those from the profilometer.

6.2.1 Site One

Site one is a flexible pavement site located on the westbound lane of Centerville Lane, which is a county road crossing Nevada State Route 88 about five km (three miles) south of the junction between U. S. Highway 395 and Route 88. The following table compares the profilometer data with the dipstick data for the left and right wheelpath:

Run	Left: m/km	inches/mile	Right: m/km	inches/mile
1	2.75	174	3.39	215
2	2.76	175	3.68	233
3	2.73	173	3.38	214
4	2.75	174	3.49	221
5	2.68	170	3.49	221
6	2.71	172	3.57	226
average:	2.73	173	3.49	221
dipstick:	2.64	167	3.38	214
difference:	+0.09	+6	+0.11	+7

Table 2 A Comparison of Dipstick and Profile Van IRIs for Site One

6.2.2 Site Two

Site two is a flexible pavement site located on the northbound number two lane of U. S. Highway 395 just north of Airport Road, south of Carson City.

Run	Left: m/km	inches/mile	Right: m/km	inches/mile
1	0.66	42	0.92	58
2	0.63	40	0.93	59
3	0.68	43	0.93	59
4	0.65	41	0.96	61
5	0.71	45	0.92	58
6	0.73	46	0.95	60
average:	0.68	43	0.93	59
dipstick:	0.80	51	0.82	52
difference:	-0.12	-8	+0.11	+7

Table 3 A Comparison of Dipstick and Profile Van IRIs for Site Two

6.2.3 Site Three

Site three is a flexible pavement site located on the eastbound lane of U. S. Highway 50 about five km (three miles) west of the junction with U. S. Highway 95A, east of Carson City.

Run	Left: m/km	inches/mile	Right: m/km	inches/mile
1	3.35	212	2.48	157
2	3.54	224	2.29	145
3	3.41	216	2.48	157
4	3.38	214	2.57	163
5	3.50	222	2.11	134
6	3.46	219	2.13	135
average:	3.44	218	2.35	149
dipstick:	3.27	207	2.30	146
difference:	+0.17	+11	+0.05	+3

Table 4 A Comparison of Dipstick and Profile Van IRIs for Site Three

6.2.4 Site Four

Site four is a rigid pavement site located on the eastbound number two lane of Interstate 80 between the Patrick and Tracy-Clark interchanges, east of Reno.

Run	Left: m/km	inches/mile	Right: m/km	inches/mile
1	1.85	117	1.77	112
2	2.00	127	1.72	109
3	1.86	118	1.78	113
4	1.86	118	1.74	110

5	1.86	118	1.66	105
6	1.82	115	1.75	111
average:	1.88	119	1.74	110
dipstick:	2.04	129	1.96	124
difference:	-0.16	-10	-0.22	-14

Table 5 A Comparison of Dipstick and Profile Van IRIs for Site Four

6.2.5 Site Five

Site five is a rigid pavement site located on the westbound number two lane of Interstate 80 between the Thisby-Derby Dam and Tracy-Clark interchanges, east of Reno.

Run	Left: m/km	inches/mile	Right: m/km	inches/mile
1	1.94	123	2.04	129
2	2.04	129	2.00	127
3	2.10	133	2.35	149
4	2.02	128	2.04	129
5	2.07	131	2.00	127
6	2.07	131	1.89	120
average:	2.04	129	2.05	130
dipstick:	1.88	119	2.43	154
difference:	+0.16	+10	-0.38	-24

Table 6 A Comparison of Dipstick and Profile Van IRIs for Site Five

6.2.6 Site Six

Site six is a rigid pavement site located on the westbound number two lane of Interstate 80 between the Patrick and Mustang interchanges, east of Reno.

Run	Left: m/km	inches/mile	Right: m/km	inches/mile
1	0.96	61	1.22	77
2	0.99	63	1.29	82
3	0.96	61	1.26	80
4	1.04	66	1.25	79
5	1.03	65	1.28	81
6	1.04	66	1.31	83
average:	1.01	64	1.26	80
dipstick:	1.45	92	1.96	124
difference:	-0.44	-28	-0.70	-44

Table 7 A Comparison of Dipstick and Profile Van IRIs for Site Six

7. CONCLUSIONS

The high speed profilometer system performed well in its field testing and should prove to be a valuable tool for Caltrans. It demonstrated excellent repeatability during the testing, varying by only a few inches per mile from run to run. It also showed little variation of measured roughness with vehicle speed from 48 up to 96 kilometers per hour (30 to 60 miles per hour). This is a distinct advantage that profiling devices have over RTRRMs, which are calibrated at a given speed (usually 56 kilometers per hour, or 35 miles per hour, well below freeway traffic speed) and must be driven at that speed during a data acquisition run in order to give accurate results.

One disadvantage of the laser height sensors is that they will not operate on wet pavement. If any type of pavement surface has standing water on it, the receiver will not be able to detect the reflected light spot from the laser. Rigid pavement can still be surveyed if it is merely damp and does not have any standing water. Flexible pavement cannot be surveyed unless it is completely dry. This type of sensor failure is detected and flagged by the profilometer program.

The sensors in the modified front bumper seemed to tolerate the harsh automotive environment better than expected. There was some concern that perhaps a forced air system would be necessary to prevent dust and dirt from entering the laser transmitter and receiver holes and contaminating the lenses. The decision was made to build the bumper without an air handler initially and to add one later if field testing showed that it was needed. No problems related to the environment were detected, however, and the only maintenance required is periodic cleaning of the lenses with photographic lens cleaning solution and cotton swabs.

The initial design for the profilometer called for both a driver and an operator to run the system. At the request of the end user, the design was changed to enable a single driver/operator to operate it. Additional safety features were installed as well, such as a rotating light, a programmable flashing light bar, a large sign on the back of the van stating that it is a survey vehicle, and a cellular phone. Subsequent testing has shown that the van can be safely operated by one person.

At the time when this project was started, the cost for a commercial profilometer system that satisfied the Caltrans requirements was estimated to be about $400,000. The cost of this project was about $275,000. The estimated cost to build a second system (with no non-recurring engineering costs) is approximately $60,000 plus the price of the vehicle. In addition, since Caltrans has all the engineering drawings and software source listings, it can make changes to the system itself without having to rely on a contractor. With that extra freedom, however, comes additional responsibility, since Caltrans must support and maintain the hardware and software itself.

8. RECOMMENDATIONS

Although the high speed profilometer performs well in its current configuration, several changes could be made to improve the operation of future systems. The following changes or upgrades should be considered:

1. A smaller vehicle could be used for the profilometer. A mini-van would have a more comfortable ride and get better gas mileage, yet it would still have plenty of room for equipment and luggage.

2. A smaller equipment rack could be used for this application. The equipment rack used in this project was roughly twice as large as necessary. A smaller equipment rack would quite easily fit into the cargo area of a mini-van.

3. Selcom AB has recently introduced a new laser height sensor that is more suitable for pavement roughness applications. The new sensor (model number 2207-505-B) is much smaller in size, lighter in weight, and less expensive than the sensor used in this project, yet it has almost the same performance.

4. The spacing of the laser sensors could be changed from 1.65 to 1.75 meters (65 inches to 69 inches). Although the 1.65 m (65 inch) spacing is adequate for simulating the distance between the left and right wheelpaths on the average passenger vehicle, most profilometers use a sensor spacing of 1.75 m (69 inches). This difference caused problems during the field testing in Nevada, since the dipstick measurements were made using 1.75 m (69 inch) spacing to accommodate the other profilometers that use the sites there.

5. A profilometer that reduces profile data into IRI values in real-time could be built. With the advanced microprocessors, high speed busses, and local area networks that are available today, a second computer could be added to the system that would take the raw sensor data from the existing data collection computer and process it into IRIs while the system is traveling down the road. This enhancement would eliminate the need for the time consuming data transfer from the van computer to the laptop computer and would reduce the amount of data storage needed for all the intermediate files generated during the data reduction process.

6. A major source of difficulty during the development of the profilometer was finding a video display that was capable of being seen in direct sunlight. Several different technologies were available including cathode ray tube (CRT), gas plasma, electroluminescent (EL), and liquid crystal display (LCD). CRTs have the highest brightness and the largest viewing angle, but are also the largest in size and weight. The remaining types are all flat panels, which are smaller and lighter

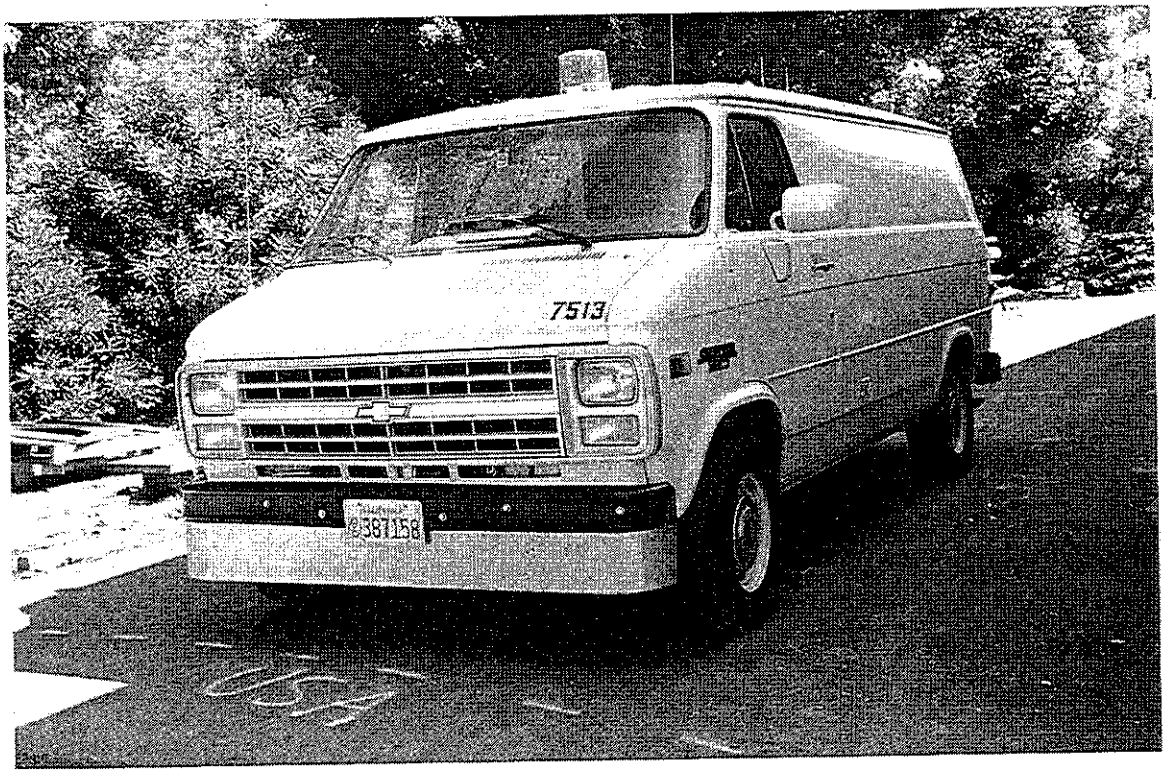
than CRTs and are therefore more suitable for vehicle applications. The first display purchased for this project was an EL panel whose visibility proved to be inadequate on clear, bright days. After searching unsuccessfully for another display, the decision was made to replace the EL display with a laptop computer and to install a small color CRT display in the back that is only used for diagnostics. The laptop has a passive color LCD that is better than the EL display but is still difficult to see under certain conditions. Recently, active matrix color LCDs have been introduced that rival the brightness of CRTs but have all the benefits of a flat panel. This type of display technology should be used on future vehicle-based applications.

9. IMPLEMENTATION

The high speed profilometer system that was built for this project is currently being used (starting in March 1994) by the Caltrans Division of Maintenance, Office of Roadway Maintenance to perform their pavement condition survey. In prior surveys, operators collected roughness data using up to six RTRRMs during a six month period every two years. With the availability of the profilometer, RTRRMs will no longer be used to collect data. A single profilometer driver/operator will survey the approximately 77,000 lane-kilometers (48,000 lane-miles) of California highway continuously and the Office of Roadway Maintenance will merge the data from the profilometer and the manual raters into a "State of the Pavement" report every two years. Fewer personnel will be needed to perform the survey and the difficult time constraint of having only six months to finish will be removed.

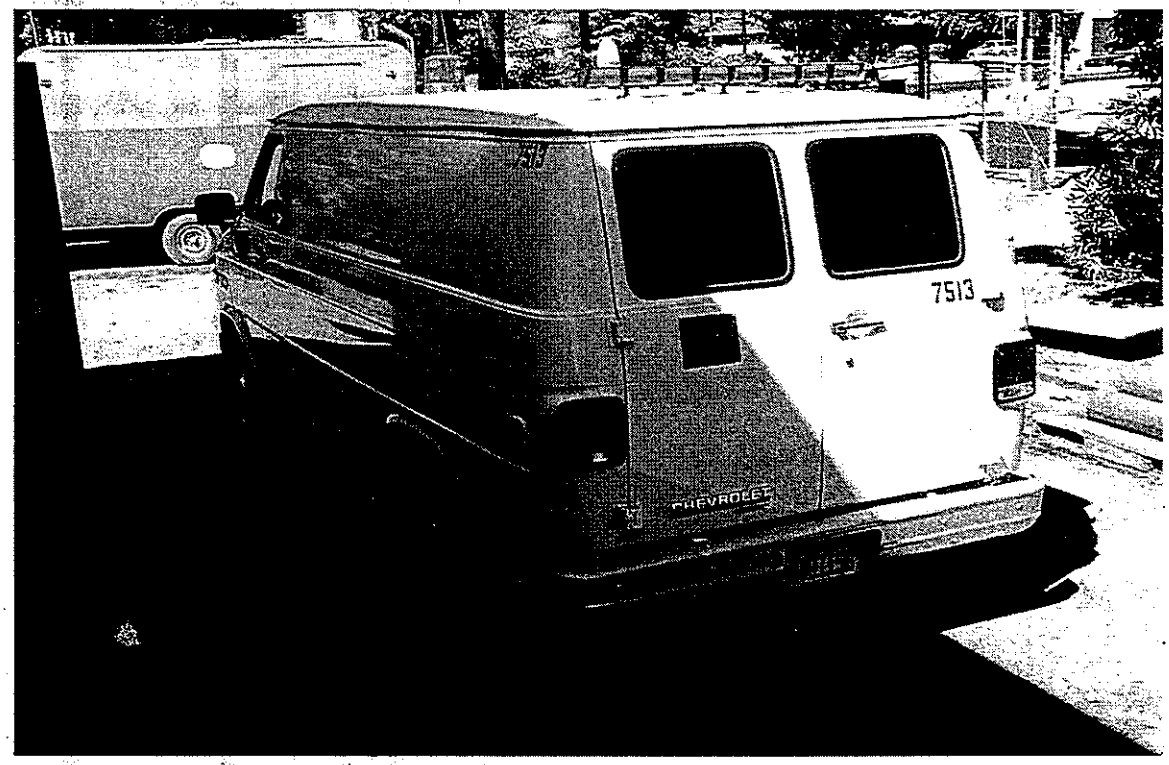
Appendix A

Engineering Data



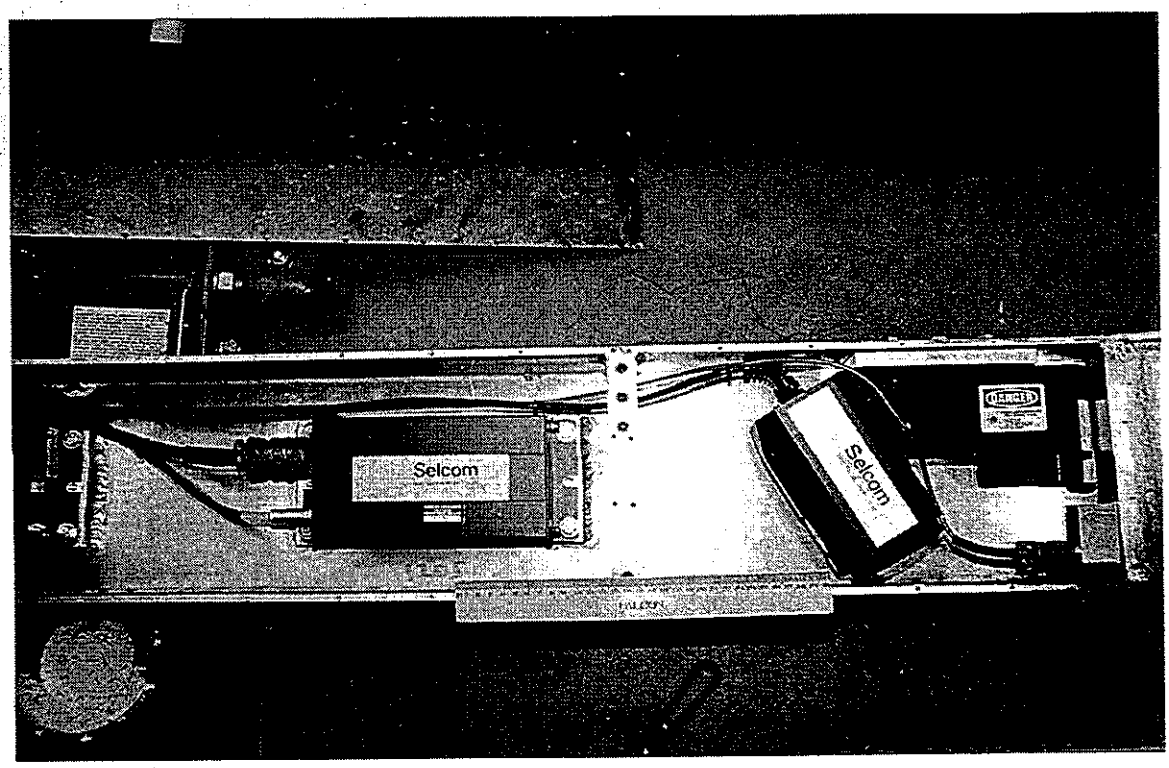
Profile Van, front view

Figure A1



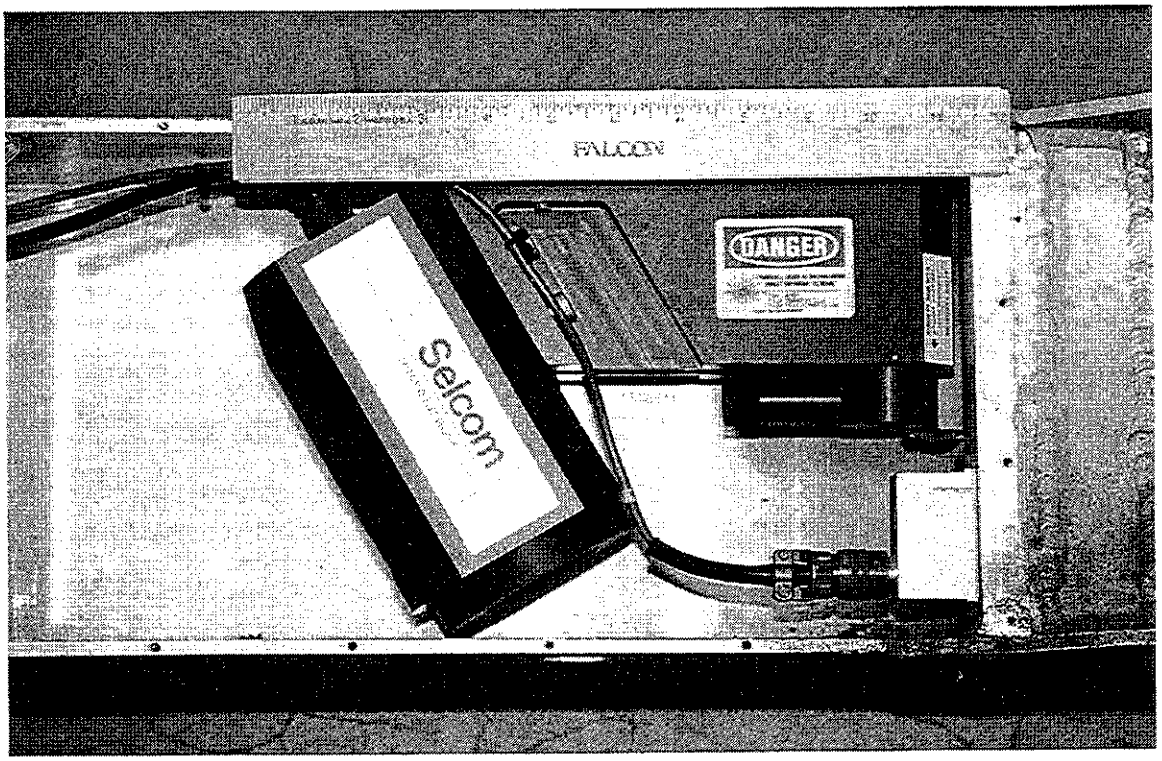
Profile Van, rear view

Figure A2



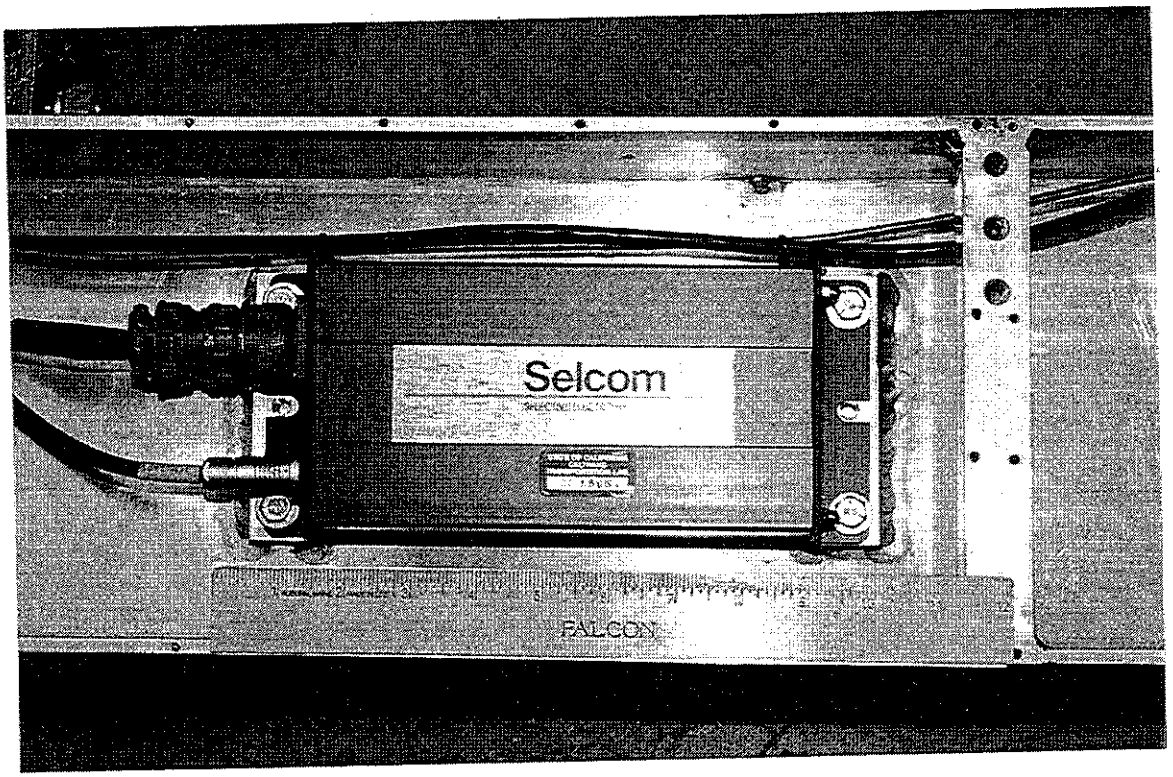
Bumper, right wheel path with cover plate removed

Figure A3



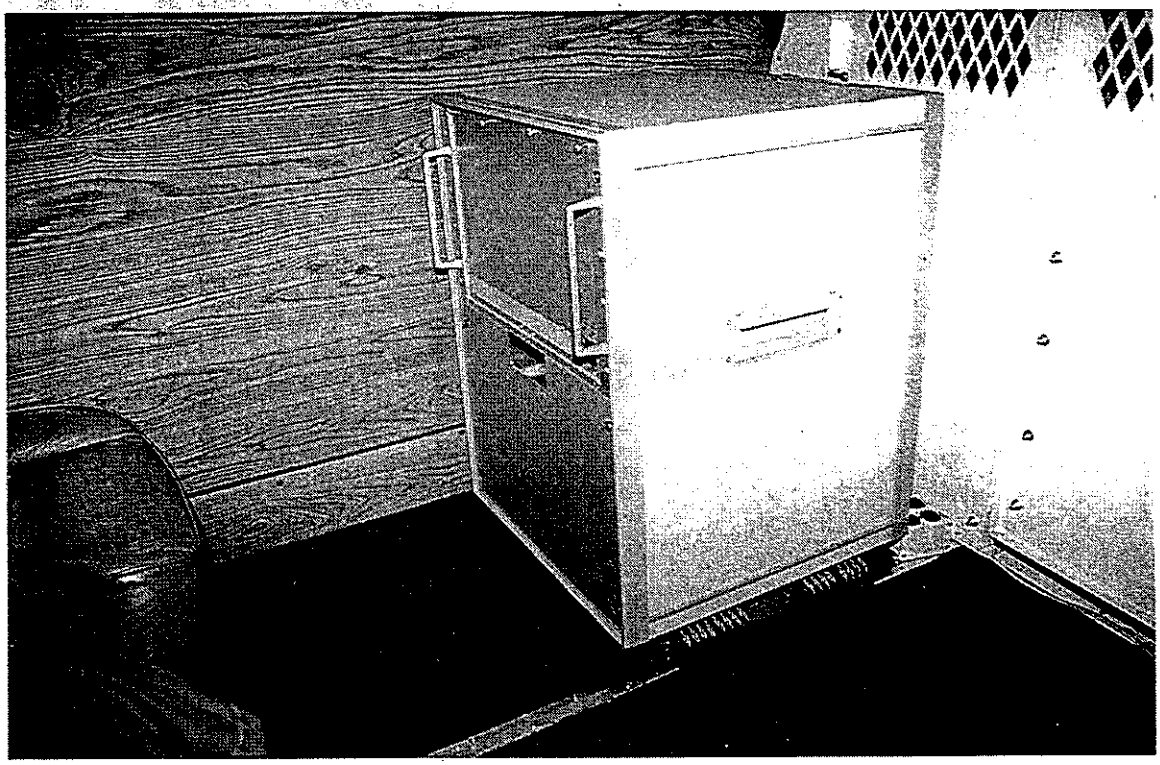
Laser Probe and Accelerometer

Figure A4



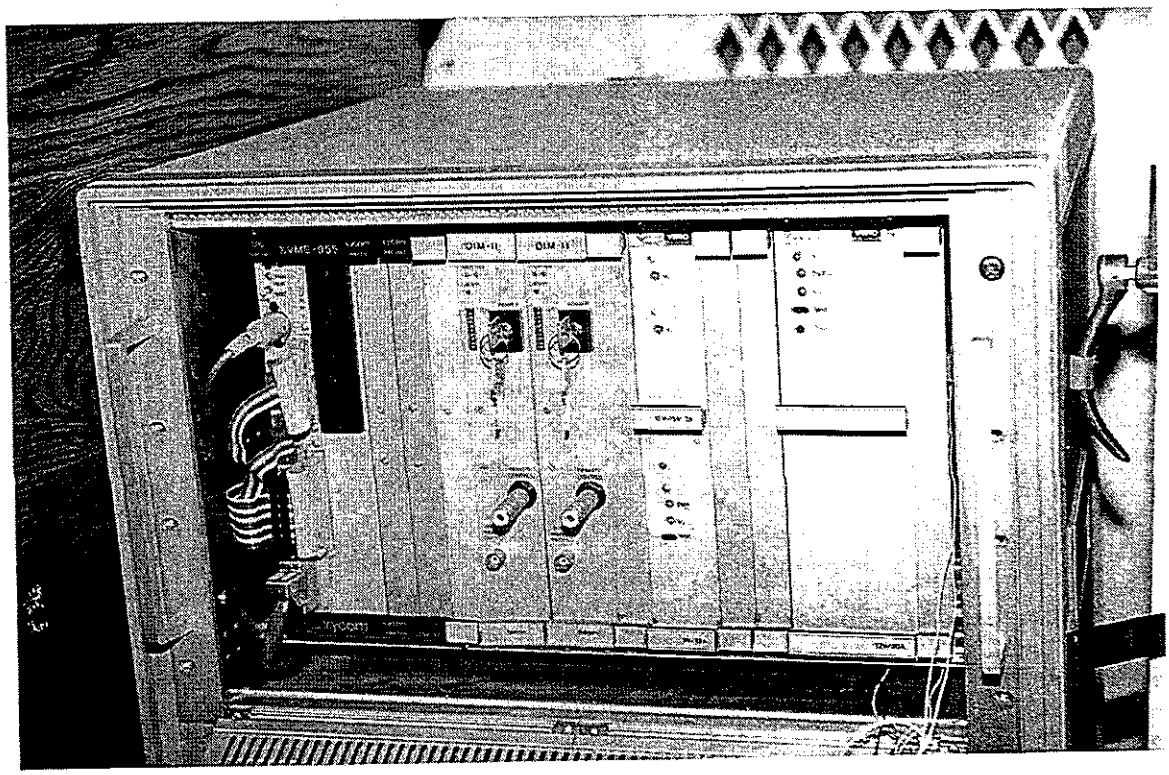
Probe Processing Unit

Figure A5



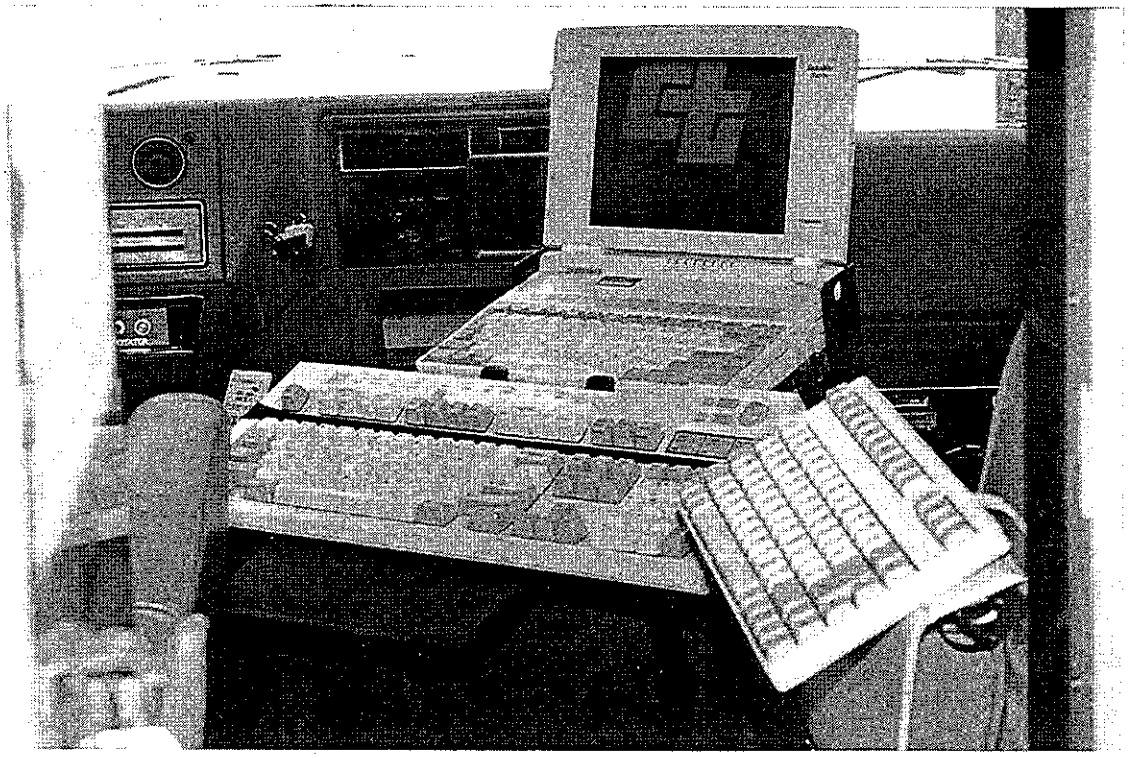
Equipment Console

Figure A6



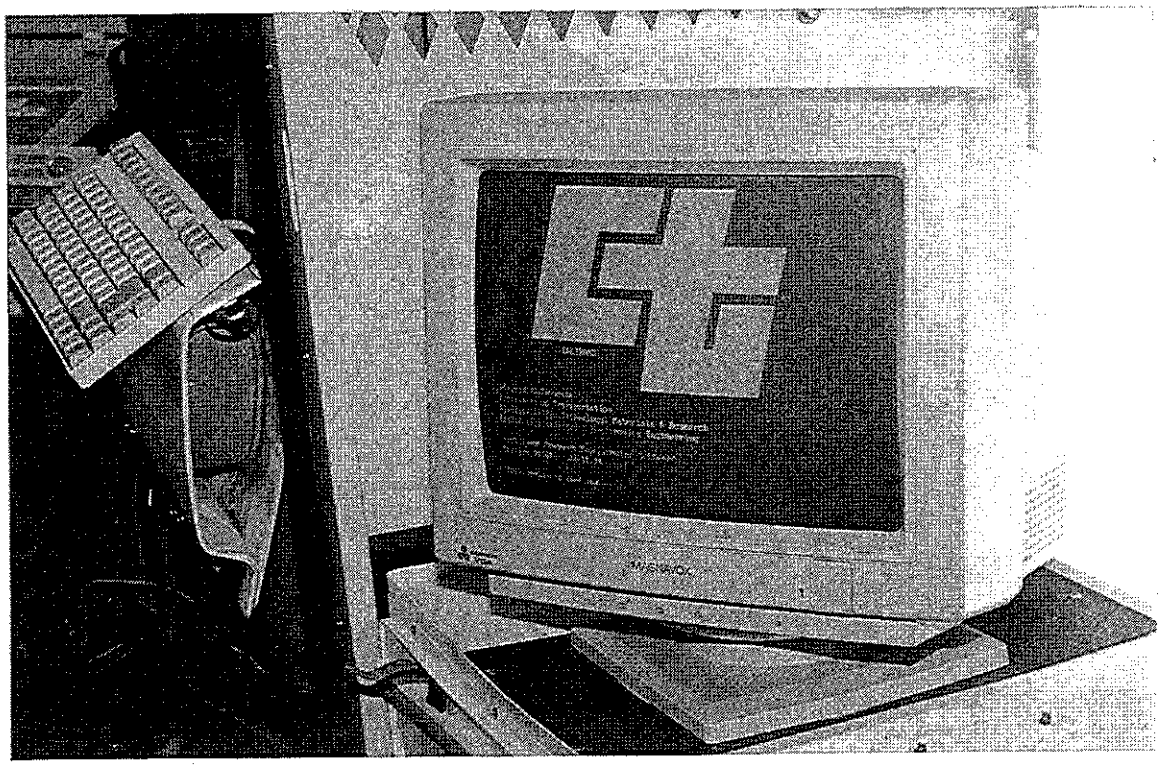
Computer Chassis, front cover open

Figure A7



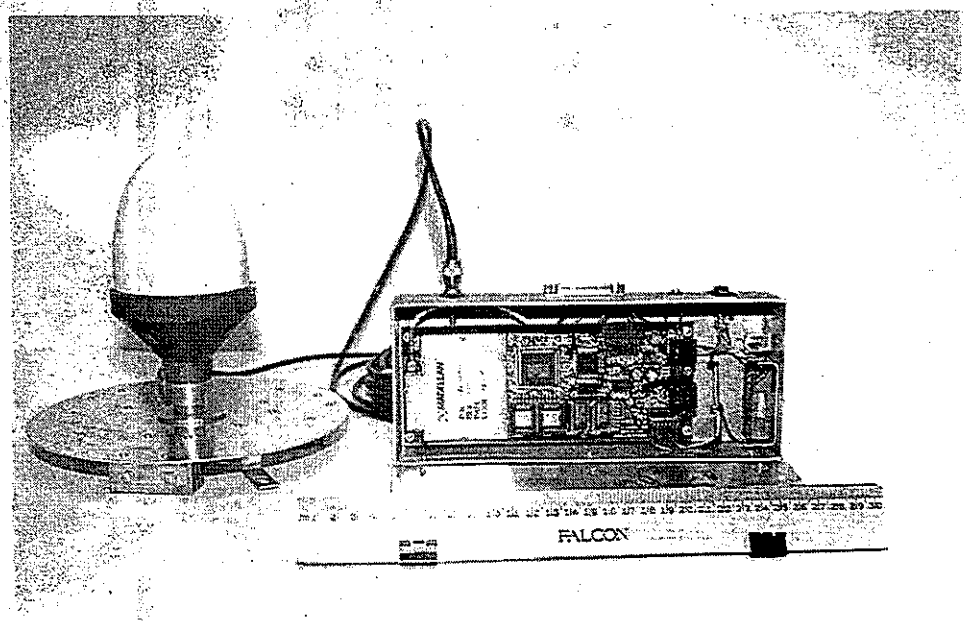
Laptop Computer and Keyboard

Figure A8



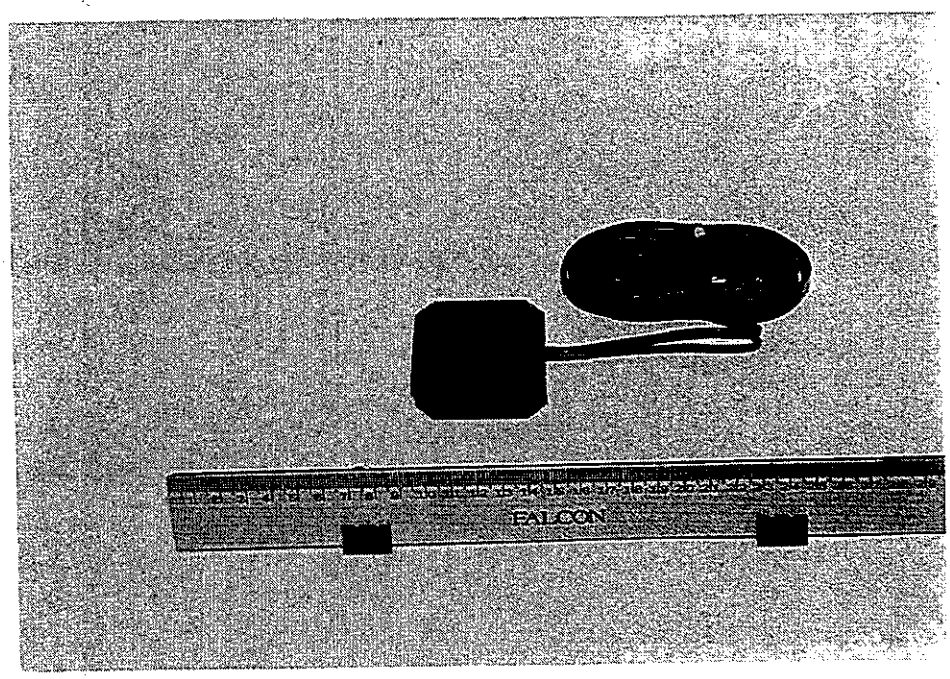
Color VGA Monitor

Figure A9



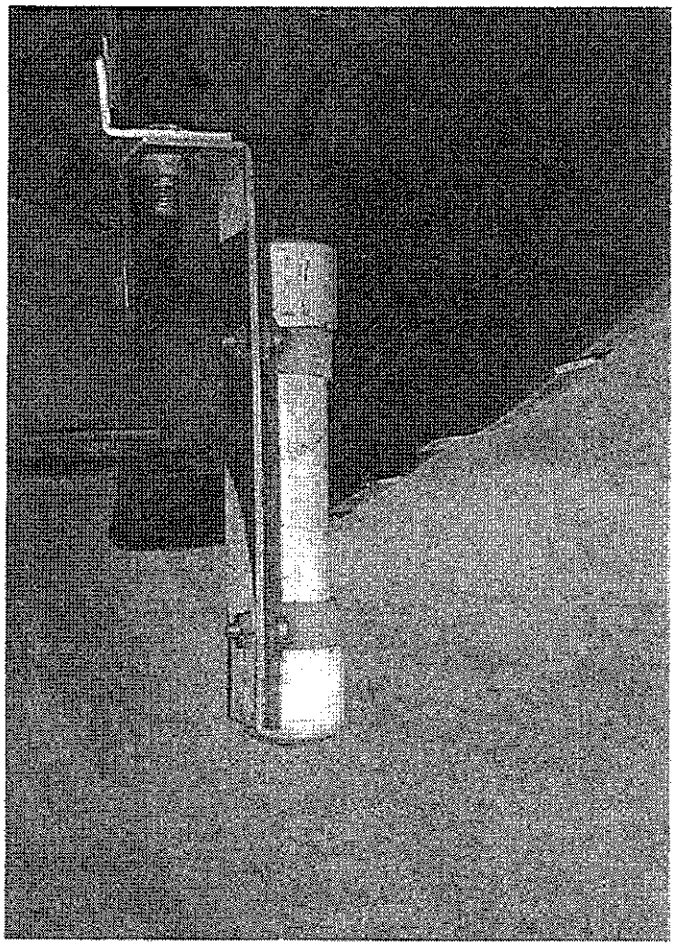
Five Channel GPS Receiver with Antenna

Figure A10



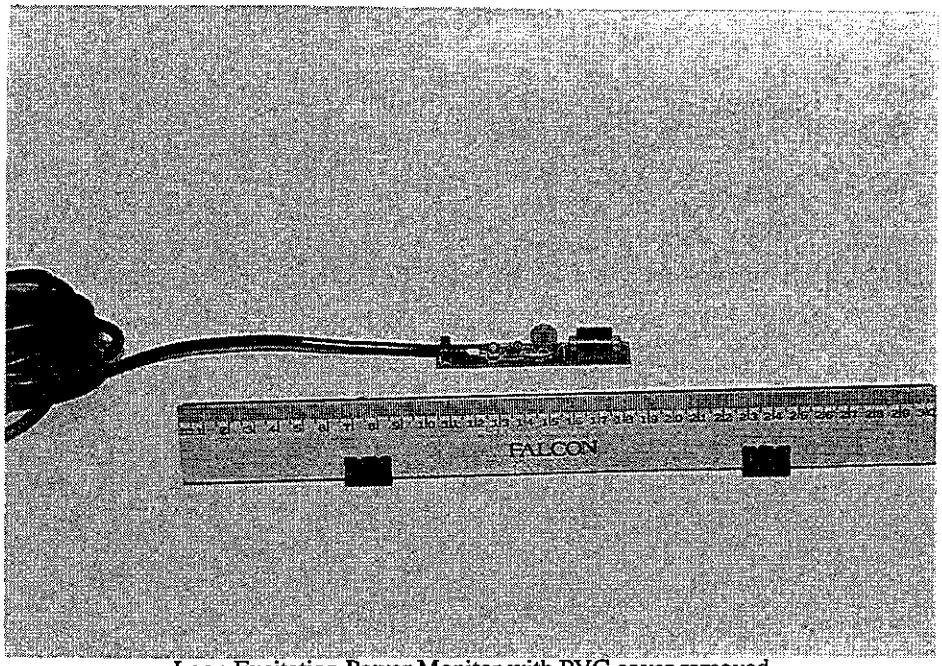
Magnetic Flux Gate Compass

Figure A11



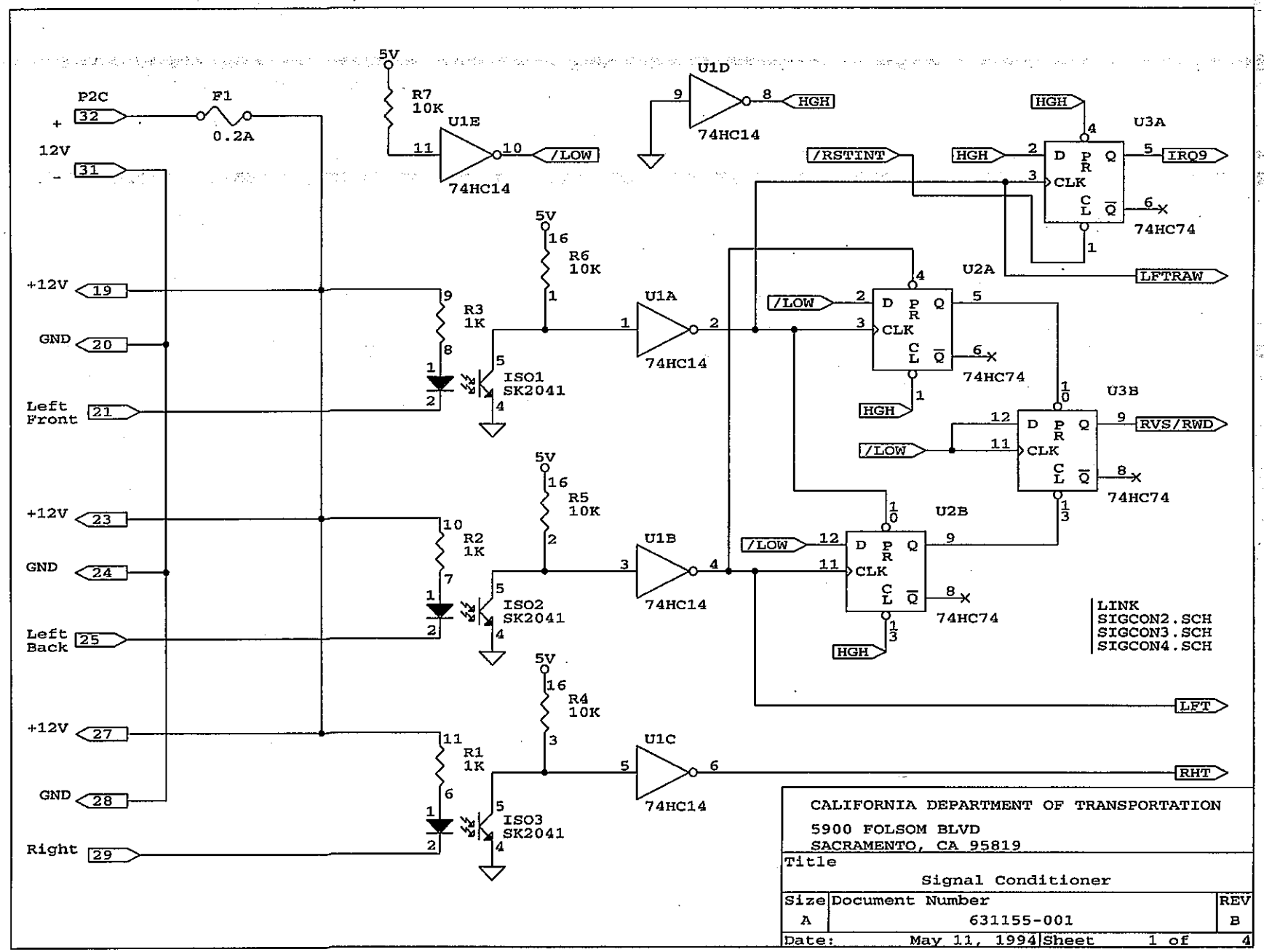
Loop Excitation Power Monitor, installed

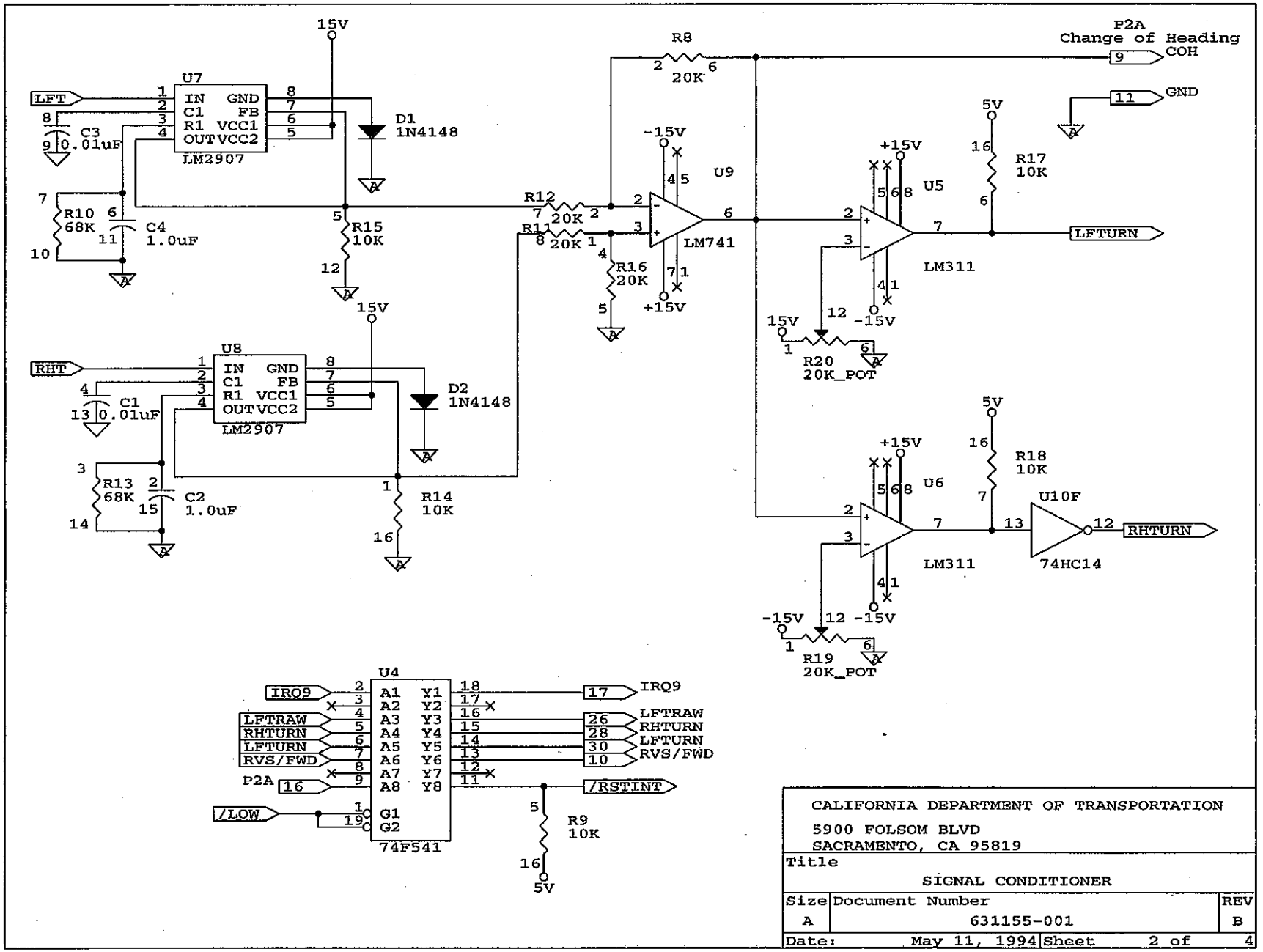
Figure A12

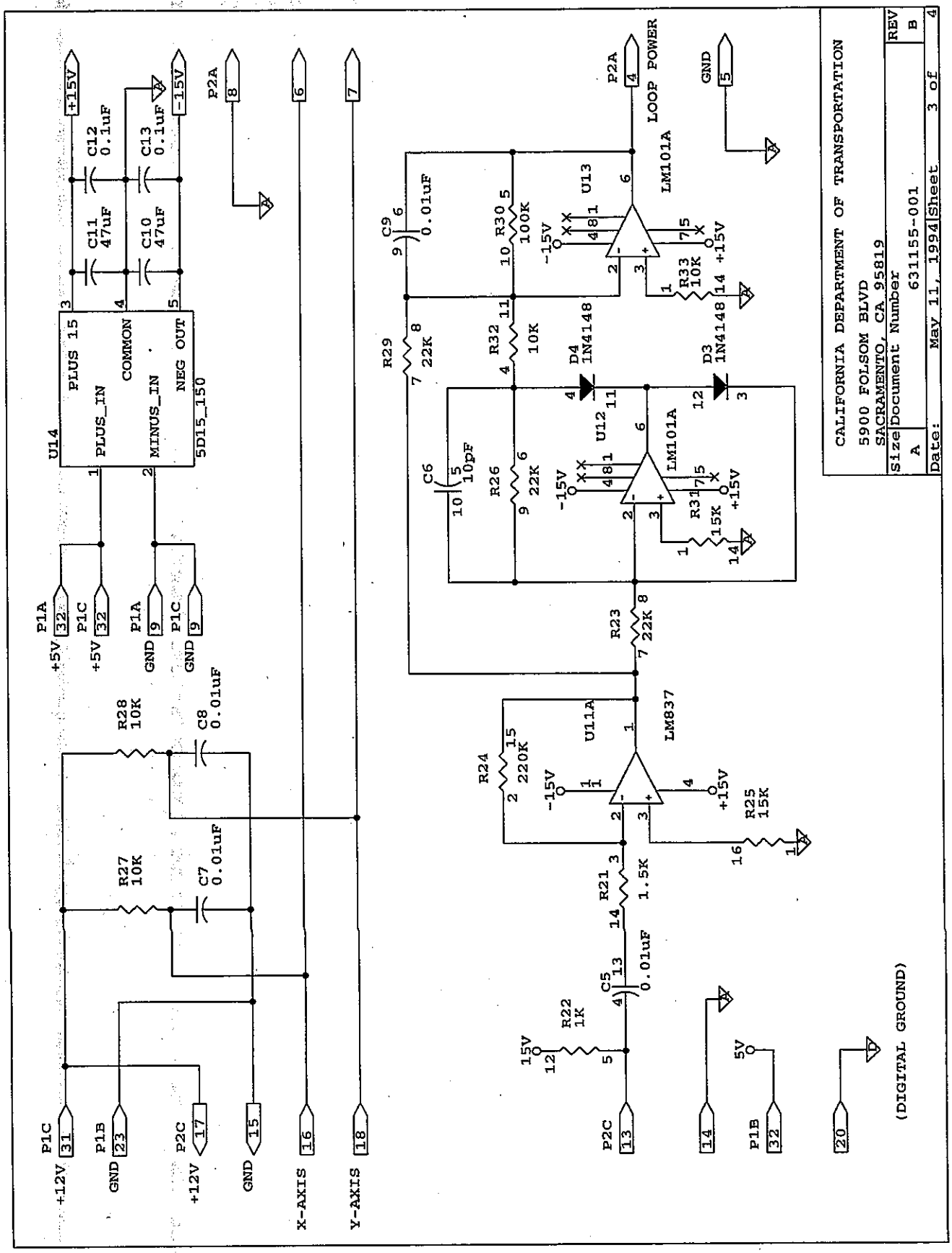


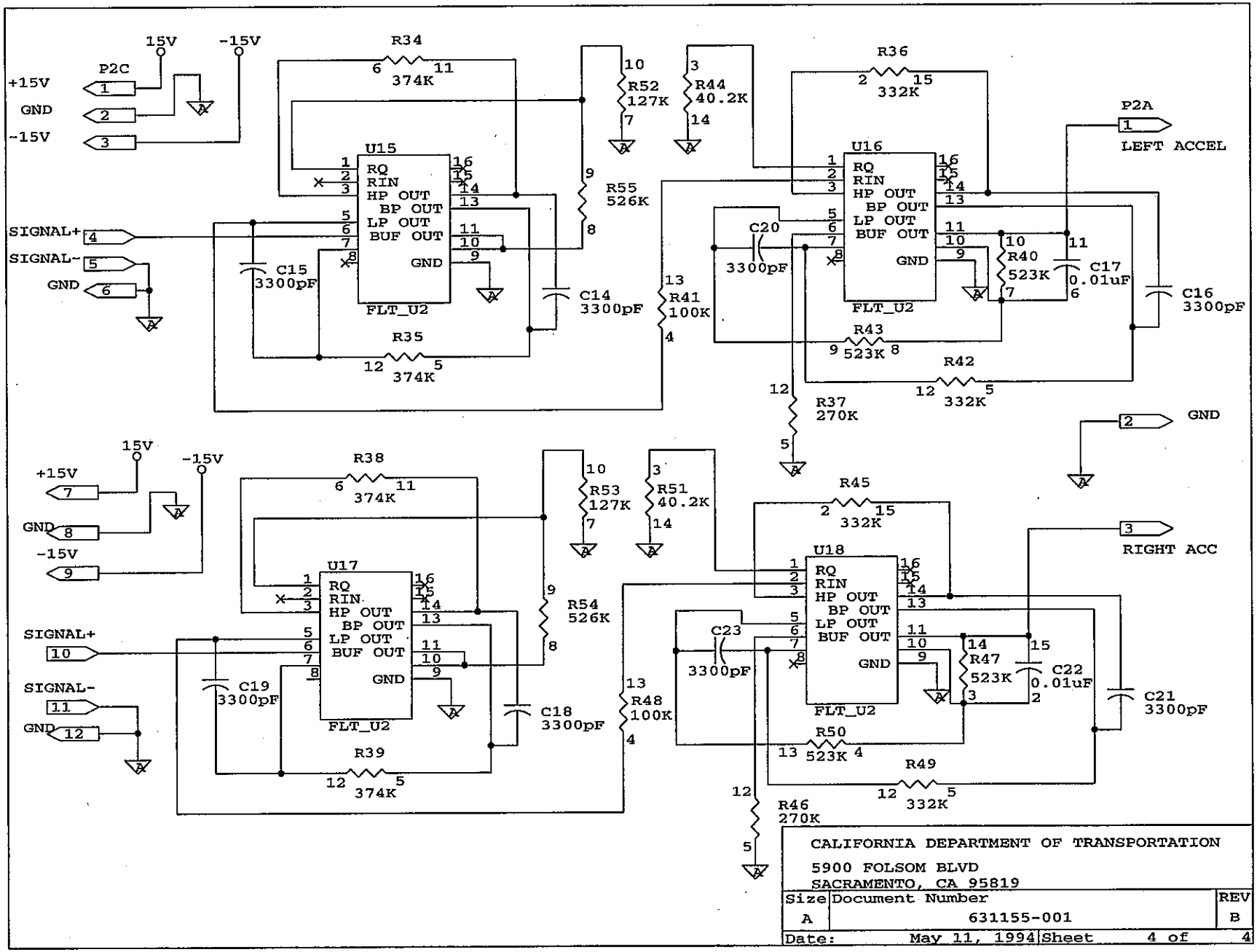
Loop Excitation Power Monitor with PVC cover removed

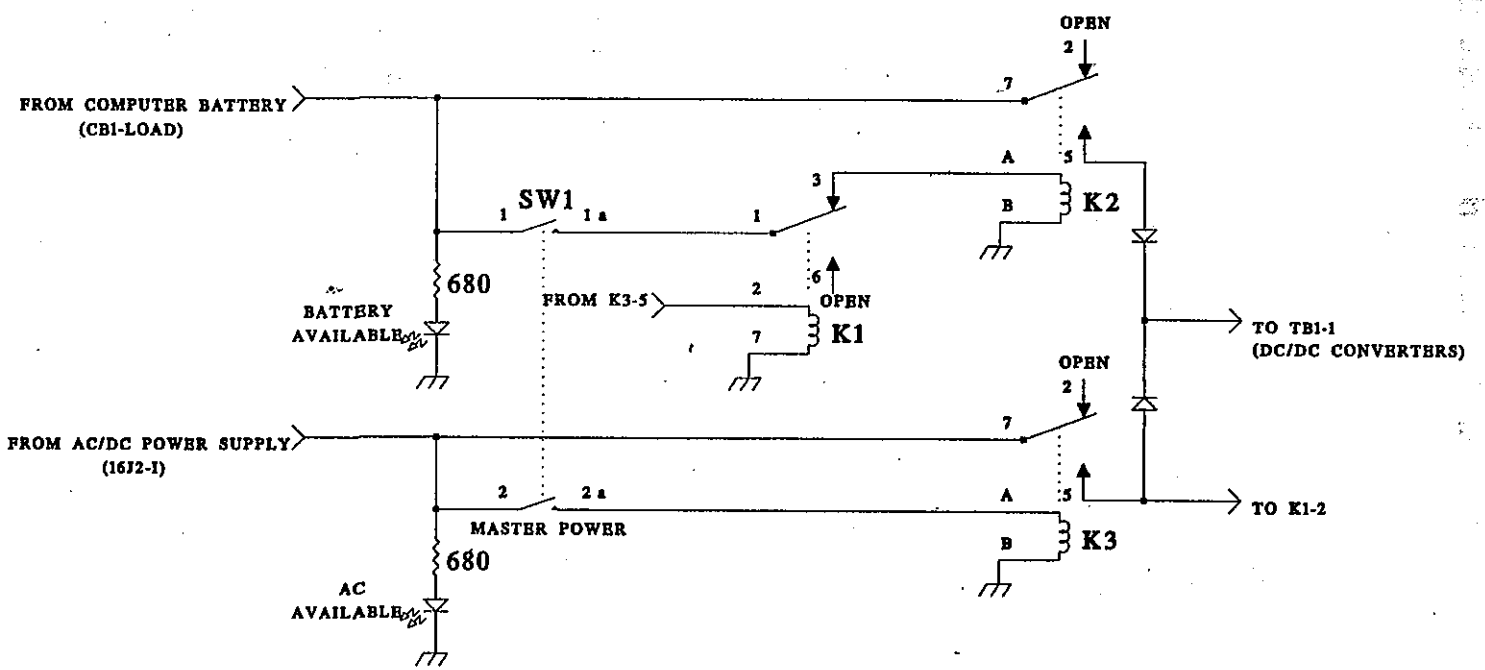
Figure A13


P2C
32
+
12V
31
-
F1
0.2A
5V
R7
10K
U1E
74HC14
/LOW
U1D
74HC14
HGH
/RSTINT
U3A
IRQ9
74HC74
LFTRAW
U2A
+12V
19
GND
20
Left
Front
21
R3
1K
ISO1
SK2041
R6
10K
U1A
74HC14
U3B
RVS/RWD
+12V
23
GND
24
Left
Back
25
R2
1K
ISO2
SK2041
R5
10K
U1B
U2B
LINK
SIGCON2.SCH
SIGCON3.SCH
SIGCON4.SCH
LFT
+12V
27
GND
28
Right
29
R1
1K
ISO3
SK2041
R4
10K
U1C
RHT
CALIFORNIA DEPARTMENT OF TRANSPORTATION
5900 FOLSOM BLVD
SACRAMENTO, CA 95819
Title
Signal Conditioner
Size
A
Document Number
631155-001
REV
B
Date:
May 11, 1994
Sheet
1 of
4

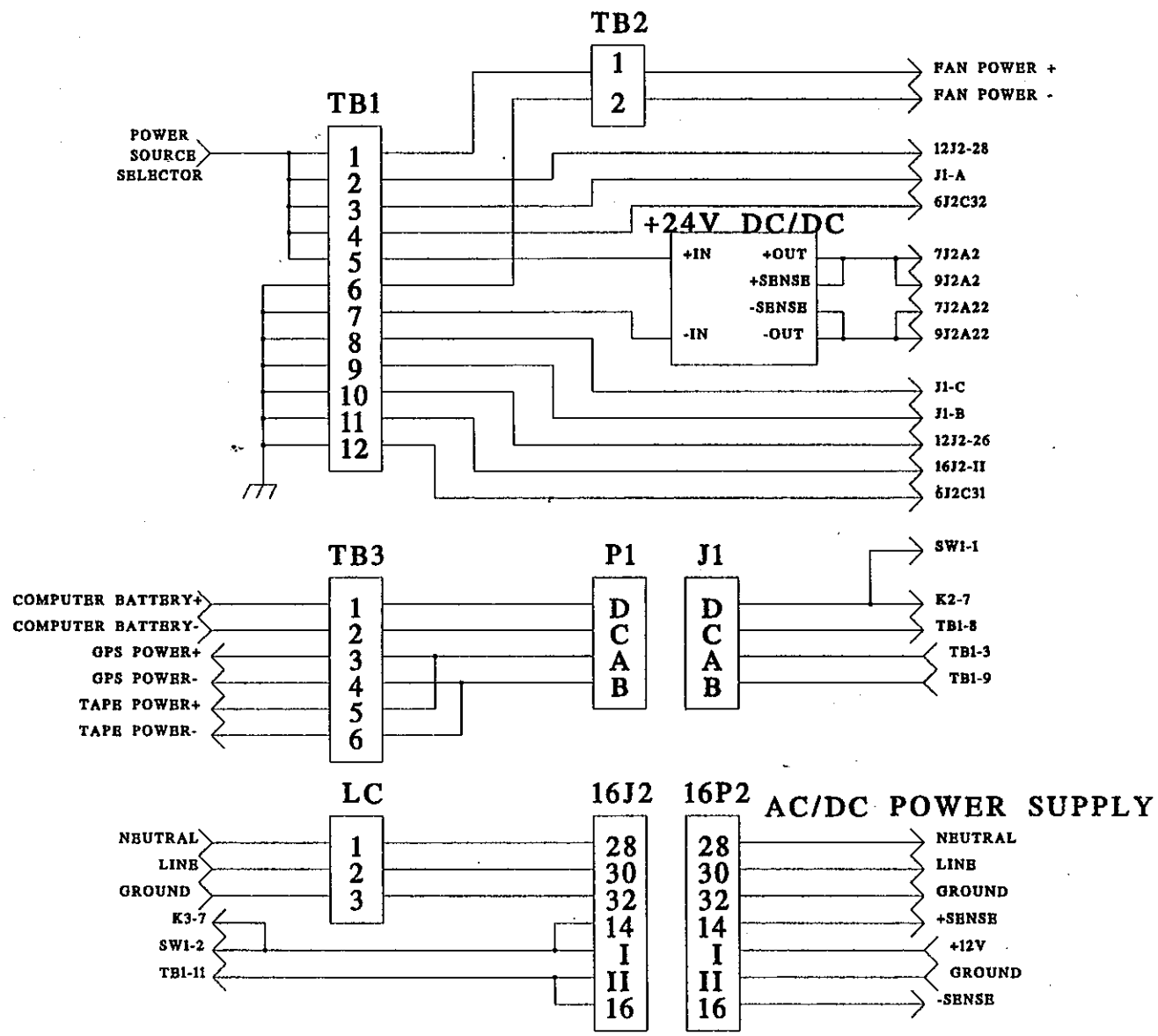

15V
U7
LFT
IN GND
C1 FB
R1 VCC1
OUT VCC2
LM2907
C3
0.01uF
R10
68K
C4
1.0uF
D1
1N4148
R15
10K
R12
20K
R11
20K
R16
20K
R8
20K
U9
LM741
-15V
+15V
P2A
Change of Heading
COH
GND
U5
LM311
R17
10K
LFTURN
R20
20K_POT
U8
RHT
C1
0.01uF
R13
68K
C2
1.0uF
D2
1N4148
R14
10K
U6
R18
10K
U10F
74HC14
RHTURN
R19
20K_POT
U4
IRQ9
LFTRAW
RHTURN
LFTURN
RVS/FWD
P2A
/LOW
A1 A2 A3 A4 A5 A6 A7 A8
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
G1 G2
74F541
/RSTINT
R9
10K
5V
CALIFORNIA DEPARTMENT OF TRANSPORTATION
5900 FOLSOM BLVD
SACRAMENTO, CA 95819
Title
SIGNAL CONDITIONER
Size A
Document Number 631155-001
REV B
Date: May 11, 1994
Sheet 2 of 4

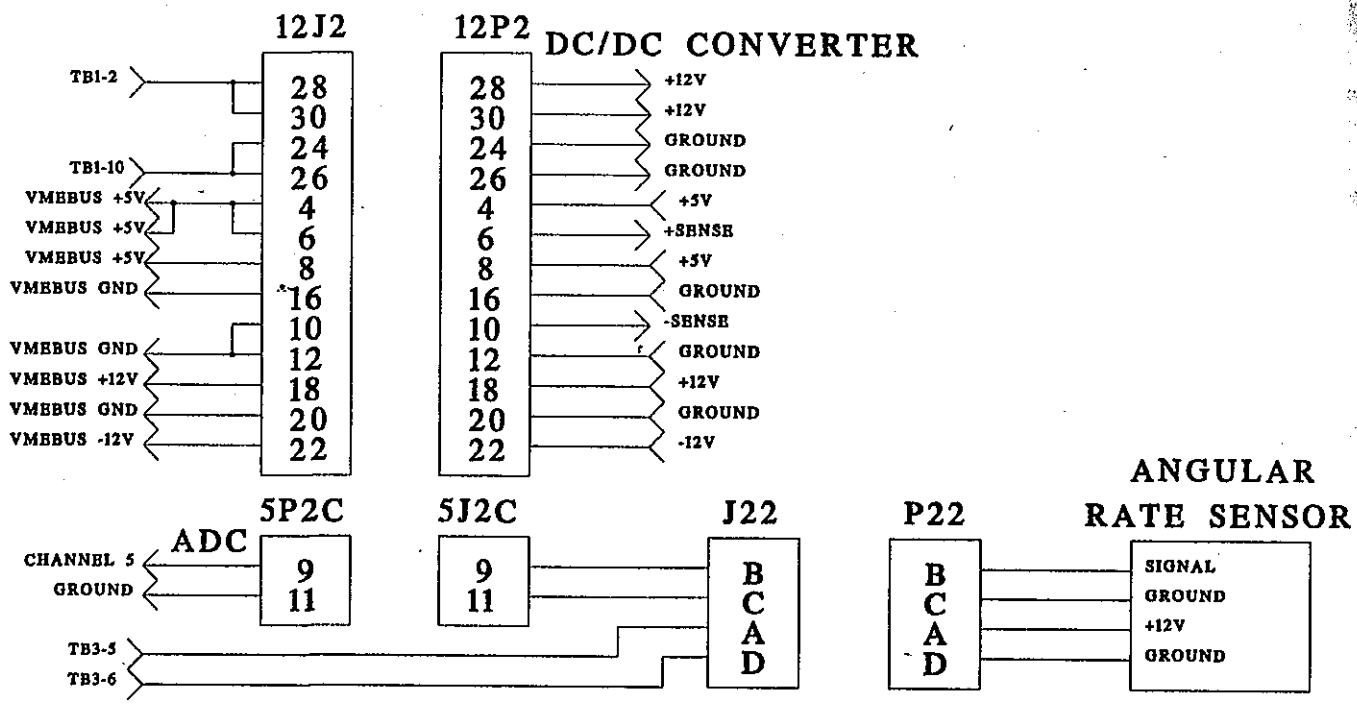

CALIFORNIA DEPARTMENT OF TRANSPORTATION
5900 FOLSOM BLVD
SACRAMENTO, CA 95819
Size A
Document Number 631155-001
REV B
Date: May 11, 1994
Sheet 3 of 4
U14
5D15_150
PLUS 15
PLUS_IN
COMMON
MINUS_IN
NEG OUT
+15V
-15V
C11 47uF
C12 0.1uF
C13 0.1uF
C10 47uF
P2A
LOOP POWER
GND
LM101A
U13
U12
U11A
LM837
R29 22K
R30 100K
R32 10K
R33 10K
R26 22K
R31 15K
R23 22K
R24 220K
R25 15K
R21 1.5K
R22 1K
C9 0.01uF
C6 10pF
D4 1N4148
D3 1N4148
C5 0.01uF
P1A +5V
P1C +5V
P1A GND
P1C GND
R28 10K
C8 0.01uF
R27 10K
C7 0.01uF
P1C +12V
P1B GND
P2C +12V
GND
X-AXIS
Y-AXIS
P2C
P1B
(DIGITAL GROUND)

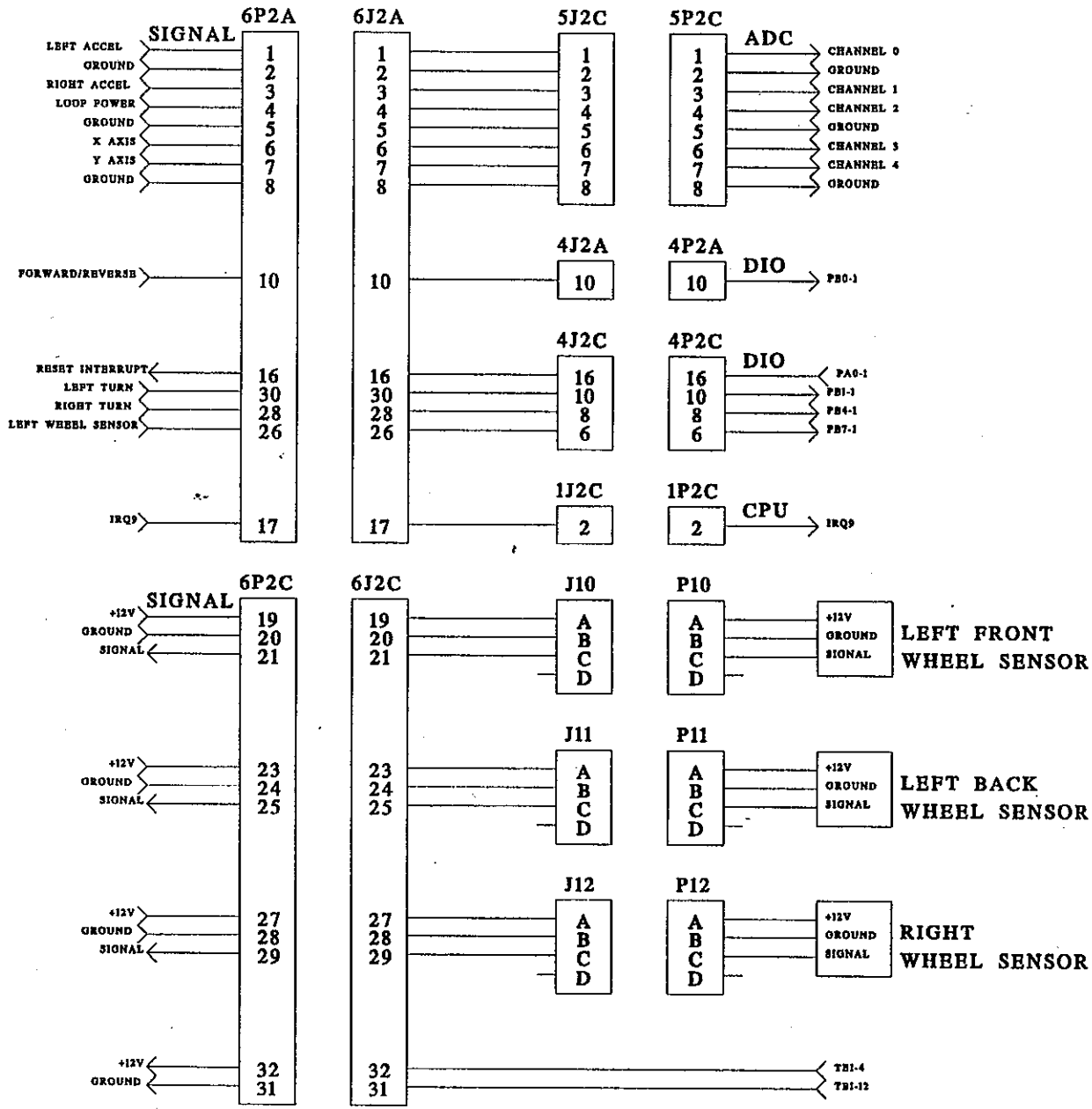

P2C
+15V
GND
-15V
15V
-15V
R34
374K
U15
RQ
RIN
HP OUT
BP OUT
LP OUT
BUF OUT
GND
FLT_U2
SIGNAL+
SIGNAL-
C15
3300pF
R35
374K
R52
127K
R44
40.2K
R55
526K
C14
3300pF
R41
100K
R36
332K
U16
C20
3300pF
R40
523K
C17
0.01uF
R43
523K
R42
332K
R37
270K
P2A
LEFT ACCEL
C16
3300pF
GND
R38
374K
U17
C19
3300pF
R39
374K
R53
127K
R51
40.2K
R54
526K
C18
3300pF
R48
100K
R45
332K
U18
C23
3300pF
R47
523K
C22
0.01uF
R50
523K
R49
332K
R46
270K
RIGHT ACC
C21
3300pF
CALIFORNIA DEPARTMENT OF TRANSPORTATION
5900 FOLSOM BLVD
SACRAMENTO, CA 95819
Size A
Document Number 631155-001
REV B
Date: May 11, 1994
Sheet 4 of 4

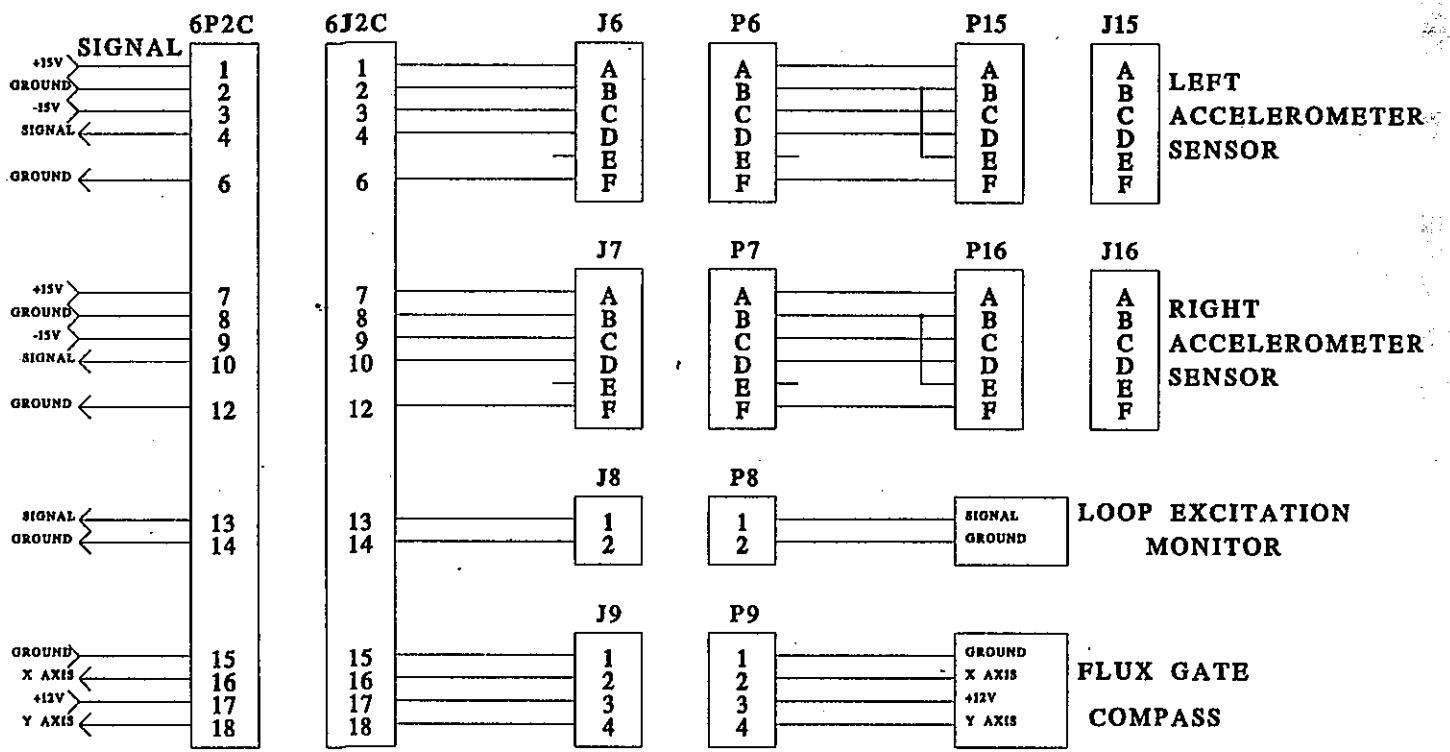


POWER SOURCE SELECTOR


TB2
1
2
FAN POWER +
FAN POWER -
TB1
POWER SOURCE SELECTOR
1
2
3
4
5
6
7
8
9
10
11
12
12J2-28
J1-A
6J2C32
+24V DC/DC
+IN
+OUT
+SENSE
-SENSE
-IN
-OUT
7J2A2
9J2A2
7J2A22
9J2A22
J1-C
J1-B
12J2-26
16J2-II
6J2C31
TB3
COMPUTER BATTERY+
COMPUTER BATTERY-
GPS POWER+
GPS POWER-
TAPE POWER+
TAPE POWER-
1
2
3
4
5
6
P1
D
C
A
B
J1
D
C
A
B
SW1-1
K2-7
TB1-8
TB1-3
TB1-9
LC
NEUTRAL
LINE
GROUND
K3-7
SW1-2
TB1-11
1
2
3
16J2
28
30
32
14
I
II
16
16P2
28
30
32
14
I
II
16
AC/DC POWER SUPPLY
NEUTRAL
LINE
GROUND
+SENSE
+12V
GROUND
-SENSE


12J2
12P2
DC/DC CONVERTER
TB1-2
TB1-10
VMEBUS +5V
VMEBUS +5V
VMEBUS +5V
VMEBUS GND
VMEBUS GND
VMEBUS +12V
VMEBUS GND
VMEBUS -12V
28
30
24
26
4
6
8
16
10
12
18
20
22
+12V
+12V
GROUND
GROUND
+5V
+SENSE
+5V
GROUND
-SENSE
GROUND
+12V
GROUND
-12V
ADC
5P2C
5J2C
J22
P22
ANGULAR
RATE SENSOR
CHANNEL 5
GROUND
9
11
B
C
A
D
SIGNAL
GROUND
+12V
GROUND
TB3-5
TB3-6


SIGNAL
6P2A
6J2A
5J2C
5P2C
ADC
LEFT ACCEL
GROUND
RIGHT ACCEL
LOOP POWER
GROUND
X AXIS
Y AXIS
GROUND
CHANNEL 0
GROUND
CHANNEL 1
CHANNEL 2
GROUND
CHANNEL 3
CHANNEL 4
GROUND
4J2A
4P2A
DIO
FORWARD/REVERSE
PB0-1
4J2C
4P2C
DIO
RESET INTERRUPT
LEFT TURN
RIGHT TURN
LEFT WHEEL SENSOR
PA0-1
PB1-1
PB4-1
PB7-1
1J2C
1P2C
CPU
IRQ9
IRQ9
SIGNAL
6P2C
6J2C
J10
P10
+12V
GROUND
SIGNAL
LEFT FRONT
WHEEL SENSOR
J11
P11
LEFT BACK
WHEEL SENSOR
J12
P12
RIGHT
WHEEL SENSOR
TB1-4
TB1-12


SIGNAL
6P2C
6J2C
J6
P6
P15
J15
+15V
GROUND
-15V
SIGNAL
GROUND
LEFT ACCELEROMETER SENSOR
J7
P7
P16
J16
+15V
GROUND
-15V
SIGNAL
GROUND
RIGHT ACCELEROMETER SENSOR
J8
P8
SIGNAL
GROUND
LOOP EXCITATION MONITOR
J9
P9
GROUND
X AXIS
+12V
Y AXIS
FLUX GATE COMPASS

Appendix B

Software Listings

Van Computer Software Modules	Page
DISTISR.ASM	B-2
TIMEISR.ASM	B-7
EXEC.C	B-11
DATA_ACQ.C	B-17
DISP_VEL.C	B-29
DATA_PRO.C	B-33
RAW.C	B-42
SLOPE.C	B-49
INIT_STM.C	B-54
CNVNAV.C	B-57
SYS_TEST.C	B-59
BOUNCE.C	B-61
WHEEL.C	B-65
STEP.C	B-70
HEIGHT.C	B-72
ADC.C	B-74
GPS.C	B-76
RS232.C	B-82
MOV_CURS.C	B-85
GETMSG.C	B-86

Laptop Computer Software Modules	Page
SERIAL.C	B-88
PROCESS.C	B-95

```
;
; DISTISR.ASM
; one inch interrupt service routine
; programmer: Greg Larson
;   program: PROFILE -- 631155
;   version: 01.00
;   created: 03/19/92
;   revised: 03/25/94
;
        .MODEL  COMPACT             ; compact memory model
        .386                        ; 80386 instruction set
;
        .DATA                       ; start of data segment
        EXTRN _odmcount:DWORD       ; odometer count
        EXTRN _lobuf:WORD           ; first flip-flop buffer
        EXTRN _hibuf:WORD           ; second flip-flop buffer
        EXTRN _bufptr:WORD          ; pointer into buffer
        EXTRN _velocity:WORD        ; velocity count data
        EXTRN _laccel:WORD          ; left accelerometer data
        EXTRN _raccel:WORD          ; right accelerometer data
        EXTRN _lheight:WORD         ; left height sensor data
        EXTRN _rheight:WORD         ; right height sensor data
        EXTRN _laccsum:WORD         ; left accel partial sum
        EXTRN _raccsum:WORD         ; right accel partial sum
        EXTRN _lhgtsum:WORD         ; left height partial sum
        EXTRN _rhgtsum:WORD         ; right height partial sum
        EXTRN _bufflag:BYTE         ; if lo, use lobuf; if hi, use hibuf
        EXTRN _fullflag:BYTE        ; buffer is full flag
        EXTRN _intcount:BYTE        ; interval count, up to twelve
_DATA ENDS                          ; end of data segment
;
;
REALWIN     EQU     0E000h          ; address of real mode window
RESET       EQU     0D011h          ; dio reset interrupt register offset
STAT        EQU     0D013h          ; dio status register offset
TMRCNTL     EQU     0D061h          ; dio timer 2 control register offset
TMRMI       EQU     0D071h          ; dio timer 2 count register offset
TMRLO       EQU     0D073h          ; dio timer 2 count register offset
ADCSTAT     EQU     0E081h          ; adc status register offset
ADCCNTL     EQU     0E081h          ; adc control register offset
ADCCHAN     EQU     0E085h          ; adc channel register offset
ADCDATA     EQU     0E086h          ; adc data register offset
LEFTHGT     EQU     0E400h          ; left height sensor data reg offset
RGHTHGT     EQU     0E800h          ; right height sensor data reg offset
BUFLEN      EQU     014A0h          ; profile data buffer length, 5280
;                                   ;   528 samples * 10 bytes/sample
;
        .CODE                       ; start of code segment
        PUBLIC _distisr             ; external definition
_distisr PROC   FAR                 ; one inch ISR
        sti                         ; enable interrupts
        push    ax                  ; save registers
        push    ds                  ;
        push    es                  ;
```

```
;
        mov     ax,DGROUP               ; point to data segment
        mov     ds,ax                   ; initialize data segment
        mov     ax,REALWIN              ; get address of real mode window
        mov     es,ax                   ; point to real mode window
;
; turn off the source of the distance-based interrupt request
;
        mov     BYTE PTR es:[RESET],0 ; reset interrupt request
        mov     BYTE PTR es:[RESET],1 ; un-reset interrupt request
;
; increment or decrement odometer count
;
        add     _odmcount,1             ; increment odometer count for now
        mov     al,es:[STAT]            ; get status byte
        rcr     al,1                    ; check for vehicle moving forward
        jmp     start
        jnc     start                   ; branch if vehicle moving forward
        sub     _odmcount,2             ; decrement odometer count
;
; get the vehicle velocity count
;
start:  cli                             ; disable interrupts
        mov     BYTE PTR es:[TMRCNTL],010h  ; stop timer
        mov     ah,es:[TMRMI]           ; get middle byte of timer
        mov     al,es:[TMRLO]           ; get low byte of timer
        mov     BYTE PTR es:[TMRCNTL],011h  ; re-start timer
        sti                             ; re-enable interrupts
        mov     _velocity,ax            ; save velocity count
;
; acquire left and right wheelpath data
;
        mov     BYTE PTR es:[ADCCHAN],0       ; select channel 0 on A/D
        mov     BYTE PTR es:[ADCCNTL],0A0h ; start sequential conversion
;
; get the left height sensor data while waiting for the left accelerometer
;
        mov     ax,es:[LEFTHGT]         ; get left height sensor data
        test    ax,ax                   ; check for valid data
        jns     lhgt                    ; branch if valid data
        mov     ax,_lheight             ; get last valid data instead
lhgt:   and     ax,0FFFh                ; mask off data bits
        mov     _lheight,ax             ; save left height sensor data
;
; now get the left accelerometer data
;
busy0:  mov     al,es:[ADCSTAT]         ; get adc status
        rcl     al,1                    ; check for adc busy
        jc      busy0                   ; branch if adc busy
;
        mov     ax,es:[ADCDATA]         ; get adc data
        mov     _laccel,ax              ; save left accelerometer data
;
; get the right height sensor data while waiting for the right accelerometer
```

```
;
        mov     ax,es:[RGHTHGT]         ; get right height sensor data
        test    ax,ax                   ; check for valid data
        jns     rhgt                    ; branch if valid data
        mov     ax,_rheight             ; get last valid data instead
rhgt:   and     ax,0FFFh                ; mask off data bits
        mov     _rheight,ax             ; save right height sensor data
;
; now get the right accelerometer data
;
busy1:  mov     al,es:[ADCSTAT]         ; get adc status
        rcl     al,1                    ; check for adc busy
        jc      busy1                   ; branch if adc busy
;
        mov     ax,es:[ADCDATA]         ; get adc data
        mov     _raccel,ax              ; save right accelerometer data
;
; compute the partial sums to prepare for averaging
;
        mov     ax,_laccsum             ; get left accel partial sum
        add     ax,_laccel              ; add left accel data
        mov     _laccsum,ax             ; save left accel partial sum
;
        mov     ax,_lhgtsum             ; get left height partial sum
        add     ax,_lheight             ; add left height data
        mov     _lhgtsum,ax             ; save left height partial sum
;
        mov     ax,_raccsum             ; get right accel partial sum
        add     ax,_raccel              ; add right accel data
        mov     _raccsum,ax             ; save right accel partial sum
;
        mov     ax,_rhgtsum             ; get right height partial sum
        add     ax,_rheight             ; add right height data
        mov     _rhgtsum,ax             ; save right height partial sum
;
; if the van has travelled one foot (twelve samples), save the averaged data
; otherwise, return to the main program
;
        inc     _intcount               ; increment interval count
        cmp     _intcount,12            ; check for one foot travelled
        jbe     done                    ; branch if one foot NOT travelled
        mov     _intcount, 1            ; re-initialize interval count
;
; after every foot of travel, save one averaged sample
;
        push    bx                      ; save registers
        push    cx                      ;
        push    dx                      ;
        push    di                      ;
;
; determine which profile buffer to use
;
        mov     di,_bufptr              ; get pointer into buffer
        mov     bx,OFFSET _lobuf        ; point to lo buffer for now
```

```
        cmp     _bufflag,0          ; check for lo buffer to be used
        je      nothi               ; branch if lo buffer to be used
        mov     bx,OFFSET _hibuf    ; point to hi buffer
;
; save the most recent vehicle velocity count
;
nothi:  mov     ax,_velocity        ; get the velocity count
        not     ax                  ; make counts positive
        mov     [bx][di],ax         ; save the velocity count in buffer
        add     di,2                ; point to next available location
;
        mov     cx,12               ; get interval divisor
;
; calculate left accelerometer data
;
        xor     dx,dx               ; clear upper word of dividend
        mov     ax,_laccsum         ; get left accel sum, lower word
        div     cx                  ; calculate average, cx has divisor
        mov     [bx][di],ax         ; save data in buffer
        add     di,2                ; point to next available location
        mov     _laccsum,0          ; clear partial sum variable
;
; calculate left height sensor data
;
        xor     dx,dx               ; clear upper word of dividend
        mov     ax,_lhgtsum         ; get left height sum, lower word
        div     cx                  ; calculate average, cx has divisor
        mov     [bx][di],ax         ; save data in buffer
        add     di,2                ; point to next available location
        mov     _lhgtsum,0          ; clear partial sum variable
;
; calculate right accelerometer data
;
        xor     dx,dx               ; clear upper word of dividend
        mov     ax,_raccsum         ; get right accel sum, lower word
        div     cx                  ; calculate average, cx has divisor
        mov     [bx][di],ax         ; save data in buffer
        add     di,2                ; point to next available location
        mov     _raccsum,0          ; clear partial sum variable
;
; calculate right height sensor data
;
        xor     dx,dx               ; clear upper word of dividend
        mov     ax,_rhgtsum         ; get right height sum, lower word
        div     cx                  ; calculate average, cx has divisor
        mov     [bx][di],ax         ; save data in buffer
        add     di,2                ; point to next available location
        mov     _rhgtsum,0          ; clear partial sum variable
;
; update buffer pointer, check for end of buffer
;
        mov     _bufptr,di          ; save offset into buffer
        cmp     di,BUFLEN           ; check for end of buffer (5280)
        jb      notend              ; branch if not end of buffer
```

```
        mov     _bufptr,0           ; re-initialize offset into buffer
        not     _bufflag            ; use other buffer now (flip-flop)
        mov     _fullflag,0FFh      ; say process full buffer
;
notend: pop     di                  ; restore registers
        pop     dx                  ;
        pop     cx                  ;
        pop     bx                  ;
;
done:   mov     al,020h             ; get end of interrupt command
        cli                         ; disable interrupts
        out     0A0h,al             ; send end of interrupt to slave PIC
        out     020h,al             ; send end of interrupt to master PIC
;
        pop     es                  ; restore registers
        pop     ds                  ;
        pop     ax                  ;
        iret                        ; return from interrupt
;
_distisr ENDP                       ; end of distisr procedure
;
_TEXT  ENDS                         ; end of code segment
;
        END                         ; end of distisr.asm
```

```
;
;
; TIMEISR.ASM
; one millisecond interrupt service routine
; programmer: Greg Larson
;   project: PROFILE -- 631155
;   version: 01.00
;   created: 03/18/92
;   revised: 06/15/93
;
        .MODEL      COMPACT     ; compact memory model
        .386                    ; 80386 instruction set
;
        .DATA                   ; start of data segment
        EXTRN _evnbuf:WORD      ; nav data buffer for even seconds
        EXTRN _oddbuf:WORD      ; nav data buffer for odd seconds
        EXTRN _navptr:WORD      ; pointer into evn and odd buffers
        EXTRN _power:WORD       ; loop power variable
        EXTRN _xaxis:WORD       ; flux gate compass, x axis
        EXTRN _yaxis:WORD       ; flux gate compass, y axis
        EXTRN _rate:WORD        ; angular rate data
        EXTRN _second:WORD      ; seconds count
        EXTRN _threshold:WORD   ; loop power level threshold
        EXTRN _loopdelay:WORD   ; one second delay count
        EXTRN _looptime:WORD    ; time when loop was detected, seconds
        EXTRN _loopind:WORD     ; 4*index at which loop was detected
        EXTRN _gpsque:WORD      ; offset to gps data queue
        EXTRN _gpshead:WORD     ; gps queue head pointer
        EXTRN _dispcnt:WORD     ; # of seconds between screen paints
        EXTRN _dcount:WORD      ; screen display interval
        EXTRN _loopflag:BYTE    ; loop detected flag
        EXTRN _dispflag:BYTE    ; display data flag
        EXTRN _navflag:BYTE     ; process navigation flag
        EXTRN _gpsflag:BYTE     ; process gps flag
_DATA ENDS                      ; end of data segment
;
;
STATUS      EQU     02FDh       ; COM2 line status register address
DATA        EQU     02F8h       ; COM2 data register address
;
REALWIN     EQU     0E000h      ; address of real mode window
ADCSTAT     EQU     0E081h      ; adc status register offset
ADCCNTL     EQU     0E081h      ; adc control register offset
ADCCHAN     EQU     0E085h      ; adc channel register offset
ADCDATA     EQU     0E086h      ; adc data register offset
TMRSTAT     EQU     0D035h      ; dio timer 1 status register offset
BUFLEN      EQU     01770h      ; length of even, odd buffers (6*1000)
;
        .CODE                   ; start of code segment
        PUBLIC _timeisr         ; external definition
_timeisr PROC   FAR             ; one millisecond ISR
        sti                     ; enable interrupts
        push    ax              ; save registers
        push    bx              ;
```

```
        push    dx                      ;
        push    di                      ;
        push    ds                      ; save data segment
        push    es                      ; save extra segment
;
        mov     ax,DGROUP               ; point to data segment
        mov     ds,ax                   ; initialize data segment
        mov     ax,REALWIN              ; get address of real mode window
        mov     es,ax                   ; point to real mode window
        mov     BYTE PTR es:[TMRSTAT],1         ; reset IRQ bit in timer status reg
;
; Check for GPS receiver data and install it in the queue
;
        mov     dx,STATUS               ; get COM2 line status register address
        in      al,dx                   ; get COM2 line status data
        rcr     al,1                    ; check for GPS data available
        jnc     strtad                  ; branch if no GPS data available
;
        mov     dx,DATA                 ; get COM2 data register address
        in      al,dx                   ; get COM2 data
        mov     bx,OFFSET _gpsque       ; point to GPS data queue
        mov     di,_gpshead             ; get queue head pointer
        mov     [bx][di],al             ; save GPS data in queue
        inc     di                      ; increment head pointer
        and     di,0FFh                 ; check for end of queue (256 long)
        mov     _gpshead,di             ; update head pointer
        cmp     al,0Ah                  ; check for a LF (complete message)
        jne     strtad                  ; branch if not a LF
        mov     _gpsflag,0FFh           ; say process GPS message
;
; start sequential a/d conversion, channels 2 - 5
;
strtad: mov     bx,ADCSTAT              ; point to adc status register
        cli                             ; disable interrupts
        mov     BYTE PTR es:[ADCCHAN],2         ; select channel 2
        mov     BYTE PTR es:[ADCCNTL],0A0h ; start sequential conversion
;
busy2:  mov     al,es:[bx]              ; get adc status, channel 2
        rcl     al,1                    ; check for adc busy
        jc      busy2                   ; branch if adc busy
;
        mov     ax,es:[ADCDATA]         ; get adc data, start next conversion
        mov     _power,ax               ; save loop power data
;
busy3:  mov     al,es:[bx]              ; get adc status, channel 3
        rcl     al,1                    ; check for adc busy
        jc      busy3                   ; branch if adc busy
;
        mov     ax,es:[ADCDATA]         ; get adc data, start next conversion
        mov     _xaxis,ax               ; save flux gate compass x axis
;
busy4:  mov     al,es:[bx]              ; get adc status, channel 4
        rcl     al,1                    ; check for adc busy
        jc      busy4                   ; branch if adc busy
```

```
;
        mov     ax,es:[ADCDATA]         ; get adc data, start next conversion
        mov     _yaxis,ax               ; save flux gate compass y axis
;
busy5:  mov     al,es:[bx]              ; get adc status, channel 5
        rcl     al,1                    ; check for adc busy
        jc      busy5                   ; branch if adc busy
;
        mov     ax,es:[ADCDATA]         ; get adc data, last channel
        sti                             ; re-enable interrupts
        mov     _rate,ax                ; save angular rate data
;
; check for loop detected

        cmp     _loopdelay,0            ; check for loop already detected
        je      chkpwr                  ; branch if loop not already detected
        dec     _loopdelay              ; decrement one second delay count
        jmp     donav                   ; do navigation
chkpwr: mov     ax,_power               ; get loop excitation monitor power
        cmp     ax,_threshold           ; check for loop detected
        jb      donav                   ; branch if loop not detected
        mov     _loopflag,0FFh          ; say loop detected
        mov     _loopdelay,1000         ; initialize one second delay count
        mov     ax,_second              ; get time at which loop was detected
        mov     _looptime,ax            ; save time at which loop was detected
        mov     ax,_navptr              ; get pointer into buffer
        mov     _loopind,ax             ; save 6*index of detected loop
;
; the following instructions store navigation data in either of two
; buffers, depending on whether the seconds count is even or odd
;
donav:  mov     di,_navptr              ; get pointer into buffer
        mov     bx,OFFSET _evnbuf       ; point to even buffer for now
        mov     ax,_second              ; get seconds count
        rcr     al,1                    ; check for even number of seconds
        jnc     compas                  ; branch if even number of seconds
        mov     bx,OFFSET _oddbuf       ; point to odd buffer
;
compas: mov     ax,_xaxis               ; get flux gate compass x axis data
        mov     [bx][di],ax             ; save x axis data in buffer
        add     di,2                    ; point to next available location
        mov     ax,_yaxis               ; get flux gate compass y axis data
        mov     [bx][di],ax             ; save y axis data in buffer
        add     di,2                    ; point to next available location
        mov     ax,_rate                ; get angular rate data
        mov     [bx][di],ax             ; save angular rate data in buffer
        add     di,2                    ; point to next available location
;
        mov     _navptr,di              ; save buffer pointer
        cmp     di,BUFLEN               ; check for buffer full, 6*1000 msec
        jb      chkgps                  ; branch if buffer not full
        mov     _navptr,0               ; re-initialize buffer pointer
        inc     _second                 ; update seconds count
        mov     _navflag,0FFh           ; say process navigation data
```

```
        dec     _dispcnt                ; check for time to display
        jne     chkgps                  ; branch if not time to display
        mov     _dispflag,0FFh          ; say display data
        mov     ax,_dcount              ; get re-initialize display count value
        mov     _dispcnt,ax             ; save re-initialize value
;
; Check for GPS receiver data and install it in the queue (AGAIN!)
;
chkgps: mov     dx,STATUS               ; get COM2 line status register address
        in      al,dx                   ; get COM2 line status data
        rcr     al,1                    ; check for GPS data available
        jnc     nodata                  ; branch if no GPS data available
;
        mov     dx,DATA                 ; get COM2 data register address
        in      al,dx                   ; get COM2 data
        mov     bx,OFFSET _gpsque       ; point to GPS data queue
        mov     di,_gpshead             ; get queue head pointer
        mov     [bx][di],al             ; save GPS data in queue
        inc     di                      ; increment head pointer
        and     di,0FFh                 ; check for end of queue (256 long)
        mov     _gpshead,di             ; update head pointer
        cmp     al,0Ah                  ; check for a LF (complete message)
        jne     nodata                  ; branch if not a LF
        mov     _gpsflag,0FFh           ; say process GPS message
;
nodata: mov     al,020h                 ; get end of interrupt command
        cli                             ; disable interrupts
        out     0A0h,al                 ; send end of interrupt to slave PIC
        out     020h,al                 ; send end of interrupt to master PIC
;
        pop     es                      ; restore extra segment
        pop     ds                      ; restore data segment
        pop     di                      ; restore registers
        pop     dx                      ;
        pop     bx                      ;
        pop     ax                      ;
        iret                            ; return from interrupt
_timeisr ENDP                           ; end of timeisr procedure
;
_TEXT ENDS                              ; end of code segment
;
        END                             ; end of timeisr.asm
;
```

```
// exec.c
// main source code to the High Speed Pavement Profilometer Project
//
// Description          :     This program is executed after the computer
//                                  has been turned on or reset
//
// Functions Included   :     main()
//
// External Functions   :     data_acq(), disp_vel(), data_pro(), sys_test()
//
// Programmer           :     Greg Larson
//
// Created              :     10/30/91
//
// Last Revised         :     06/01/94
//
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <dos.h>
#include <io.h>
#include <bios.h>
#include <graphics.h>

#define DATA_ACQ     10
#define DISP_VEL     12
#define DATA_PRO     14
#define SYS_TEST     16
#define EXIT_DOS     18
#define PROMPT       20

#define STDOUT       1
#define STDAUX       3

struct probuf {                                  // profile data buffer
        unsigned int velcount;                   // velocity count data
        unsigned int laccel;                     // left accelerometer data
        unsigned int lheight;                    // left height data
        unsigned int raccel;                     // right accelerometer data
        unsigned int rheight;                    // right height data
        } lobuf[528], hibuf[528];                // lo, hi profile data buffers

// 528 samples = 0.1 mile of data

struct navbuf {                                  // navigation data buffer
        unsigned int x_axis;                     // x-axis data
        unsigned int y_axis;                     // y-axis data
        unsigned int rate;                       // angular rate data
        } evnbuf[1000], oddbuf[1000];            // even, odd seconds buffer

// 1000 samples = 1.0 seconds of data

char gpsque[256];                                // GPS queue
```

```
unsigned int gpshead;                    // GPS queue head pointer
unsigned int gpstail;                    // GPS queue tail pointer

float counts_mile;                       // # of odometer counts per mile
unsigned long odmcount, oldodmcount;     // current, old odometer count
unsigned long loopodm;                   // odometer count at last loop

unsigned long sum;                       // velocity sum for bounce
unsigned long millisec;                  // millisecond count for bounce

unsigned int bufptr;                     // pointer into profile buffer
unsigned int navptr;                     // pointer into nav buffers

unsigned int velocity;                   // velocity count
unsigned int laccel, raccel;             // left & right accelerometer
unsigned int lheight, rheight;           // left & right height
unsigned int power;                      // loop excitation monitor power
unsigned int xaxis, yaxis;               // flux gate compass axes
unsigned int rate;                       // angular rate data

unsigned int second;                     // seconds count variable
unsigned int loopdelay = 1000;           // one second delay count var
unsigned int loopind = 0;                // 6*index where loop detected
unsigned int looptime = 0;               // time when loop was detected
unsigned int threshold = 8191;           // loop power threshold, 3.5 V
unsigned int dcount = 1;                 // display interval counter
unsigned int dispcnt;                    // # of seconds between screen

unsigned int laccsum, raccsum;           // left, right accel sums
unsigned int lhgtsum, rhgtsum;           // left, right height sums

unsigned char intcount;                  // interval counter
unsigned char bufflag;                   // which profile buffer flag
unsigned char fullflag;                  // profile buffer full flag
unsigned char loopflag;                  // loop detected flag
unsigned char dispflag;                  // display data flag
unsigned char navflag;                   // nav buffer ready flag
unsigned char gpsflag;                   // GPS message ready flag
unsigned char countup;                   // PM increasing flag
unsigned char laptop;                    // flag for laptop in use
/*****************************************************************************/
void main(argc, argv)                    //
int argc;                                // argument count
char *argv[];                            // pointer to arguments
{                                        // start of main
int oldstdout;                           // file handle
FILE *fp_odo;                            // odometer count file pointer
int ch;                                  // keyboard input
unsigned char linenum;                   // line where cursor is
char i;                                  // loop index
char counts[8];                          // buffer for # of counts/mile
char mask;                               // disable interrupt mask
char far *gencntrl;                      // pointer to general control
char far *portAddr;                      // pointer to data direction
```

```
char far *portBddr;                          // pointer to data direction
char far *portCddr;                          // pointer to data direction
char far *portAcntl;                         // pointer to control register
char far *portBcntl;                         // pointer to control register
char far *portCdata;                         // ptr to port C data reg
char far *resetint;                          // ptr to reset IRQ9 reg
char far *control, *gain;                    // ptr to adc control, gain

int gdriver = VGA;                           // use VGA driver
int gmode = VGAHI;                           // use high resolution mode

int cee[18] = {120, 8, 312, 8, 299, 72, 203, 72, 183, 168, 279, 168,
               266, 232, 74, 232, 120, 8};
int tee[34] = {328, 8, 424, 8, 408, 88, 504, 88, 491, 152, 395, 152, 379, 232,
               395, 232, 408, 168, 488, 168, 462, 296, 274, 296, 299, 152,
               203, 152, 216, 88, 312, 88, 328, 8};
//
// check the command line arguments
//
laptop = 0;                                  // say no laptop
if(argc > 1) {                               // check for laptop
   strlwr(argv[1]);                          // convert to lower case
   if(strcmp(argv[1], "laptop")) {           // check for incorrect argument
         printf("Incorrect command line argument, please try again!\n");
         exit(0);                            // return to DOS
         }                                   // end of if
   else{                                     // redirect output to laptop
         laptop = 0xFF;                      // say use laptop
         init_COM1();                        // initialize COM1: port
         oldstdout = dup(STDOUT);            // duplicate stdout handle
         dup2(STDAUX, STDOUT);                       // copy handle into stdout
         }                                   // end of else
   }                                         // end of if
if(argc > 2) {                               // check for time between screen
   dcount = atoi(argv[2]);                   // get command line argument
   if(dcount == 0 || dcount > 10)            // check for command line error
         dcount = 1;                         // re-initialize
   }                                         // end of if
//
// initialize the number of odometer counts per mile
//
counts_mile = 63360.0;                       // default value
if((fp_odo = fopen("ODOMETER.TXT", "rt+")) != NULL) {  // open odometer file
   fscanf(fp_odo, "%s", counts);             // read # of counts per mile
   counts_mile = atof(counts);               // convert from text to float
   }                                         // end of if
fclose(fp_odo);                              // close odometer file
//
// initialize the CPU Board
//
mask = inportb(0xA1) | 0x06;                 // get disable interrupts mask
outportb(0xA1, mask);                        // disable IRQ9 and IRQ10
outportb(0x30, 0x2E);                        // access VMEbus short I/O
outportb(0x34, 0xFF);                        // upper 8 VMEbus address
```

```
outportb(0x36, 0x08);                    // disable byte swapping
//
// initialize the DIO Board
//
gencntrl = MK_FP(0xE000, 0xD001);        // make ptr to general control
portAddr = MK_FP(0xE000, 0xD005);        // make ptr to data direction
portBddr = MK_FP(0xE000, 0xD007);        // make ptr to data direction
portCddr = MK_FP(0xE000, 0xD009);        // make ptr to data direction
portAcntl = MK_FP(0xE000, 0xD00B);       // make ptr to control reg
portBcntl = MK_FP(0xE000, 0xD00F);       // make ptr to control reg
resetint = MK_FP(0xE000, 0xD011);        // make ptr to reset IRQ9 reg
portCdata = MK_FP(0xE000, 0xD019);       // make ptr to port c data reg

*gencntrl = 0x0F;                        // mode 0
*portAcntl = 0x28;                       // submode 00
*portBcntl = 0x68;                       // submode 01
*portAddr = 0xFF;                        // all outputs
*portBddr = 0x00;                        // all inputs
*portCddr = 0x13;                        // initialize data direction
*portCdata = 0xFE;                       // port A output, port B input
*resetint = 0x01;                        // initialize reset IRQ9 bit
//
// initialize the ADC Board
//
control = MK_FP(0xE000, 0xE081);         // ptr to adc control reg
*control = 0x10;                         // reset adc board
*control = 0x00;                         // un-reset adc board

gain = MK_FP(0xE000, 0xE085);            // ptr to adc gain register
for(i = 0; i < 8; i++)                   // initialize gain register
   *gain = i + 0x60;                     // gain of  2
*gain = 0xE0;                            // gain of 10,  left accel
*gain = 0xE1;                            // gain of 10, right accel
*gain = 0x27;                            // gain of  1, 5VDC test
//
// print the header and software version number
//
if(!laptop) {                            // check for laptop not used
   initgraph(&gdriver, &gmode, "");      // initialize graphics mode
   setbkcolor(BLACK);                    // black background
   setcolor(7);                          // light gray foreground
   fillpoly(9, cee);                     // draw a "C"
   fillpoly(17, tee);                    // draw a "T"

   mov_curs(16, 18);                     // move cursor to line 16
   printf("CALTRANS");                   // print message

   mov_curs(20,1);                       // move cursor to line 20
   printf("State of California\n");      // print header
   printf("Department of Transportation\n");
   printf("Division of New Technology, Materials & Research\n");
   printf("Office of Electrical & Electronics Engineering\n\n");
   printf("High Speed Pavement Profilometer Software\n");
   printf("Version 00.01    08/01/94\n\n");
```

```
   printf("Press Return to continue...");
   getkey();                                  // wait for CR to continue
   printf("\x1B[=3h");                        // color text, 80 columns X 25
   printf("\x1B[1;37;40m");                   // bold on, white text on black
   _setcursortype(_NOCURSOR);                 // turn off cursor
   }                                          // end of if
//
// print the main menu
//
while(1) {                                    // loop forever
   printf("\x1B[2J");                         // clear screen
   mov_curs(4,24);                            // move cursor
   printf("HIGH SPEED PAVEMENT PROFILOMETER"); // print out main menu
   mov_curs(8,35);                            // move cursor
   printf("Main Menu");
   mov_curs(DATA_ACQ,8);                      // move cursor
   printf("Data Acquisition");
   mov_curs(DISP_VEL,8);                      // move cursor
   printf("Display Van Speed and GPS Data");
   mov_curs(DATA_PRO,8);                      // move cursor
   printf("Data Reduction and Processing");
   mov_curs(SYS_TEST,8);                      // move cursor
   printf("System Test");
   mov_curs(EXIT_DOS,8);                      // move cursor
   printf("Exit to DOS");
   mov_curs(PROMPT,4);                        // move cursor
   printf("Move the pointer to the desired operation and press Return...");
//
// point to the first menu entry
//
   mov_curs(DATA_ACQ,4);                      // move cursor
   printf("==>");                             // print the pointer
   mov_curs(DATA_ACQ,4);                      // move cursor
//
// point to each menu entry as the up and down cursor arrow keys are
// pressed; when carriage return is entered, perform the selected function
//
   linenum = DATA_ACQ;                        // initialize line number
   while((ch = getkey()) != 0x0D)             // wait for a <CR>
         if(ch == 0) {                        // check for arrow key
            ch = getkey();                    // get second character
            if((ch == 0x48) || (ch == 0x50)) {  // check for up or down arrow
                  printf("  ");               // print over arrow

                  if(ch == 0x48)              // check for up arrow
                     linenum = (linenum == DATA_ACQ) ? EXIT_DOS : linenum - 2;
                  else                        // check for down arrow
                     linenum = (linenum == EXIT_DOS) ? DATA_ACQ : linenum + 2;

                  mov_curs(linenum,4);        // move cursor to new line
                  printf("==>");              // print the pointer
                  mov_curs(linenum,4);        // move cursor to start
                  }                           // end of if
            }                                 // end of if
```

```
        switch(linenum) {                          // check operation selected
            case DATA_ACQ :
                data_acq();                        // data acquisition selected
                break;
            case DISP_VEL :
                disp_vel();                        // display velocity selected
                break;
            case DATA_PRO :
                data_pro();                        // data processing selected
                break;
            case SYS_TEST :
                sys_test();                        // system test selected
                break;
            case EXIT_DOS :
                _setcursortype(_NORMALCURSOR);     // turn cursor on again
                printf("\x1B[2J");                 // clear screen
                dup2(oldstdout, STDOUT);           // put stdout back
                close(oldstdout);                  // close file handle
                exit(0);                           // exit to DOS selected
            }                                      // end of switch
        }                                          // end of while
    }                                              // end of exec.c
```

```
// data_acq.c
// data acquisition source code for High Speed Pavement Profilometer Project
//
// Description          :     This program acquires data from the sensors
//                            and saves it in a file
//                            It is selected from the main menu
//
// Functions Included   :     data_acq(), write_pro_data(), write_nav_
//                            data(), write_loop_data(), write_GPS_msg(),
//                            write_key_data(), display_data(), get_bufcount()
//
// External Functions   :     serial_init(), serial_out(), timeisr(),
//                            distisr()
//
// Programmer           :     Greg Larson
//
// Created              :     11/04/91
//
// Last Revised         :     08/23/94
//
#include <dir.h>
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <io.h>

#define CPATH       "RAW\\"
#define  DPATH      "D:\\"
#define SAMPLES     528
#define DATA        0x3F8

struct probuf {                          // profile data buffer
        unsigned int velcount;           // velocity count data
        unsigned int laccel;             // left accelerometer data
        unsigned int lheight;            // left height data
        unsigned int raccel;             // right accelerometer data
        unsigned int rheight;            // right height data
        };

struct navbuf {                          // navigation data buffer
        unsigned int x_axis;             // x-axis data
        unsigned int y_axis;             // y-axis data
        unsigned int rate;               // angular rate data
        };

extern void serial_init();               // initialize COM2 port
extern void serial_out();                // send command to GPS
extern void GPS_status();                // print GPS receiver status
extern unsigned char check_GPS();        // check the status of the GPS
extern void far interrupt timeisr();     // time-based ISR
extern void far interrupt distisr();     // distance-based ISR
```

```
extern struct probuf lobuf[], hibuf[];          // lo, hi profile data buffers
extern struct navbuf evnbuf[], oddbuf[];        // even, odd nav data buffers
extern unsigned long odmcount, oldodmcount;     // current, old odometer count
extern unsigned long loopodm;                   // odometer count at last loop
extern float counts_mile;                       // # of odometer counts per mile
extern unsigned int looptime, loopind;          // loop time and index
extern unsigned int loopdelay;                  // loop delay
extern unsigned int second;                     // seconds count
extern unsigned int navptr;                     // pointer into nav buffers
extern unsigned int bufptr;                     // profile data buffer pointer
extern unsigned int laccsum, raccsum;           // left, right accel sums
extern unsigned int lhgtsum, rhgtsum;           // left, right height sums
extern unsigned int gpshead, gpstail;           // GPS head, tail pointer
extern unsigned int dispcnt;                    // # of seconds between screen
extern unsigned char intcount;                  // interval counter
extern unsigned char bufflag, fullflag, navflag, loopflag, dispflag, gpsflag;
extern unsigned char countup;                   // PM increasing flag
extern unsigned char laptop;                    // laptop in use flag

char linenum = 4;                               // current cursor position

/*****************************************************************************/
int data_acq(void)                              // acquire data from sensors
{                                               // start of data_acq()
void write_pro_data();                          // save profile data buffer
void write_nav_data();                          // save navigation data buffer
void display_data();                            // display data on second line
void write_GPS_msg();                           // save GPS messages
void write_loop_data();                         // save loop data buffer
void write_key_data();                          // save keyboard data

void interrupt (*oldIRQ9)();                    // old IRQ9  function
void interrupt (*oldIRQ10)();                   // old IRQ10 function

struct ffblk ffblk;                             // find file data structure
struct date date;                               // date data structure
struct time t;                                  // time data structure

FILE *fp_set,                                   // setup file pointer
   *fp_raw,                                     // raw sensor data file ptr
   *fp_nav,                                     // dead reckoning file ptr
   *fp_gps,                                     // gps data file pointer
   *fp_key;                                     // landmark data file ptr

FILE *c_nav,                                    // c:\ drive file pointers
   *c_gps;                                      //

int navbuf[5];                                  // scratch buffer for copying

char string[80],                                // y/n response buffer, 80!
   filename[16],                                // file name buffer
   buffer[30];                                  // scratch buffer

char ch = 0;                                    // keyboard input
```

```
char begPM[10], endPM[10];                          // beginning, ending post mile

char GPS_status_on[] = "$PMGLI,00,H00,2,A,00*48\r\n";
char GPS_status_off[] = "$PMGLI,00,H00,0,A,00*4A\r\n";

char GPS_position_on[]  = "$PMGLI,00,B00,2,A,00*42\r\n";
char GPS_position_off[] = "$PMGLI,00,B00,0,A,00*40\r\n";

int done = 0;                                       // file search done flag

char *funckey[20] = {     "Speed below 20mph ",     // F1, text for function keys
                          "Speed above 20mph ",     // F2
                          "Flex to Rigid    ",      // F3
                          "Rigid to Flex    ",      // F4
                          "Railroad Crossing ",     // F5
                          "Lane Closed      ",      // F6
                          "Back in Lane     ",      // F7
                          "Unusual Occurrence",     // F8
                          "Bridge Approach  ",      // F9
                          "Bridge Departure "};     // F10

char mask;                                          // mask variable for 8259
char escflag = 0x00;                                // exit to main menu flag
char processed = 0x00;                              // file processed flag

char far *resetint;                                 // pointer to reset int bit
char far *status;                                   // pointer to status port
char far *portCdata;                                // pointer to port C data
char far *timer1con;                                // pointer to timer 1 control
char far *load1hi;                                  // pointer to timer 1 load hi
char far *load1mi;                                  // pointer to timer 1 load mi
char far *load1lo;                                  // pointer to timer 1 load lo
char far *timer2con;                                // pointer to timer 2 control
char far *load2hi;                                  // pointer to timer 2 load hi
char far *load2mi;                                  // pointer to timer 2 load mi
char far *load2lo;                                  // pointer to timer 2 load lo

printf("\x1B[2J");                                  // clear screen
printf("Data Acquisition Run Setup\n\n");           // print message
//
// get the run file name and check to see if it already exists
//
do {
   printf("Enter the file name for this run (seven characters or less!) >");
   getmsg(filename);                                // save the file name
   if(strlen(filename) == 0)                        // check for just CR entered
         return(0);                                 // return to main menu if CR
   filename[7] = '\0';                              // truncate file name
   strcpy(buffer, CPATH);                           // copy path into buffer
   strcat(buffer, filename);                        // add file name to buffer
   strcat(buffer, ".set");                          // add extension to buffer
   done = findfirst(buffer, &ffblk, 0);             // see if the file exists
   if(!done) {                                      // file exists
```

```
        printf("   %s.set already exists!  Overwrite it? (y/n) >", filename);
        getmsg(string);                          // get y/n response
        if(string[0] == 'y' || string[0] == 'Y')// check for yes
          done = 0xFFFF;                         // say overwrite
        }                                        // end of if
    } while(!done);                              // loop if not done

if((fp_set = fopen(buffer, "wt")) == NULL) {     // open setup file
    printf("   Error, cannot open setup file!");
    delay(4000);                                 // pause for four seconds
    return(0);                                   // return to main menu
    }
//
// Prompt the operator for the setup information
//
getdate(&date);                                  // get the date
fprintf(fp_set, "\n%2d/%02d/%2d\n", date.da_mon, date.da_day, date.da_year-1900);
fprintf(fp_set, "File Name: %s\n", filename);    // save the file name

printf("Enter the operator's name >");           // prompt for the operator
getmsg(string);                                  // get the operator's name
fprintf(fp_set, " Operator: %s\n", string);      // save operator's name

printf("Enter the number of the district >");    // prompt for the district
getmsg(string);                                  // get the district
fprintf(fp_set, " District: %s\n", string);      // save the district in file

printf("Enter the name of the county >");        // prompt for the county
getmsg(string);                                  // get the county
fprintf(fp_set, "   County: %s\n", string);      // save the county in file

printf("Enter the name of the route >");         // prompt for the route
getmsg(string);                                  // get the route
fprintf(fp_set, "    Route: %s\n", string);      // save the route in file

do {                                             // do...
    printf("Enter the lane number (1 - 9) >");   // prompt for the lane number
    getmsg(string);                              // get the lane number
    } while(string[0] < '1' || string[0] > '9');// ...until limits are met
fprintf(fp_set, "     Lane: %s\n", string);      // save the lane # in file

printf("Enter the direction of travel >");       // prompt for the direction
getmsg(string);                                  // get the direction
fprintf(fp_set, "Direction: %s\n", string);      // save the direction in file

printf("Enter the pavement type >");             // prompt for pavement type
getmsg(string);                                  // get the pavement type
fprintf(fp_set, " Pavement: %s\n", string);      // save pavment type in file

printf("Enter the beginning post mile >");       // prompt for post mile
getmsg(string);                                  // get beginning post mile
fprintf(fp_set, " Begin PM: %s\n", string);      // save beginning PM in file

printf("Enter the beginning odometer >");        // prompt for odometer
```

```
getmsg(begPM);                          // get beginning odometer
fprintf(fp_set, "Begin ODM: %s\n", begPM);   // save beginning ODM in file

printf("Enter the ending post mile >");     // prompt for post mile
getmsg(string);                         // get ending post mile
fprintf(fp_set, "   End PM: %s\n", string);   // save ending PM in file

printf("Enter the ending odometer >");      // prompt for odometer
getmsg(endPM);                          // get ending odometer
fprintf(fp_set, "  End ODM: %s\n", endPM);   // save ending ODM in file

countup = 0;                            // say PM decreasing for now
if(atof(endPM) >= atof(begPM))          // check for PM increasing
  countup = 0xFF;                       // say PM increasing

printf("Enter the comments for this run >");   // prompt for comments
getmsg(string);                         // get the comments
fprintf(fp_set, " Comments: %s\n", string);   // save comments in file

fclose(fp_set);                         // close setup file
//
// Open all of the data files on the C: and D: drives
//
strcpy(buffer, CPATH);                  // copy path into buffer
strcat(buffer, filename);               // add file name to buffer
strcat(buffer, ".raw");                 // add extension to buffer
fp_raw = fopen(buffer, "wb");           // open raw sensor data file
fwrite(&processed, 1, 1, fp_raw);       // write file processed flag
fwrite(&processed, 1, 1, fp_raw);       // dummy value to align data
fwrite(&date, sizeof(date), 1, fp_raw);  // write date into file

strcpy(buffer, CPATH);                  // copy path into buffer
strcat(buffer, filename);               // add file name to buffer
strcat(buffer, ".nav");                 // add extension to buffer
c_nav = fopen(buffer, "wb");            // open c:\ drive nav file
fwrite(&date, sizeof(date), 1, c_nav);  // write date into file
strcpy(buffer, DPATH);                  // copy path into buffer
strcat(buffer, filename);               // add file name to buffer
strcat(buffer, ".nav");                 // add extension to buffer
fp_nav = fopen(buffer, "wb+");          // open d:\ drive nav file

strcpy(buffer, CPATH);                  // copy path into buffer
strcat(buffer, filename);               // add file name to buffer
strcat(buffer, ".gps");                 // add extension to buffer
c_gps = fopen(buffer, "wt");            // open c:\ drive gps file
fprintf(c_gps, "%2d/%02d/%2d  ", date.da_mon, date.da_day, date.da_year-1900);
strcpy(buffer, DPATH);                  // copy path into buffer
strcat(buffer, filename);               // add file name to buffer
strcat(buffer, ".gps");                 // add extension to buffer
fp_gps = fopen(buffer, "wt+");          // open gps data file

strcpy(buffer, CPATH);                  // copy path into buffer
strcat(buffer, filename);               // add file name to buffer
strcat(buffer, ".key");                 // add extension to buffer
```

```
fp_key = fopen(buffer, "wt+");                  // open keyboard data file
// strcpy(buffer, DPATH);                       // copy path into buffer
// strcat(buffer, filename);                    // add file name to buffer
// strcat(buffer, ".key");                      // add extension to buffer
// fp_key = fopen(buffer, "wt+");               // open landmark data file
//
// initialize the serial ports, timers, and interrupts
//
serial_init();                                  // initialize COM2

resetint = MK_FP(0xE000, 0xD011);               // make ptr to reset int bit
status = MK_FP(0xE000, 0xD013);                 // make ptr to status port
portCdata = MK_FP(0xE000, 0xD019);              // make ptr to port C data
timer1con = MK_FP(0xE000, 0xD021);              // make ptr to timer 1 control
load1hi = MK_FP(0xE000, 0xD027);                // make ptr to timer 1 load hi
load1mi = MK_FP(0xE000, 0xD029);                // make ptr to timer 1 load mi
load1lo = MK_FP(0xE000, 0xD02B);                // make ptr to timer 1 load lo
timer2con = MK_FP(0xE000, 0xD061);              // make ptr to timer 2 control
load2hi = MK_FP(0xE000, 0xD067);                // make ptr to timer 2 load hi
load2mi = MK_FP(0xE000, 0xD069);                // make ptr to timer 2 load mi
load2lo = MK_FP(0xE000, 0xD06B);                // make ptr to timer 2 load lo

*timer1con = 0xA0;                              // periodic interrupt mode
*load1hi = 0x00;                                // load high count value
*load1mi = 0x00;                                // load middle count value
*load1lo = 0xFA;                                // load low count value, 250

*timer2con = 0x10;                              // elapsed time mode
*load2hi = 0xFF;                                // load high count value
*load2mi = 0xFF;                                // load middle count value
*load2lo = 0xFF;                                // load low count value

oldIRQ9 = getvect(0x0A);                        // save old IRQ9 function
setvect(0x0A, distisr);                         // initialize IRQ9 vector

oldIRQ10 = getvect(0x72);                       // save old IRQ10 function
setvect(0x72, timeisr);                         // initialize IRQ10 vector
mask = inportb(0xA1);                           // get interrupt mask
//
// Check the GPS Receiver
//
*timer1con = 0xA1;                              // start one millisecond timer
*portCdata = 0xEE;                              // enable timer interrupt
outportb(0x30, 0xAE);                           // enable VMEbus interrupts
outportb(0xA1, mask&0xFB);                      // enable IRQ10 on CPU

if(check_GPS()) {                               // check the GPS receiver
   outportb(0xA1, mask);                        // disable IRQ10 on CPU
   outportb(0x30, 0x2E);                        // disable VMEbus interrupts
   *portCdata = 0xFE;                           // disable timer interrupt
   *timer1con = 0xA0;                           // stop one millisecond timer
   setvect(0x72, oldIRQ10);                     // restore old IRQ10 vector
   return(0);                                   // return to the main menu
   }                                            // end of if
```

```
//
// Initialize all global variables
//
done = 0x0000;                          // initialize done flag

odmcount = 0;                           // reset odometer count
oldodmcount = 0;                        // reset old odometer count
loopodm = 0;                            // reset count at last loop
bufptr = 0;                             // reset pointer into buffer
intcount = 1;                           // reset interval counter
laccsum = 0;                            // left accel partial sum
raccsum = 0;                            // right accel partial sum
lhgtsum = 0;                            // left height partial sum
rhgtsum = 0;                            // right height partial sum
bufflag = 0;                            // reset buffer flag
fullflag = 0;                           // reset buffer full flag
dispflag = 0xFF;                        // set display data flag
navflag = 0;                            // reset process nav data flag
gpsflag = 0;                            // reset process gps data flag
dispcnt = 1;                            // initialize # of seconds
loopdelay = 1000;                       // initialize loop delay

printf("\x1B[2J");                      // clear screen
serial_out(GPS_status_on);              // turn on GPS status message
//
// Start the data acquisition
//
printf("Press \'B\' to begin the data acquisition run...");
while(tolower(ch) != 'b') {             // wait for a "b" from keyboard
   GPS_status();                        // print GPS status
   if(chkkey())                         // check for input from keybrd
         ch = getkey();                 // get the key
   }                                    // end of while

/*****************************************************************************/
second = 0;                             // reset second count
navptr = 0;                             // reset pointer into nav buffer
gpshead = 0;                            // reset head pointer
gpstail = 0;                            // reset tail pointer
gpsflag = 0;                            // reset gps flag

gettime(&t);                            // get the current time
serial_out(GPS_status_off);             // turn off GPS status message
printf("\x1B[2J");                      // clear screen
//
// watch the raw wheel pulse signal and wait for a rising edge before
// enabling the distance-based interrupt
//
while(*status < 0);                     // wait for low raw signal
while(*status >= 0);                    // detect rising edge
*resetint = 0x00;                       // reset distance-based int
*resetint = 0x01;                       // un-reset interrupt
*timer2con = 0x11;                      // start velocity timer
mask = inportb(0xA1);                   // get interrupt mask
```

```
outportb(0xA1, mask&0xFD);                          // enable IRQ9

serial_out(GPS_position_on);                        // turn on GPS message

fwrite(&t, sizeof(t), 1, fp_raw);                   // write time into files
fwrite(&t, sizeof(t), 1, c_nav);                    //
fprintf(c_gps, "%2d:%02d:%02d.%02d\n", t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
fprintf(fp_key, "%2d:%02d:%02d.%02d\n", t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);

printf(" Ending PM: %7.3f  Beginning PM: %7.3f      \nCurrent PM: ",
        atof(endPM), atof(begPM));                  // print header
mov_curs(4,1);                                      // move cursor to line 4

do {                                                // time critical loop
  if(fullflag)                                      // check for full profile buffer
        write_pro_data(fp_raw, &escflag, &done);    // save full buffer
  if(navflag)                                       // check for nav buffer ready
        write_nav_data(fp_nav);                     // save nav buffer
  if(dispflag)                                      // check for time to display
        display_data(begPM);                        // display data on second line
  if(gpsflag)                                       // check for GPS message ready
        write_GPS_msg(fp_gps);                      // save GPS message
  if(loopflag)                                      // check for loop detected
        write_loop_data(fp_key);                    // save loop data
  if(chkkey())                                      // check for input from keybrd
        write_key_data(fp_key, &funckey, &escflag, &done);        // save key data
  } while(!done);                                   // loop until done

gettime(&t);                                        // get the current time

mask = inportb(0xA1) | 0x06;                        // get disable interrupt mask
outportb(0xA1, mask);                               // disable IRQ9 and IRQ10
outportb(0x30, 0x2E);                               // disable VMEbus interrupts
*portCdata = 0xFE;                                  // disable timer interrupt
*timer1con = 0xA0;                                  // stop one millisecond timer
*timer2con = 0x10;                                  // stop velocity timer

setvect(0x0A, oldIRQ9);                             // restore old IRQ9 vector
setvect(0x72, oldIRQ10);                            // restore old IRQ10 vector

serial_out(GPS_position_off);                       // turn off GPS messages

fclose(fp_raw);                                     // close raw sensor data file
printf("\x1B[2J");                                  // clear screen
printf("End of data acquisition!\n\n");             // print message
//
// copy the files on the D:\ drive (RAMdrive) to the C:\ drive
//
printf("Saving RAMdrive data files...\n");          // print message

rewind(fp_nav);                                     // rewind nav file pointer
while(fread(navbuf, 10, 1, fp_nav))                 // read d:\ drive data until EOF
  fwrite(navbuf, 10, 1, c_ nav);                    // write data onto c:\ drive
fclose(c_nav);                                      // close c:\ drive file
```

```
fclose(fp_nav);                                  // close navigation data file

rewind(fp_gps);                                  // rewind GPS file pointer
while(fgets(string, 80, fp_gps))                 // read d:\ drive data until EOF
  fputs(string, c_gps);                          // write data onto c:\ drive
fclose(c_gps);                                   // close c:\ drive file
fclose(fp_gps);                                  // close GPS data file

// rewind(fp_key);                               // rewind keyboard file pointer
// while(fgets(string, 80, fp_key))              // read d:\ drive data until EOF
//   fputs(string, c_key);                       / / write data onto c:\ drive
// fprintf(c_key, "%2d:%02d:%02d.%02d", t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
// fclose(c_key);                                // close c:\ drive file
fclose(fp_key);                                  // close keyboard data file

delay(3000);                                     // pause for three seconds
return(0);                                       // normal return to main menu
}                                                // end of data_acq()
/*****************************************************************************/
void write_pro_data(FILE *fp_raw, char *escflag, int *done)
{                                                // save full profile data buffer
fullflag = 0;                                    // reset buffer full flag

if(*escflag)                                     // check for escape key pressed
  *done = 0xFFFF;                                // say done

if(bufflag)                                      // check for low buffer
  fwrite(&lobuf, SAMPLES*5*2, 1, fp_raw);        // save low buffer in file
else                                             // use high buffer
  fwrite(&hibuf, SAMPLES*5*2, 1, fp_raw);        // save high buffer in file
}                                                // end of write_pro_data
/*****************************************************************************/
void write_nav_data(FILE *fp_nav)                // save navigation data buffer
{
struct {
    unsigned long odmcount;                      // odometer count
    unsigned int x_axis;                         // x axis data
    unsigned int y_axis;                         // y axis data
    unsigned int rate;                           // angular rate
    } navdata;                                   // navigation data structure

unsigned long xtotal = 0,                        // sum of x axis data
              ytotal = 0,                        // sum of y axis data
              rtotal = 0;                        // sum of angular rate data

int i;                                           // index

navflag = 0;                                     // reset nav buffer ready flag

if(second & 0x0001)                              // check for odd seconds
  for(i = 0; i < 1000; i++) {                    // one second of raw data
        xtotal = xtotal + evnbuf[i].x_axis;      // sum x-axis data
        ytotal = ytotal + evnbuf[i].y_axis;      // sum y-axis data
        rtotal = rtotal + evnbuf[i].rate;        // sum angular rate data
```

```
        }
else                                        // even seconds
   for(i = 0; i < 1000; i++) {              // one second of raw data
         xtotal = xtotal + oddbuf[i].x_axis;  // sum x-axis data
         ytotal = ytotal + oddbuf[i].y_axis;  // sum y-axis data
         rtotal = rtotal + oddbuf[i].rate;    // sum angular rate data
         }

navdata.odmcount = odmcount;                // get odometer count
navdata.x_axis = xtotal/1000;               // calculate x axis average
navdata.y_axis = ytotal/1000;               // calculate y axis average
navdata.rate = rtotal/1000;                 // calculate angular rate ave.

fwrite(&navdata, 10, 1, fp_nav);            // save buffer in file

}                                           // end of write_nav_data
/*****************************************************************************/
void display_data(char begPM[])             // display data on second line
{
float curPM, elapsedM;                      // current & elapsed miles
float speed;                                // vehicle speed
int hour, min, sec, left;                   // elapsed time

dispflag = 0;                               // reset display data flag
printf("\x1B[s");                           // save cursor position
mov_curs(2, 13);                            // move cursor to line 2

elapsedM = odmcount/counts_mile;            // calculate elapsed miles
if(countup)                                 // check for PM increasing
   curPM = atof(begPM) + elapsedM;          // calculate current post mile
else{                                       // PM is decreasing
   curPM = atof(begPM) - elapsedM;          // calculate current post mile
   if(curPM < 0)                            // check for negative PM
         curPM = 0.0;                       // force current PM to 0
   }                                        // end of else

hour = second/3600;                         // calculate hours
left = second%3600;                         // calculate remainder
min = left/60;                              // calculate minutes
sec = left%60;                              // calculate seconds

speed = (odmcount - oldodmcount)*0.05682;   // convert pulses to mph
oldodmcount = odmcount;                     // update old odometer count

printf("%7.3f    Elapsed M: %7.3f  Elapsed T: %2d:%02d:%02d  Speed: %4.2f",
         curPM, elapsedM, hour, min, sec, speed);

printf("\x1B[u");                           // restore cursor position
}                                           // end of display_data
/*****************************************************************************/
void write_GPS_msg(FILE *fp_gps)            // save gps message
{
read_message(fp_gps);                       // just do it
}                                           // end of write_GPS_msg
```

```
/*****************************************************************************/
void write_loop_data(FILE *fp_key)                  // save loop data buffer
{
int hour, min, sec, msec, left;                     // elapsed time
float elapsedM;                                     // elapsed miles

loopflag = 0;                                       // reset loop detected flag

if(odmcount > loopodm + 120) {                      // do not report the same loop
   hour = looptime/3600;                            // calculate hours
   left = looptime%3600;                            // calculate remainder
   min = left/60;                                   // calculate minutes
   sec = left%60;                                   // calculate seconds
   msec = loopind/6;                                // calculate milliseconds

   elapsedM = odmcount/counts_mile;                 // calculate elapsed miles

   fprintf(fp_key, "%2d:%02d:%02d.%03d  %7.3f  Loop\n",
                    hour, min, sec, msec, elapsedM);
   printf("Loop");                                  // print message
   if(linenum == 25) {                              // check for last line
         linenum = 4;                               // re-initialize line number
         mov_curs(4, 1);                            // move cursor to line 4
         }                                          // end of if
   else{                                            // not last line
         linenum++;                                 // increment line number
         printf("\n\x1B[K");                        // erase new line, cursor to end
         }                                          // end of else
   }                                                // end of if
}                                                   // end of write_loop_data
/*****************************************************************************/
void write_key_data(FILE *fp_key, char *funckey[], char *escflag, int *done)
{
char ch;                                            // keyboard character
int fkey;                                           // function key
int hour, min, sec, msec, left;                     // elapsed time
float elapsedM;                                     // elapsed miles

hour = second/3600;                                 // calculate hours
left = second%3600;                                 // calculate remainder
min = left/60;                                      // calculate minutes
sec = left%60;                                      // calculate seconds
msec = navptr/6;                                    // calculate milliseconds
elapsedM = odmcount/counts_mile;                    // calculate elapsed miles

// ch = (char)(getkey()&0x007F);                    // get the key

if(!laptop)
   ch = (char)getch();
else
   ch = inportb(DATA);

if(ch >= '0' && ch <= '9') {                        // check for number key
   if(ch == '0')                                    // check for zero key
```

```
        ch = 9;                                     // force key to nine
    else                                            // not zero key
        ch = ch - '1';                              // subtract ASCII bias
    if(fprintf(fp_key, "%2d:%02d:%02d.%03d  %7.3f  %s\n",
                hour, min, sec, msec, elapsedM, *(funckey+ch)) == EOF) {
        printf("Error writing to keyboard file!");
        exit(1);                                    // exit to DOS
        }                                           // end of if
    printf("%7.3f  %s", elapsedM, *(funckey+ch));
    if(linenum == 25) {                             // check for last line
        linenum = 4;                                // re-initialize line number
        mov_curs(4, 1);                             // move cursor to line 4
        }                                           // end of if
    else{                                           // not last line
        linenum++;                                  // increment line number
        printf("\n\x1B[K");                         // erase new line, cursor to end
        }                                           // end of else
    }                                               // end of if
//  else if(fkey == 12)                             // stop data acquisition NOW!
//      *done = 0xFFFF;                             // say done

else if(ch == 0x1B) {                               // check for escape pressed
    *escflag = 0xFF;                                // say escape key pressed
    fprintf(fp_key, "%2d:%02d:%02d.%03d  %7.3f  Escape\n",
                hour, min, sec, msec, elapsedM);
    mov_curs(4, 1);                                 // move cursor to line 4
    for(linenum = 4; linenum < 25; linenum++)       // clear out
        printf("\x1B[K\n");                         // clear until end of line
    printf("\x1B[K");                               // clear line 25
    mov_curs(4, 1);                                 // move cursor to line 4
    printf("Waiting for the last buffer to fill...");
    }                                               // end of if
}                                                   // end of write_key_data
```

```
// disp_vel.c
// display van speed and GPS position data source code for High Speed Pavement
// Profilometer Project
//
// Description          :     This program displays van speed data and GPS
//                            position data on the screen
//                            It is selected from the main menu
//
// Functions Included   :     disp_vel()
//
// External Functions   :     None
//
// Programmer           :     Greg Larson
//
// Created              :     08/12/93
//
// Last Revised         :     02/01/94
//
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <io.h>

extern void serial_init();                          // initialize COM2 port
extern void serial_out();                           // send command to GPS
extern void GPS_messages_off();                     // turn off GPS messages
extern void far interrupt timeisr();                // time-based ISR

extern unsigned long odmcount, oldodmcount;         // current, old odometer count
extern unsigned long loopodm;                       //
extern unsigned int loopdelay;                      //
extern float counts_mile;                           // # of odometer counts per mile
extern unsigned int second;                         // seconds count
extern char gpsque[];                               // GPS data queue
extern unsigned int gpshead, gpstail;               // GPS head, tail pointer
extern unsigned char dispflag, gpsflag;             // display, GPS data flags
extern unsigned char navflag, bufflag, fullflag;
extern unsigned int bufptr, navptr;
extern unsigned int dispcnt;

/*****************************************************************************/
int disp_vel(void)                                  // display van speed
{                                                   // start of disp_vel()
void far interrupt odometer_isr();                  // odometer isr
void interrupt (*oldIRQ9)();                        // old IRQ9 vector
void interrupt (*oldIRQ10)();                       // old IRQ10 vector

char GPS_position_on[]  = "$PMGLI,00,B00,2,A,00*42\r\n";
char GPS_position_off[] = "$PMGLI,00,B00,0,A,00*40\r\n";

char mask;                                          // mask variable for 8259
```

```
char far *resetint;                    // ptr to reset interrupt bit
char far *status;                      // pointer to status port
char far *portCdata;                   // pointer to port C data
char far *timer1con;                   // pointer to timer 1 control
char far *load1hi;                     // pointer to timer 1 load hi
char far *load1mi;                     // pointer to timer 1 load mi
char far *load1lo;                     // pointer to timer 1 load lo

unsigned char msg_ptr;                 // message pointer
char message[128];                     // message buffer

int linenum = 4;                       // present cursor position
float elapsedM;                        // current miles
char begPM[] = "0.000";                // beginning post mile
float speed;                           // vehicle speed
int hour, min, sec, left;              // elapsed time

printf("\x1B[2J");                     // clear screen
//
// initialize the serial ports, timers, and interrupts
//
serial_init();                         // initialize COM2
GPS_messages_off();                    // turn off all GPS messages
delay(200);                            // wait for 0.2 seconds

resetint = MK_FP(0xE000, 0xD011);      // make ptr to reset int bit
status = MK_FP(0xE000, 0xD013);        // make ptr to status port
portCdata = MK_FP(0xE000, 0xD019);     // make ptr to port C data
timer1con = MK_FP(0xE000, 0xD021);     // make ptr to timer 1 control
load1hi = MK_FP(0xE000, 0xD027);       // make ptr to timer 1 load hi
load1mi = MK_FP(0xE000, 0xD029);       // make ptr to timer 1 load mi
load1lo = MK_FP(0xE000, 0xD02B);       // make ptr to timer 1 load lo

*timer1con = 0xA0;                     // periodic interrupt mode
*load1hi = 0x00;                       // load high count value
*load1mi = 0x00;                       // load middle count value
*load1lo = 0xFA;                       // load low count value, 250

oldIRQ9  = getvect(0x0A);              // save old IRQ9 vector
setvect(0x0A, odometer_isr);           // initialize IRQ9 vector
oldIRQ10 = getvect(0x72);              // save old IRQ10 vector
setvect(0x72, timeisr);                // initialize IRQ10 vector
mask = inportb(0xA1);                  // get interrupt mask

second = 0;                            // reset second count
gpshead = 0;                           // reset head pointer
gpstail = 0;                           // reset tail pointer
gpsflag = 0;                           // reset flag
odmcount = 0;                          // reset odometer count
oldodmcount = 0;                       // reset old odometer count
loopodm = 0;                           //
loopdelay = 1000;                      //
bufptr = 0;                            //
bufflag = 0;                           //
```

```
fullflag = 0;                                   //
navptr = 0;                                     //
navflag = 0;                                    //
dispcnt = 1;                                    //

*timer1con = 0xA1;                              // start one millisecond timer
*portCdata = 0xEE;                              // enable timer interrupt
outportb(0x30, 0xAE);                           // enable VMEbus interrupts
outportb(0xA1, mask&0xFB);                      // enable IRQ10 on CPU

serial_out(GPS_position_on);                    // turn on GPS message

printf(" Ending PM:  XX.XXX  Beginning PM: %7.3f\nCurrent PM: ", atof(begPM));
mov_curs(4, 1);                                 // move cursor to line 4

// watch the raw wheel pulse signal and wait for a rising edge before
// enabling the distance-based interrupt

while(*status < 0);                             // wait for low raw signal
while(*status >= 0);                            // detect rising edge
*resetint = 0x00;                               // reset distance-based int
*resetint = 0x01;                               // un-reset interrupt
mask = inportb(0xA1);                           // get interrupt mask
outportb(0xA1, mask&0xFD);                      // enable IRQ9

dispflag = 0xFF;                                // say time to display data
do {                                            // time critical loop
  if(dispflag) {                                // check for time to display
        dispflag = 0;                           // reset display data flag

        printf("\x1B[s");                       // save cursor position
        mov_curs(2, 13);                        // move cursor to line 2

        elapsedM = odmcount/counts_mile;        // calculate elapsed miles

        hour = second/3600;                     // calculate hours
        left = second%3600;                     // calculate remainder
        min = left/60;                          // calculate minutes
        sec = left%60;                          // calculate seconds

        speed = (odmcount - oldodmcount)*0.05682;        // convert pulses to mph
        oldodmcount = odmcount;                 // update old odometer count

        printf("%7.3f    Elapsed M: %7.3f  Elapsed T: %2d:%02d:%02d  Speed: %4.2f",
                elapsedM, elapsedM, hour, min, sec, speed);

        printf("\x1B[u");                       // restore cursor position
        }                                       // end of if

  if(gpsflag)     {                             // check for GPS message ready
        gpsflag = 0;                            // reset GPS flag
        msg_ptr = 0;                            // initialize pointer into msg

        do {                                    // do...
```

```
            gpstail = gpstail & 0xFF;                   // reset gpstail at 0x100 (256)
            message[msg_ptr++] = gpsque[gpstail];
            } while(gpsque[gpstail++] != '\n');         // ...until end of message

         message[--msg_ptr] = 0;                        // install null terminator
         printf("%s", message);                         // print GPS position data

         if(linenum == 25) {                            // check for last line
            linenum = 4;                                // re-initialize cursor position
            mov_curs(4, 1);                             // move cursor to line 4
            }                                           // end of if
         else{                                          // not last line
            linenum++;                                  // increment line number
            printf("\n\x1B[K");                         // erase new line, cursor to end
            }                                           // end of else
         }                                              // end of if

   } while(!chkkey());                                  // wait for escape key

mask = inportb(0xA1) | 0x06;                            // get disable interrupt mask
outportb(0xA1, mask);                                   // disable IRQ9 and IRQ10
outportb(0x30, 0x2E);                                   // disable VMEbus interrupts
*portCdata = 0xFE;                                      // disable timer interrupt
*timer1con = 0xA0;                                      // stop one millisecond timer

setvect(0x0A, oldIRQ9);                                 // restore old IRQ9 vector
setvect(0x72, oldIRQ10);                                // restore old IRQ10 vector

serial_out(GPS_position_off);                           // turn off GPS messages

printf("\x1B[2J");                                      // clear screen
printf("End of data display!");                         // print message
delay(3000);                                            // wait three seconds

return(0);                                              // normal return to main menu
}                                                       // end of disp_vel()
/*****************************************************************************/
void far interrupt odometer_isr(void)
{                                                       // start of odometer_isr()
char far *resetint;                                     // pointer to reset int bit

resetint = MK_FP(0xE000, 0xD011);                       // make ptr to reset int bit
*resetint = 0x00;                                       // reset distance-based int
*resetint = 0x01;                                       // un-reset interrupt

odmcount++;                                             // increment odometer count

outportb(0xA0, 0x20);                                   // send EOI to  slave 8259
outportb(0x20, 0x20);                                   // send EOI to master 8259
}                                                       // end of odometer_isr()
```

```
// data_pro.c
// data reduction and processing source code
//
// Description          :   This program reduces and processes raw sensor
//                              data into slope profile files, and processes
//                              slope profile data into elevation profile and
//                              IRI files.  It is selected from the main menu
//
// Functions Included   :       data_pro(), getraw(), getslp(), getnav()
//
// External Functions   :       raw(), slope(), cnvnav()
//
// Programmer           :       Greg Larson
//
// Created              :       11/04/91
//
// Last Revised         :       06/13/94
//
#include <dir.h>
#include <stdio.h>

#define RAW_DATA     10
#define SLP_DATA     12
#define BAT_DATA     14
#define NAV_DATA     16
#define RETURN       18
#define PROMPT       20

#define  RAWPATH     "RAW\\"
#define  SLOPEPATH   "SLOPE\\"
//
int raw();                                  // process raw sensor data
int slope();                                // process slope sensor data
int cnvnav();                               // convert navigation data
/*****************************************************************************/
int data_pro(void)
{                                           // start of data_pro()
int getraw();                               // get the raw data file name
int getslp();                               // get the slope file name
int getbat();                               // get the batch file names
int getnav();                               // get the navigation  file name

unsigned char ch;                           // input char
unsigned char linenum;                      // menu line number

// print the data reduction and processing menu

while(1) {                                  // loop forever
   printf("\x1B[2J");                       // clear screen
   mov_curs(4,24);                          // move cursor to line 4
   printf("HIGH SPEED PAVEMENT PROFILOMETER"); // print menu
   mov_curs(8,23);
   printf("Data Reduction and Processing Menu");
   mov_curs(RAW_DATA,8);
```

```
    printf("Process Raw Sensor Data");
    mov_curs(SLP_DATA,8);
    printf("Process Slope Profile Data");
    mov_curs(BAT_DATA,8);
    printf("Batch Process Raw Sensor Data");
    mov_curs(NAV_DATA,8);
    printf("Convert Navigation Binary Data File to Text");
    mov_curs(RETURN,8);
    printf("Return to the Main Menu");
    mov_curs(PROMPT,4);
    printf("Move the pointer to the desired operation and press Return...");

    mov_curs(RAW_DATA,4);                    // move the cursor to line 4
    printf("==>");                           // print the pointer
    mov_curs(RAW_DATA,4);                    // move cursor to start

// highlight the menu entries as the up and down cursor arrow keys are
// pressed; when carriage return is entered, perform the highlighted function

    linenum = RAW_DATA;                      // initialize line number
    while((ch = getkey()) != 0x0D)           // wait for a <CR>
        if(ch == 0) {                        // check for arrow key
          ch = getkey();                     // get second character
          if((ch == 0x48) || (ch == 0x50)) { // check for up or down arrow
                printf("  ");                // print spaces

                if(ch == 0x48)               // check for up arrow
                  linenum = (linenum == RAW_DATA) ? RETURN : linenum - 2;
                else                         // check for down arrow
                  linenum = (linenum == RETURN) ? RAW_DATA : linenum + 2;

                mov_curs(linenum,4);         // move cursor to new line
                printf("==>");               // print the pointer
                mov_curs(linenum,4);         // move cursor to start
                }                            // end of if
          }                                  // end of if(ch == 0)

    printf("\x1B[2J");                       // clear screen
    switch(linenum) {                        // check operation selected
        case RAW_DATA :
          getraw();                          // process raw data selected
          break;
        case SLP_DATA :
          getslp();                          // process slope data selected
          break;
        case BAT_DATA :
          getbat();                          // process batch data selected
          break;
        case NAV_DATA :
          getnav();                          // convert nav data selected
          break;
        default :
          return(0);                         // return to main menu
        }                                    // end of switch
```

```
   }                                          // end of while(1)
}                                             // end of data_pro()
/*****************************************************************************/
int getraw(void)
{                                             // start of getraw()
int done;                                     // directory search done flag
struct ffblk ffblk;                           // file finder data structure
char path[] = RAWPATH;                        // raw data directory

printf("Available raw sensor data files:\n\n");   // print header
done = findfirst("RAW\\*.RAW", &ffblk, 0);    // search directory
if(done) {
   printf("   No raw sensor data files are available!");
   delay(3000);                               // pause
   return(0);                                 // return to process menu
   }
if(!getfile(&done, &ffblk, &path))            // get file name
   raw(ffblk.ff_name);                        // process raw data file
return(0);                                    // exit
}                                             // end of getraw()
/*****************************************************************************/
int getslp(void)
{                                             // start of getslp()
int done;                                     // directory search done flag
struct ffblk ffblk;                           // file finder data structure
char path[] = SLOPEPATH;                      // slope data directory

printf("Available slope profile data files:\n\n");
done = findfirst("SLOPE\\*.SLP", &ffblk, 0);  // search directory
if(done) {
   printf("   No slope profile data files are available!");
   delay(3000);                               // pause
   return(0);                                 // return to process menu
   }                                          // end of if
if(!getfile(&done, &ffblk, &path))            // get file name
   slope(ffblk.ff_name);                      // process slope data file
return(0);                                    // exit
}                                             // end of getslp()
/*****************************************************************************/
int getnav(void)
{                                             // start of getnav()
int done;                                     // directory search done flag
struct ffblk ffblk;                           // file finder data structure
char path[] = RAWPATH;                        // raw data directory

printf("Available navigation data files:\n\n");
done = findfirst("RAW\\*.NAV", &ffblk, 0);    // search directory
if(done) {
   printf("   No navigation data files are available!");
   delay(3000);                               // pause
   return(0);                                 // return to process menu
   }                                          // end of if
if(!getfile(&done, &ffblk, &path))            // get file name
   cnvnav(ffblk.ff_name);                     // process navigation data file
```

```
return(0);                                          // exit
}                                                   // end of getnav()
/*****************************************************************************/
int getfile(int *done, struct ffblk *ffblk, char *path)
{                                                   // start of getfile()
unsigned char filecnt;                              // number of files found
unsigned char i, ch;                                // loop index, input char
unsigned char linenum, column;                      // line, column number
unsigned char processed;                            // file already processed flag
char file_name[84][13];                             // array of file names
char buffer[25];                                    // scratch buffer
FILE *fileptr;                                      // pointer to data file
//
// find each file name
//
for(filecnt = 0; !(*done) && filecnt < 84; filecnt++) {          // count .XXX files
   strcpy(file_name[filecnt], (*ffblk).ff_name);    // copy file name
   *done = findnext(ffblk);                         // search for files
   }
//
// print out all the file names
//
linenum = 3;                                        // initialize line number
for(i = 0; i < filecnt; i++, linenum++) {           // print out file names
   if(linenum == 24) linenum = 3;                   // reset line number
   switch(i/21) {                                   // 21 file names per column
         case 1 :                                   // check for second column
            mov_curs(linenum,21);                   // move cursor to second column
            break;                                  // break
         case 2 :                                   // check for third column
            mov_curs(linenum,41);                   // move cursor to third column
            break;                                  // break
         case 3 :                                   // check for fourth column
            mov_curs(linenum,61);                   // move cursor to fourth column
         }                                          // end of switch
   strcpy(buffer, path);                            // install path in buffer
   strcat(buffer, file_name[i]);                    // add file name to path
   fileptr = fopen(buffer, "rb");                   // open file for reading
   fread(&processed, 1, 1, fileptr);                // get file processed flag
   fclose(fileptr);                                 // close file
   if(processed)                                    // check for file processed
         printf("   * %s\n", file_name[i]);         // print processed file name
   else                                             // un-processed file
         printf("     %s\n", file_name[i]);         // print un-processed file name
   }                                                // end of for
mov_curs(25,1);                                     // move cursor to last line
printf("Move the pointer to the desired file name and press return...");

mov_curs(3,1);                                      // move cursor to line 3
printf("==>");                                      // print the pointer
mov_curs(3,1);                                      // move cursor to line 3
linenum = 3;                                        // initialize line number

i = 0;                                              // initialize file name index
```

```
while((ch = getkey()) != 0x0D) {                    // wait for a <CR>
  if(ch == 0x1B)                                    // check for escape
        return(1);                                  // say escape pressed
  if(ch == 0) {                                     // check for arrow key
        ch = getkey();                              // get second character
        if((ch == 0x48) || (ch == 0x50)) {          // check for up or down arrow
          printf("   ");                            // print spaces

          if(ch == 0x48) {                          // check for up arrow
                if(linenum == 3) linenum = 23;      // top line, wrap around
                else linenum--;                     // otherwise, move up one line

                if(i == 0) {                        // check for first file
                  i = filecnt - 1;                  // point to last file
                  linenum = filecnt%21 + 2;         // calculate line number
                  if(filecnt%21 == 0)               // check for multiple of 21
                        linenum = 23;               // move cursor to last line
                  }                                 // end of if
                else i--;                           // not first file, decrement
                }                                   // end of if

          else {                                    // process down arrow
                if(linenum == 23) linenum = 3;      // bottom line, wrap around
                else linenum++;                     // otherwise, move down one line

                if(i == filecnt - 1) {              // check for last file
                  i = 0;                            // point to first file
                  linenum = 3;                      // move cursor to top line
                  }                                 // end of if
                else i++;                           // not last file, increment
                }                                   // end of else

          switch(i/21) {                            // 21 file names per column
                case 0 :                            // check for first column
                  column = 1;                       // cursor position
                  break;                            // break
                case 1 :                            // check for second column
                  column = 21;                      // cursor position
                  break;                            // break
                case 2 :                            // check for third column
                  column = 41;                      // cursor position
                  break;                            // break
                default:                            // fourth column
                  column = 61;                      // cursor position
                }                                   // end of switch

          mov_curs(linenum,column);                 // move cursor to new line
          printf("==>");                            // print the pointer
          mov_curs(linenum,column);                 // move cursor to start
          }                                         // end of if
        }                                           // end of if(ch == 0)
  }                                                 // end of while

printf("\x1B[2J");                                  // clear screen
```

```
strcpy((*ffblk).ff_name, file_name[i]);          // copy the selected file name
return(0);                                       // exit
}                                                // end of getfile()
/*****************************************************************************/
int getbat(void)
{                                                // start of getbat()
int done;                                        // directory search done flag
struct ffblk ffblk;                              // file finder data structure
unsigned char i, j, ch;                          // loop indices, input char
unsigned char column, linenum;                   // cursor column, line number
unsigned char processed;                         // file already processed flag
char file_name[84][13];                          // array of file names
char tag_name[84][13];                           // array of tagged file names
unsigned char filecnt;                           // number of files found
unsigned char tagcnt = 0;                        // number of tagged files
char name[10], buffer[20];                       // file name, scratch buffers
FILE *fp_report;                                 // pointer to report file
FILE *fp_raw;                                    // pointer to raw data file
//
// check for any files available
//
printf("Available raw sensor data files:\n\n");  // print header
done = findfirst("RAW\\*.RAW", &ffblk, 0);       // search directory
if(done) {
   printf("   No raw sensor data files are available!");
   delay(3000);                                  // pause
   return(0);                                    // return to process menu
   }                                             // end of if
//
// find each file name and pad it with spaces
//
for(filecnt = 0; !done && filecnt < 84; filecnt++) {   // count .RAW files
   strcpy(file_name[filecnt], ffblk.ff_name);    // copy file name
   for(i = 0; file_name[filecnt][i] != '\0'; i++);   // find null terminator
   i--;                                          // compensate for loop count
   while(i++ < 11)                               // pad with spaces
         file_name[filecnt][i] = ' ';            // install a space
   file_name[filecnt][12] = '\0';                // install null terminator
   done = findnext(&ffblk);                      // search for files
   }                                             // end of for
//
// print out all the file names
//
for(i = 0; i < filecnt; i++) {                   // print out file names
   linenum = wherey();                           // get current line number
   if(linenum == 24) linenum = 3;                // reset to top of column
   switch(i/21) {                                // 21 file names per column
         case 1 :                                // check for second column
            mov_curs(linenum,21);                // move cursor to second column
            break;                               // break
         case 2 :                                // check for third column
            mov_curs(linenum,41);                // move cursor to third column
            break;                               // break
         case 3 :                                // check for fourth column
```

```
            mov_curs(linenum,61);                   // move cursor to fourth column
          }                                         // end of switch
    strcpy(buffer, "RAW\\");                        // install path in buffer
    strcat(buffer, file_name[i]);                   // add file name to path
    fp_raw = fopen(buffer, "rb");                   // open file for reading
    fread(&processed, 1, 1, fp_raw);                // get file processed flag
    fclose(fp_raw);                                 // close file
    if(processed)                                   // check for file processed
          printf("* %s\n", file_name[i]);           // print processed file name
    else                                            // un-processed file
          printf("  %s\n", file_name[i]);           // print un-processed file name
    }                                               // end of for
mov_curs(25,1);                                     // move cursor to last line
printf("Press return to tag a file; Press escape to start processing...");
//
// highlight the first file name
//
mov_curs(3,3);                                      // move cursor to line 3
printf("%s<===", file_name[0]);                     // re-print the first file name
mov_curs(3,3);                                      // move cursor to line 3
linenum = 3;                                        // initialize line number
//
// highlight the tagged files and wait for an Esc to begin processing
//
i = 0;                                              // initialize file name index
while((ch = getkey()) != 0x1B) {                    // wait for an <Esc>
    if(ch == 0x0D) {                                // check for <CR>
          printf("\x1B[7m");                        // set to inverse video
          printf("%s", file_name[i]);               // tag file name
          printf("    ");                           // cover up pointer
          strcpy(tag_name[tagcnt++], file_name[i]); // save tagged file name

          if(linenum == 23) linenum = 3;            // bottom line, wrap around
          else linenum++;                           // otherwise, move down one line

          if(i == filecnt - 1) {                    // check for last file
            i = 0;                                  // go to first file
            linenum = 3;                            // move cursor to top line
            }                                       // end of if
          else i++;                                 // not last file

          switch(i/21) {                            // 21 file names per column
            case 0 :                                // check for first column
                  column = 3;                       // cursor position
                  break;                            // break
            case 1 :                                // check for second column
                  column = 23;                      // cursor position
                  break;                            // break
            case 2 :                                // check for third column
                  column = 43;                      // cursor position
                  break;                            // break
            case 3 :                                // check for fourth column
                  column = 63;                      // cursor position
            }                                       // end of switch
```

```
        mov_curs(linenum,column);                   // move cursor
        printf("\x1B[0m");                          // set to normal video
        printf("%s<===", file_name[i]);             // re-print file name
        mov_curs(linenum,column);                   // move cursor
        }                                           // end of if
    if(ch == 0) {                                   // check for arrow key
        ch = getkey();                              // get second character
        if((ch == 0x48) || (ch == 0x50)) {          // check for up or down arrow
          printf("%s    ", file_name[i]);           // re-print current line

          if(ch == 0x48) {                          // check for up arrow
                if(linenum == 3) linenum = 23;      // top line, wrap around
                else linenum--;                     // otherwise, move up one line

                if(i == 0) {                        // check for first file
                  i = filecnt - 1;                  // go to last file
                  linenum = filecnt%21 + 2;         // calculate line number
                  }                                 // end of if
                else i--;                           // not first file
                }                                   // end of if

          else {                                    // check for down arrow
                if(linenum == 23) linenum = 3;      // bottom line, wrap around
                else linenum++;                     // otherwise, move down one line

                if(i == filecnt - 1) {              // check for last file
                  i = 0;                            // go to first file
                  linenum = 3;                      // move cursor to top line
                  }                                 // end of if
                else i++;                           // not last file
                }                                   // end of if

          switch(i/21) {                            // 21 file names per column
                case 0 :                            // check for first column
                  column = 3;                       // cursor position
                  break;                            // break
                case 1 :                            // check for second column
                  column = 23;                      // cursor position
                  break;                            // break
                case 2 :                            // check for third column
                  column = 43;                      // cursor position
                  break;                            // break
                case 3 :                            // check for fourth column
                  column = 63;                      // cursor position
                }                                   // end of switch

          mov_curs(linenum,column);                 // move cursor
          printf("%s<===", file_name[i]);           // re-print file name
          mov_curs(linenum,column);                 // move cursor
          }                                         // end of if
        }                                           // end of if(ch == 0)
    }                                               // end of while
fp_report = fopen("DATA\\REPORT.TXT", "wt");        // overwrite old report file
```

```
fclose(fp_report);                          // close report file
//
// process the selected raw data files
//
for(i = 0; i < tagcnt; i++) {               // process each tagged file
   for(j = 0; tag_name[i][j] != '.'; j++)   // mask off extension
         name[j] = tag_name[i][j];          // install one char at a time
   name[j] = '\0';                          // install terminator

   strcpy(buffer, name);                    // install file name in buffer
   strcat(buffer, ".RAW");                  // add extension to file name
   if(raw(buffer))                          // process raw data file
         continue;                          // error in processing

   strcpy(buffer, name);                    // install file name in buffer

   strcat(buffer, "L.SLP");                 // add extension to file name
   slope(buffer);                           // process slope profile data

   strcpy(buffer, name);                    // install file name in buffer
   strcat(buffer, "R.SLP");                 // add extension to file name
   slope(buffer);                           // process slope profile data
   }                                        // end of for

return(0);                                  // return file name
}                                           // end of getbat()
```

```
// raw.c
// raw sensor data processing code
//
// Description          :   This program processes raw sensor data into
//                              left and right slope profile files
//
// Functions Included   :       raw(), convert()
//
// External Functions   :       None
//
// Programmer           :       Greg Larson
//
// Created              :       06/23/92
//
// Last Revised         :       01/10/94
//
// NOTES:
// 6350 mps-clock counts = 0.0254 m X 250000 clock counts/sec
// 1.9154E-3 (m/sec**2)/count = (2 g)/(5 volts) X (1 volt)/(10 volts)
//                              X 4.8828E-3 volts/count X (9.80665 m/sec**2)/g
//
// LONGWAVE = 1/3 of the longest wavelength of interest, or about 50 meters
// COFINT = 1 - DELTAX/LONGWAVE = 1 - 0.3048/50 = 0.993904
//
// 528 samples per tenth of a mile; process data in tenth of a mile increments
//
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <io.h>
//
#define DELTAX          0.3048          // one inch in meters
#define COFINT          0.993904        // see notes above
#define HEADER_SIZE     10L             // processed flag, time of day
#define BUF_SIZE        2640L           // 5 words X 528 samples
#define RAWPATH         "RAW\\"         // path for raw files
#define DATAPATH        "DATA\\"        // path for data files
#define SLOPEPATH       "SLOPE\\"       // path for slope profile files
#define CONVACC         1.9154E-3;      // convert accel to m/sec**2
#define LIMIT           1.0             // limit value, meters/second
#define SAMPLES         528             // number of samples per tenth

float limit(float new, float old, float limval);
//_____________________________________________________________________
int raw(char file_name[])
{                                       // start of raw()
void convert(FILE *fp_raw, char name[], int numbuf);

FILE *fp_report;                        // file pointer, report file
FILE *fp_raw;                           // file pointer, raw data
FILE *fp_slpl, *fp_slpr;                // file pointer, slope data
FILE *fp_vel;                           // file pointer, velocity
FILE *fp_accl, *fp_accr;                // file pointer, acceleration
FILE *fp_hgtl, *fp_hgtr;                // file pointer, height
```

```
char buffer[25] = DATAPATH;                    // path buffer
char name[9];                                  // file name buffer
char processed = 0xFF;                         // file processed flag
long rawlength;                                // length of raw data file
float *slpbuf, *slpptr;                        // memory buffers
float *velbuf, *velptr;                        //
float *accbuf, *accptr;                        //
float *hgtbuf;                                 //
float vel, vel2;                               // vehicle velocity, m/sec
float accel, accelx;                           // time, distance based accel.
float accslope;                                // accelerometer integration
float hgtslope;                                // height differentiation
int i, j;                                      // loop index
int numbuf;                                    // number of buffers
int percent;                                   // percent completed

printf("\x1B[2J");                             // clear screen
strcat(buffer, "REPORT.TXT");                  // add file name to path
fp_report = fopen(buffer, "at+");              // open report file
fprintf(fp_report, "\nProcessing %s sensor data file\n", file_name);
printf("Processing %s sensor data file\n\n", file_name);        // print msg

strcpy(buffer, RAWPATH);                       // copy path into buffer
strcat(buffer, file_name);                     // add file name to path
fp_raw = fopen(buffer, "rb+");                 // open raw data file
rawlength = filelength(fileno(fp_raw)) - HEADER_SIZE;     // file length, bytes
numbuf = rawlength/(2*BUF_SIZE);               // get number of 0.1 mile bufs
if(numbuf == 0) {                              // check for small file
   printf("   File is too small! (less than 0.1 mile of data)");
   fprintf(fp_report, "   File is too small! (less than 0.1 mile of data)");
   fclose(fp_raw);                             // close raw data file
   fclose(fp_report);                          // close report file
   delay(3000);                                // pause for 3 seconds
   return(1);                                  // return to process menu
   }                                           // end of if

fwrite(&processed, 1, 1, fp_raw);              // say file has been processed
processed = 0;                                 // reset processed flag
for(i = 0; file_name[i] != '.'; i++)           // mask off file extension
   name[i] = file_name[i];                     // copy file name into buffer
name[i] = '\0';                                // install null terminator

fprintf(fp_report, "Converting raw data to engineering units\n");
convert(fp_raw, name, numbuf);                 // convert raw data to metric

slpbuf = (float *)malloc(SAMPLES*4);           // allocate memory in bytes
velbuf = (float *)malloc(SAMPLES*4);           //
accbuf = (float *)malloc(SAMPLES*4);           //
hgtbuf = (float *)malloc(SAMPLES*4+4);         // get extra sample for diff.

strcpy(buffer, DATAPATH);                      // copy path into buffer
strcat(buffer, name);                          // add file name to path
strcat(buffer, ".VEL");                        // add extension to file name
fp_vel = fopen(buffer, "rb");                  // open vehicle velocity file
```

```
printf("\n\nComputing left slope profile\n\n");        // print message
fprintf(fp_report, "Computing left slope profile\n");
strcpy(buffer, DATAPATH);                               // copy path into buffer
strcat(buffer, name);                                   // add file name to path
strcat(buffer, "L.ACC");                                // add extension to file name
fp_accl = fopen(buffer, "rb");                          // open left accelerometer file
strcpy(buffer, DATAPATH);                               // copy path into buffer
strcat(buffer, name);                                   // add file name to path
strcat(buffer, "L.HGT");                                // add extension to file name
fp_hgtl = fopen(buffer, "rb");                          // open left height file
strcpy(buffer, SLOPEPATH);                              // copy path into buffer
strcat(buffer, name);                                   // add file name to path
strcat(buffer, "L.SLP");                                // add extension to file name
fp_slpl = fopen(buffer, "wb");                          // open left slope data file
fwrite(&processed, 1, 1, fp_slpl);                      // write file not processed flag
fwrite(&processed, 1, 1, fp_slpl);                      // align data for viewing!
fwrite(&processed, 1, 1, fp_slpl);                      //
fwrite(&processed, 1, 1, fp_slpl);                      //

accslope = 0.0;                                         // integration variable
fread(accbuf, 4, 2, fp_accl);                           // read two dummy accel values
printf("0   percent completed...\r");                   // print message
for(i = 1; i <= numbuf; i++) {                          // process each 0.1 mile buffer
   fread(velbuf, 4, SAMPLES, fp_vel);                   // read velocity data
   fread(accbuf, 4, SAMPLES, fp_accl);                  // read accelerometer data
   fread(hgtbuf, 4, SAMPLES + 1, fp_hgtl);              // read height data

   velptr = velbuf;                                     // point to first data point
   accptr = accbuf;                                     //

   for(j = 0, slpptr = slpbuf; j < SAMPLES; j++, slpptr++) {
         vel = *(velptr++);                             // velocity in m/sec
         vel2 = vel*vel;                                // velocity squared
         accel = *(accptr++);                           // acceleration in m/sec**2
         accelx = accel/vel2;                           // time to distance-based
         accslope = COFINT*accslope + accelx*DELTAX;    // integrate accel
         hgtslope = (*(hgtbuf+j+1)*COFINT - *(hgtbuf+j))/DELTAX;
         *slpptr = -(accslope + hgtslope);              // compute left slope profile
         }                                              // end of for

   fwrite(slpbuf, 4, SAMPLES, fp_slpl);                 // save left slope profile data
   fseek(fp_hgtl, -4L, SEEK_CUR);                       // back-up pointer one sample
   percent = (int)(((float)i/numbuf)*100);              // calculate percent completed
   printf("%d\r", percent);                             // print message
   }                                                    // end of for
fclose(fp_slpl);                                        // close data file
rewind(fp_vel);                                         // move pointer to start
fclose(fp_accl);                                        // close data files
fclose(fp_hgtl);                                        //

printf("\n\nComputing right slope profile\n\n");        // print message
fprintf(fp_report, "Computing right slope profile\n");
strcpy(buffer, DATAPATH);                               // copy path into buffer
```

```
strcat(buffer, name);                          // add file name to path
strcat(buffer, "R.ACC");                       // add extension to file name
fp_accr = fopen(buffer, "rb");                 // open left accelerometer file
strcpy(buffer, DATAPATH);                      // copy path into buffer
strcat(buffer, name);                          // add file name to path
strcat(buffer, "R.HGT");                       // add extension to file name
fp_hgtr = fopen(buffer, "rb");                 // open left height file
strcpy(buffer, SLOPEPATH);                     // copy path into buffer
strcat(buffer, name);                          // add file name to path
strcat(buffer, "R.SLP");                       // add extension to file name
fp_slpr = fopen(buffer, "wb");                 // open left slope data file
fwrite(&processed, 1, 1, fp_slpr);             // write file not processed flag
fwrite(&processed, 1, 1, fp_slpr);             // align data for viewing
fwrite(&processed, 1, 1, fp_slpr);             //
fwrite(&processed, 1, 1, fp_slpr);             //

accslope = 0.0;                                // integration variable
fread(accbuf, 4, 2, fp_accr);                  // read two dummy accel values
printf("0   percent completed...\r");          // print message
for(i = 1; i <= numbuf; i++) {                 // process each 0.1 mile buffer
   fread(velbuf, 4, SAMPLES, fp_vel);          // read velocity data
   fread(accbuf, 4, SAMPLES, fp_accr);         // read accelerometer data
   fread(hgtbuf, 4, SAMPLES + 1, fp_hgtr);     // read height data

   velptr = velbuf;                            // point to first data point
   accptr = accbuf;                            //

   for(j = 0, slpptr = slpbuf; j < SAMPLES; j++, slpptr++) {
         vel = *(velptr++);                    // velocity in m/sec
         vel2 = vel*vel;                       // velocity squared
         accel = *(accptr++);                  // acceleration in m/sec**2
         accelx = accel/vel2;                  // time to distance-based
         accslope = COFINT*accslope + accelx*DELTAX;   // integrate accel
         hgtslope = (*(hgtbuf+j+1)*COFINT - *(hgtbuf+j))/DELTAX;
         *slpptr = -(accslope + hgtslope);     // compute right slope profile
         }                                     // end of for

   fwrite(slpbuf, 4, SAMPLES, fp_slpr);        // save right slope profile data
   fseek(fp_hgtr, -4L, SEEK_CUR);              // back-up pointer one sample
   percent = (int)(((float)i/numbuf)*100);     // calculate percent completed
   printf("%d\r", percent);                    // print message
   }                                           // end of for
fclose(fp_slpr);                               // close data files
fclose(fp_vel);                                //
fclose(fp_accr);                               //
fclose(fp_hgtr);                               //


free(slpbuf);                                  // release memory
free(velbuf);                                  //
free(accbuf);                                  //
free(hgtbuf);                                  //

printf("\n\nProcessing completed!");           // print message
```

```
fprintf(fp_report, "Processing completed!\n");      //
fclose(fp_report);                                  // close report file
delay(2000);                                        // pause for two seconds
return(0);                                          // normal return
}                                                   // end of raw()
//_____________________________________________________________________
void convert(FILE *fp_raw, char name[], int numbuf)
{                                                   // start of convert()
FILE *fp_vel;                                       // file pointer, velocity data
FILE *fp_accl, *fp_hgtl;                            // file pointer, accel, height
FILE *fp_accr, *fp_hgtr;                            // file pointer, accel, height
char buffer[25] = DATAPATH;                         // path buffer
int *rawbuf, *rawend, *rawptr;                      // buffer, pointer to buffer
float *velbuf, *velptr;                             // velocity buffer, pointer
float *accbuf, *accptr;                             // accelerometer buffer, pointer
float *hgtbuf, *hgtptr;                             // height buffer, pointer
float vel, oldvel;                                  // velocity, old velocity
float height;                                       // height
float oldlheight, oldrheight;                       // old left, right height
int rawhgt;                                         // raw height data
int i;                                              // loop index
int percent;                                        // percent completed

printf("Converting raw data to engineering units\n\n");      // print message

strcat(buffer, name);                               // add file name to path
strcat(buffer, ".VEL");                             // add extension to file name
fp_vel  = fopen(buffer, "wb");                      // open velocity data file

strcpy(buffer, DATAPATH);                           // copy path into buffer
strcat(buffer, name);                               // add file name to path
strcat(buffer, "L.ACC");                            // add extension to file name
fp_accl = fopen(buffer, "wb");                      // open left accel data file

strcpy(buffer, DATAPATH);                           // copy path into buffer
strcat(buffer, name);                               // add file name to path
strcat(buffer, "L.HGT");                            // add extension to file name
fp_hgtl = fopen(buffer, "wb");                      // open left height data file

strcpy(buffer, DATAPATH);                           // copy path into buffer
strcat(buffer, name);                               // add file name to path
strcat(buffer, "R.ACC");                            // add extension to file name
fp_accr = fopen(buffer, "wb");                      // open right accel data file

strcpy(buffer, DATAPATH);                           // copy path into buffer
strcat(buffer, name);                               // add file name to path
strcat(buffer, "R.HGT");                            // add extension to file name
fp_hgtr = fopen(buffer, "wb");                      // open right height data file

rawbuf = (int *)malloc(SAMPLES*5*2);                // allocate memory in bytes
rawend = rawbuf + BUF_SIZE;                         // end of raw buffer
velbuf = (float *)malloc(SAMPLES*4);                // allocate memory in bytes
accbuf = (float *)malloc(SAMPLES*4);                //
hgtbuf = (float *)malloc(SAMPLES*4);                //
```

```
oldvel = 25.4;                                  // initialize old velocity
oldlheight = 0.0;                               // initialize old left height
oldrheight = 0.0;                               // initialize old right height
rewind(fp_raw);                                 // point to start of file
fread(buffer, 1, HEADER_SIZE, fp_raw);          // remove header data
printf("0   percent completed...\r");           // print message
for(i = 1; i <= numbuf; i++) {                  // process each 0.1 mile buffer
   fread(rawbuf, 2, BUF_SIZE, fp_raw);          // read raw data into buffer
//
// convert vehicle velocity data
//
   for(velptr = velbuf, rawptr = rawbuf; rawptr < rawend; rawptr = rawptr+5) {
         vel = 6350.0/(*rawptr);                // convert velocity data, m/sec
         vel = limit(vel, oldvel, 1.0);         // limit change in velocity
         *(velptr++) = vel;                     // install data in buffer
         oldvel = vel;                          // update last limited velocity
         }                                      // end of for
   fwrite(velbuf, 4, SAMPLES, fp_vel);          // save velocity buffer
//
// convert left accelerometer and height sensor data
//
   accptr = accbuf;                             // initialize buffer pointer
   hgtptr = hgtbuf;                             // initialize buffer pointer
   for(rawptr = rawbuf + 1; rawptr < rawend; rawptr = rawptr + 5) {
         *(accptr++) = (*rawptr-0x0800)*CONVACC;        // convert accel to m/sec**2
         rawhgt = *(rawptr+1);                  // get raw height data
         height = (rawhgt - 0x0800)*31.25E-6;   // convert height to meters
         height = limit(height, oldlheight, 0.006);  // limit height
         *(hgtptr++) = height;                  // save height in meters
         oldlheight = height;                   // update old limited height
         }                                      // end of for
   fwrite(accbuf, 4, SAMPLES, fp_accl);         // save left accel buffer
   fwrite(hgtbuf, 4, SAMPLES, fp_hgtl);         // save left height buffer

   if(i == numbuf) {                            // check for last buffer
         accptr--;                              // backup pointer one sample
         fwrite(accptr, 4, 1, fp_accl);         // save one dummy sample
         fwrite(accptr, 4, 1, fp_accl);         // save another dummy sample
         fwrite(&height, 4, 1, fp_hgtl);        // save one dummy sample
         }                                      // end of if
//
// convert right acelerometer and height sensor data
//
   accptr = accbuf;                             // initialize buffer pointer
   hgtptr = hgtbuf;                             // initialize buffer pointer
   for(rawptr = rawbuf + 3; rawptr < rawend; rawptr = rawptr + 5) {
         *(accptr++) = (*rawptr-0x0800)*CONVACC;        // convert accel to m/sec**2
         rawhgt = *(rawptr+1);                  // get raw height data
         height = (rawhgt - 0x0800)*31.25E-6;   // convert height to meters
         height = limit(height, oldrheight, 0.006);  // limit height
         *(hgtptr++) = height;                  // save height in meters
         oldrheight = height;                   // update old limited height
         }                                      // end of for
```

```
   fwrite(accbuf, 4, SAMPLES, fp_accr);          // save right accel buffer
   fwrite(hgtbuf, 4, SAMPLES, fp_hgtr);          // save right height buffer

   if(i == numbuf) {                             // check for last buffer
         accptr--;                               // backup pointer one sample
         fwrite(accptr, 4, 1, fp_accr);          // save one dummy sample
         fwrite(accptr, 4, 1, fp_accr);          // save another dummy sample
         fwrite(&height, 4, 1, fp_hgtr);         // save one dummy sample
         }                                       // end of if

   percent = (int)(((float)i/numbuf)*100);       // calculate percent completed
   printf("%d\r", percent);                      // print message
   }                                             // end of for

fclose(fp_raw);                                  // close files
fclose(fp_vel);                                  //
fclose(fp_accl);                                 //
fclose(fp_hgtl);                                 //
fclose(fp_accr);                                 //
fclose(fp_hgtr);                                 //

free(rawbuf);                                    // release memory
free(velbuf);
free(accbuf);
free(hgtbuf);
}                                                // end of convert()
//-----------------------------------------------------------------------
float limit(float new, float old, float limval)
{                                                // start of limit()
if(new > old) {                                  // check for new bigger
   if(new - old > limval)                        // check for change > limit
         return(old + limval);                   // return limited change
   }                                             // end of if
else {                                           // old is bigger
   if(old - new > limval)                        // check for change > limit
         return(old - limval);                   // return limited change
   }                                             // end of else
return(new);                                     // return unchanged value
}                                                // end of limit()
```

```
// slope.c
// data processing source code for High Speed Pavement Profilometer Project
//
// Description          :    This program processes slope profile data
//                               into elevation profile and IRI files
//
// Functions Included   :    slope()
//
// External Functions   :    None
//
// Programmer           :    Greg Larson
//
// Created              :    06/23/92
//
// Last Revised         :    05/18/93
//
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <alloc.h>
#include <math.h>

#define COFINT          0.993904            // see notes in raw.c
#define DELTAX          0.3048              // sample interval in meters
#define SAMPLES_11M     (11/DELTAX) + 1     // number of samples in 11 m
#define SAMPLES         528                 // number of samples per tenth
#define SAMPLES_L       528L                //

#define RAWPATH         "RAW\\"             // path for raw files
#define DATAPATH        "DATA\\"            // path for data files
#define SLOPEPATH       "SLOPE\\"           // path for slope profiles
#define  ELEVPATH       "ELEV\\"            // path for elevation profiles
#define IRIPATH         "IRI\\"             // path for IRI files

extern void init_STM(float STM[4][4], float PRM[4]);

//_____________________________________________________________________
int slope(char file_name[])
{
void elevation(FILE *fp_slope, char name[], int numbuf);
void roughness(FILE *fp_slope, char name[], int numbuf);

FILE *fp_report;                            // file pointer, report file
FILE *fp_slope;                             // file pointer, slope profile
char name[9];                               // file name buffer
char buffer[25] = DATAPATH;                 // path buffer
long slplength;                             // length of slope profile file
int numbuf;                                 // number of buffers
char processed = 0xFF;                      // file already processed flag
int i;                                      // loop index

printf("\x1B[2J");                          // clear screen
strcat(buffer, "REPORT.TXT");               // add file name to path
fp_report = fopen(buffer, "at+");           // open report file
```

```
fprintf(fp_report, "\nProcessing %s slope profile data file\n", file_name);
printf("Processing %s slope profile data file\n\n", file_name); // print msg

strcpy(buffer, SLOPEPATH);                          // copy path into buffer
strcat(buffer, file_name);                          // add file name to path
fp_slope = fopen(buffer, "rb+");                    // open slope profile data file
fwrite(&processed, 1, 1, fp_slope);                 // say file has been processed
slplength = filelength(fileno(fp_slope)) - 4L;      // file length, bytes
numbuf = slplength/(SAMPLES*4);                     // number of 0.1 mile buffers

for(i = 0; file_name[i] != '.'; i++)                // mask off file extension
   name[i] = file_name[i];                          // copy file name into buffer
name[i] = '\0';                                     // install null terminator

fprintf(fp_report, "Computing elevation profile\n");
elevation(fp_slope, name, numbuf);                  // compute elevation profile
fprintf(fp_report, "Computing IRI\n");
roughness(fp_slope, name, numbuf);                  // compute roughness data

fclose(fp_slope);                                   // close slope profile file
printf("\n\nProcessing completed!");                // print message
fprintf(fp_report, "Processing completed!\n");      //
fclose(fp_report);                                  // close report file
delay(2000);                                        // pause
return(0);                                          // normal return
}                                                   // end of slope()
//_____________________________________________________________________
void elevation(FILE *fp_slope, char name[], int numbuf)
{
FILE *fp_elev;                                      // file pointer, elevation
char buffer[25] = ELEVPATH;                         // file name buffer
float *slpbuf, *slpptr, *slpend;                    // buffer, pointer to buffer
float elevpro;                                      // elev profile integration var
int percent;                                        // percent completed
int i;                                              // loop index

printf("Computing elevation profile\n\n");          // print message

strcat(buffer, name);                               // add file name to path
strcat(buffer, ".ELV");                             // add extension to file name
fp_elev = fopen(buffer, "wb");                      // open elevation profile file

slpbuf = (float *)malloc(SAMPLES*4);                // allocate memory in bytes
slpend = slpbuf + SAMPLES - 1;                      // end of buffer
for(slpptr = slpbuf; slpptr <= slpend; slpptr++)
   *slpptr = 0.0;                                   // clear data buffer
for(i = 1; i <= numbuf; i++)                        // loop numbuf times
   fwrite(slpbuf, 4, SAMPLES, fp_elev);             // create file of cleared data

fseek(fp_slope, 0L, SEEK_END);                      // point to end of file

printf("0   percent completed...\r");               // print message
for(i = 1; i <= numbuf; i++) {                      // process each 0.1 mile buffer
   fseek(fp_elev, -1*4*SAMPLES_L, SEEK_CUR);        // move back one buffer size
```

```
  fseek(fp_slope, -1*4*SAMPLES_L, SEEK_CUR);// move back one buffer size
  fread(slpbuf, 4, SAMPLES, fp_slope);          // read slope profile data
  for(slpptr = slpend; slpptr >= slpbuf; slpptr--) {
        if(i == 1 && slpptr == slpend)          // check for end point
           elevpro = *slpend;                   // initialize end point
        elevpro = COFINT*elevpro + (*slpptr)*DELTAX;   // integrate slope
        *slpptr = elevpro;                      // save slope profile in buffer
        }                                       // end of for
  fwrite(slpbuf, 4, SAMPLES, fp_elev);          // store elevation profile data
  fseek(fp_elev, -1*4*SAMPLES_L, SEEK_CUR);     // move back one buffer size
  fseek(fp_slope, -1*4*SAMPLES_L, SEEK_CUR);// move back one buffer size
  percent = (int)(((float)i/numbuf)*100);       // calculate percent completed
  printf("%d\r", percent);                      // print message
  }                                             // end of for

free(slpbuf);                                   // release memory
fclose(fp_elev);                                // close elevation data file
}                                               // end of elevation()
//_____________________________________________________________________
void roughness(FILE *fp_slope, char name[], int numbuf)
{                                               // start of slope()
FILE *fp_iri, *fp_set, *fp_key;                 // file pointers
char buffer[25] = IRIPATH;                      // file name buffer
char textbuf[80];                               // temporary buffer

float beginODM, endODM;                         // beginning, ending odometer

float *slpbuf, *slpptr, *slpend;                // buffer, pointer to buffer

float x1, x2, x3, x4;                           // old response variables
float x1n, x2n, x3n, x4n;                       // new response variables

float STM[4][4];                                // state transition matrix
float PRM[4];                                   // particular response matrix

float sp, rough;                                // slope profile, roughness
float diff;                                     // x1 - x3

char processed;                                 // file already processed flag
int iri;                                        // iri
int percent;                                    // percent completed
int i, j;                                       // loop index

printf("\n\nComputing IRI\n\n");                // print message
strcat(buffer, name);                           // add file name to path
strcat(buffer, ".IRI");                         // add extension to file name
fp_iri = fopen(buffer, "wt");                   // open IRI data file

name[strlen(name) - 1] = '\0';                  // remove wheelpath letter

strcpy(buffer, RAWPATH);                        // copy path into buffer
strcat(buffer, name);                           // add file name to path
strcat(buffer, ".SET");                         // add extension to file name
fp_set = fopen(buffer, "rt");                   // open SET data file
```

```
while(fgets(textbuf, 80, fp_set) != 0)          // read setup file
   fprintf(fp_iri, "%s", textbuf);              // copy setup data to iri file
fprintf(fp_iri, "\n");                          // put in blank line

rewind(fp_set);                                 // re-initialize pointer
for(i = 0; i < 12; i++)                         // get first 12 entries
   fgets(textbuf, 80, fp_set);                  // read one entry
for(i = 11, j = 0; i < strlen(textbuf); i++, j++)  // point to begin ODM
   buffer[j] = textbuf[i];                      // install begin ODM in buffer
buffer[j] = '\0';                               // install null terminator
beginODM = atof(buffer);                        // initialize begin ODM
fgets(textbuf, 80, fp_set);                     // read End PM
fgets(textbuf, 80, fp_set);                     // read end ODM
for(i = 11, j = 0; i < strlen(textbuf); i++, j++)  // point to end ODM
   buffer[j] = textbuf[i];                      // install end ODM in buffer
buffer[j] = '\0';                               // install null terminator
endODM = atof(buffer);                          // initialize end ODM

strcpy(buffer, RAWPATH);                        // copy path into buffer
strcat(buffer, name);                           // add file name to path
strcat(buffer, ".KEY");                         // add extension to file name
fp_key = fopen(buffer, "rt");                   // open KEY data file

while(fgets(textbuf, 80, fp_key) != 0)          // read keyboard file
   fprintf(fp_iri, "%s", textbuf);              // copy key data to iri file
fprintf(fp_iri, "\n\n");                        // put in blank lines

rewind(fp_slope);                               // point to start of file
fread(&processed, 1, 1, fp_slope);              // point past processed flag
fread(&processed, 1, 1, fp_slope);              // dummy reads!
fread(&processed, 1, 1, fp_slope);              //
fread(&processed, 1, 1, fp_slope);              //

slpbuf = (float *)malloc(SAMPLES*4);            // allocate memory in bytes
slpend = slpbuf + SAMPLES - 1;                  // end of buffer
fread(slpbuf, 4, SAMPLES, fp_slope);            // read first 0.1 mile buffer

for(i = 0, x1 = 0; i < SAMPLES_11M; i++)
   x1 = x1 + *(slpbuf+i);                       // sum slopes for first 11 m
x1 = x1/i;                                      // get average slope
x2 = 0;                                         // initialize other variables
x3 = x1;                                        //
x4 = 0;                                         //

init_STM(STM, PRM);                             // initialize STM, PRM matrices

rewind(fp_slope);                               // point to start of file
fread(&processed, 1, 1, fp_slope);              // point past processed flag
fread(&processed, 1, 1, fp_slope);              // dummy reads!
fread(&processed, 1, 1, fp_slope);              //
fread(&processed, 1, 1, fp_slope);              //

printf("0 percent completed...\r", percent);    // print message
```

```
for(i = 1; i <= numbuf; i++) {                          // process each 0.1 mile buffer
   rough = 0;                                           // initialize roughness
   fread(slpbuf, 4, SAMPLES, fp_slope);                 // read slope profile data
   for(slpptr = slpbuf; slpptr <= slpend; slpptr++) {
         sp = *slpptr;                                  // get the slope profile sample
         x1n = x1*STM[0][0]+x2*STM[0][1]+x3*STM[0][2]+x4*STM[0][3]+PRM[0]*sp;
         x2n = x1*STM[1][0]+x2*STM[1][1]+x3*STM[1][2]+x4*STM[1][3]+PRM[1]*sp;
         x3n = x1*STM[2][0]+x2*STM[2][1]+x3*STM[2][2]+x4*STM[2][3]+PRM[2]*sp;
         x4n = x1*STM[3][0]+x2*STM[3][1]+x3*STM[3][2]+x4*STM[3][3]+PRM[3]*sp;
         x1 = x1n;                                      // update response variables
         x2 = x2n;                                      //
         x3 = x3n;                                      //
         x4 = x4n;                                      //
         diff = (x1 - x3 < 0) ? -(x1 - x3): (x1 - x3);  // absolute value
         rough = rough + DELTAX*diff;                   // update roughness variable
         }                                              // end of for
   iri = rough*39.3696*10;                              // convert to inches/mile
   if(endODM > beginODM)                                // check for PM increasing
         fprintf(fp_iri, "%5.2f   %3d   %6.3f\n", i/10.0, iri, beginODM + i/10.0);
   else
         fprintf(fp_iri, "%5.2f   %3d   %6.3f\n", i/10.0, iri, beginODM - i/10.0);
   percent = (int)(((float)i/numbuf)*100);              // calculate percent completed
   printf("%d\r", percent);                             // print message
   }                                                    // end of for

fclose(fp_iri);                                         // close data files
fclose(fp_set);                                         //
fclose(fp_key);                                         //
free(slpbuf);                                           // release memory
}                                                       // end of roughness()
```

```
// init_STM.c
// initialize the state transition and particular response matrices
//
// Description          :     initialize the ST and PR matrices
//
// Functions Included   :     init_STM()
//
// External Functions   :     None
//
// Created              :     02/16/93
//
// Last Revised         :     04/13/93
//
#define  K1     653.0                        // sec**-2
#define  K2     63.3                         // sec**-2
#define  U      0.15                         // no units
#define  C      6.0                          // sec**-1

#define DX      0.3048                       // sample distance, meters
#define DT      (DX*3600*0.001)/80           // convert sample dist to time

//_____________________________________________________________________
void init_STM(float STM[4][4], float PRM[4])
{                                            // start of init_STM()
void matrix_copy(float source[4][4], float dest[4][4]);
void matrix_scale(float matrix[4][4], float scale);
void matrix_add(float source[4][4], float dest[4][4]);
void matrix_mult(float source[4][4], float dest[4][4]);

float A[4][4], A_inv[4][4];                  // A and A**-1 matrices
float Adt[4][4], update[4][4];               // A*dt, temporary matrices
float inter[4];                              // intermediate value
float B30, sum;                              // B matrix, multiply variable
float dt = DT;                               // sample time
char i, j;                                   // loop indices

for(j = 0; j < 4; j++) {                     // initialize working matrices
   for(i = 0; i < 4; i++) {                  //
         A[i][j] = 0;                        // clear A matrix for now
         A_inv[i][j] = 0;                    // clear A inverse matrix
         STM[i][j] = 0;                      // clear STM matrix
         }                                   // end of for
   STM[j][j] = 1;                            // STM = Identity Matrix
   }                                         // end of for

A[0][1] = 1;                                 // initialize A matrix
A[1][0] = -K2;                               //
A[1][1] = -C;                                //
A[1][2] = K2;                                //
A[1][3] = C;                                 //
A[2][3] = 1;                                 //
A[3][0] = K2/U;                              //
A[3][1] = C/U;                               //
A[3][2] = -(K1+K2)/U;                        //
```

```
A[3][3] = -C/U;                          //

A_inv[0][0] = -0.09478673;               // values are specific to A
A_inv[0][1] = -0.01732918;               //
A_inv[0][2] =  0.09478673;               //
A_inv[0][3] = -0.00022971;               //
A_inv[1][0] =  1.00000000;               //
A_inv[2][1] = -0.00153139;               //
A_inv[2][3] = -0.00022971;               //
A_inv[3][2] =  1.00000000;               //

matrix_copy(A, Adt);                     // A --> Adt
matrix_scale(Adt, dt);                   // Adt = A*dt
matrix_add(Adt, STM);                    // STM = I + A*dt

matrix_copy(Adt, update);                // Adt --> update
for(i = 2; i < 10; i++) {                // loop until tenth term
   matrix_mult(Adt, update);             // update = update*Adt
   matrix_scale(update, (1.0/i));        // update = update/(i!)
   matrix_add(update, STM);              // sum first ten terms
   }                                     // end of for

B30 = K1/U;                              // initialize B matrix

inter[0] = STM[0][3]*B30;                // (STM - I)*B
inter[1] = STM[1][3]*B30;                //
inter[2] = STM[2][3]*B30;                //
inter[3] = (STM[3][3] - 1)*B30;          //

for(i = 0; i < 4; i++) {                 // PRM = (A**-1)*(STM - I)*B
   sum = 0.0;                            // initialize sum variable
   for(j = 0; j < 4; j++)                // matrix multiply
         sum = sum + A_inv[i][j]*inter[j]; // intermediate sum
   PRM[i] = sum;                         //
   }                                     // end of for
}                                        // end of init_STM()
//_____________________________________________________________________
void matrix_copy(float source[4][4], float dest[4][4])
{                                        // start of matrix_copy()
char i, j;                               // loop index

for(j = 0; j < 4; j++)                   // outer loop
   for(i = 0; i < 4; i++)                // inner loop
         dest[i][j] = source[i][j];      // copy contents
}                                        // end of matrix_copy()
//_____________________________________________________________________
void matrix_scale(float matrix[4][4], float scale)
{                                        // start of matrix_scale()
char i, j;                               // loop index

for(j = 0; j < 4; j++)                   // outer loop
   for(i = 0; i < 4; i++)                // inner loop
         matrix[i][j] = scale*matrix[i][j]; // scale matrix
}                                        // end of matrix_scale()
```

```
//_____________________________________________________________________
void matrix_add(float source[4][4], float dest[4][4])
{                                                   // start of matrix_add()
char i, j;                                          // loop index

for(j = 0; j < 4; j++)                              // outer loop
   for(i = 0; i < 4; i++)                           // inner loop
         dest[i][j] = source[i][j] + dest[i][j];    // add contents
}                                                   // end of matrix_add()
//_____________________________________________________________________
void matrix_mult(float source[4][4], float dest[4][4])
{                                                   // start of matrix_mult()
char i, j, k;                                       // loop index
float sum;                                          // intermediate sum
float matrix[4][4];                                 // scratch matrix

for(i = 0; i < 4; i++)                              // way outer loop
   for(j = 0; j < 4; j++) {                         // outer loop
         sum = 0.0;                                 // initialize intermediate sum
         for(k = 0; k < 4; k++)                     // inner loop
            sum = sum + source[i][k]*dest[k][j];    // intermediate sum
         matrix[i][j] = sum;                        // new value
         }                                          // end of for
matrix_copy(matrix, dest);                          // copy matrix into destination
}                                                   // end of matrix_mult()
```

```
// cnvnav.c
// convert a binary navigation data file into an ASCII data file
//
// Description          :    This program converts binary navigation
//                               data into ASCII data
//
// Functions Included   :       cnvnav()
//
// External Functions   :       None
//
// Programmer           :       Greg Larson
//
// Created              :       06/17/93
//
// Last Revised         :       06/17/93
//
#include <string.h>
#include <stdio.h>
#include <alloc.h>
#include <io.h>

#define HEADER_SIZE   8L
#define RAWPATH       "RAW\\"
#define DATAPATH      "DATA\\"

int cnvnav(char file_name[])
{                                                   // start of cnvnav()
struct    {
          unsigned long odometer;
          unsigned int x_axis;
          unsigned int y_axis;
          unsigned int rate;
          } navdata;                                // navigation data structure

char buffer[25] = RAWPATH;                          // scratch buffer
char name[9];                                       // file name buffer
unsigned char todbuf[8];                            // time of day buffer
int year;                                           // year variable
unsigned int huge *navbuf;                          // navigation data buffer
unsigned int huge *navptr;                          // must be on seperate lines!
unsigned int huge *navend;                          //
unsigned long navlength;                            // length of nav data file
char i;                                             // index variable

FILE *fp_nav, *fp_prn;                              // file pointers

printf("\x1B[2J");                                  // clear screen
printf("Converting %s navigation data file...\n\n", file_name);// print msg
strcat(buffer, file_name);                          // add file name to path
fp_nav = fopen(buffer, "rb");                       // open navigation data file

for(i = 0; file_name[i] != '.'; i++)                // mask off file extension
   name[i] = file_name[i];                          // copy file name into buffer
name[i] = '\0';                                     // install null terminator
```

```
strcpy(buffer, DATAPATH);                          // install path in buffer
strcat(buffer, name);                              // add file name to path
strcat(buffer, ".PRN");                            // add extension to file name
fp_prn = fopen(buffer, "wt+");                     // open ASCII file

fread(todbuf, 1, HEADER_SIZE, fp_nav);             // read time of day data
year = (((int)todbuf[1]) << 8) + (int)todbuf[0] - 1900;
fprintf(fp_prn, "%2d/%02d/%2d\n", todbuf[3], todbuf[2], year);
fprintf(fp_prn, "%2d:%02d:%02d.%02d\n", todbuf[5], todbuf[4], todbuf[7],
                                        todbuf[6]);

navlength = (filelength(fileno(fp_nav)) - HEADER_SIZE)/2;  // file length
navbuf = (unsigned int huge *)farmalloc(navlength * sizeof(int));
navptr = navbuf;                                   // initialize nav data pointer
fread(navbuf, 2, navlength, fp_nav);               // read entire nav data file

navdata.odometer = 0;                              // get starting point data
navdata.x_axis = *(navptr + 2);                    //
navdata.y_axis = *(navptr + 3);                    //
navdata.rate = *(navptr + 4);                      //
fprintf(fp_prn, "%7ld, %4d, %4d, %4d\n", navdata.odometer, navdata.x_axis,
                                        navdata.y_axis, navdata.rate);

navend = navbuf + navlength;                       // point to end of nav data
for(navptr = navbuf; navptr < navend; navptr = navptr + 5) {
   navdata.odometer = ((long)(*(navptr + 1)) << 16) + (long)(*navptr);
   navdata.x_axis = *(navptr + 2);
   navdata.y_axis = *(navptr + 3);
   navdata.rate = *(navptr + 4);

   fprintf(fp_prn, "%7ld, %4d, %4d, %4d\n", navdata.odometer, navdata.x_axis,
                                        navdata.y_axis, navdata.rate);
   }                                               // end of for

fclose(fp_nav);                                    // close data files
fclose(fp_prn);                                    //

free(navbuf);                                      // release memory

printf("Conversion completed!");                   // print message
delay(2000);                                       // pause for 2 seconds
return(0);                                         // normal return
}
```

```
// sys_test.c
// system test source code for the High Speed Pavement Profilometer Project
//
// Description          :     This program performs the system test
//
// Functions Included   :     sys_test(), prntest()
//
// External Functions   :     bounce(), step(), height(), wheel(), adc()
//
// Programmer           :     Greg Larson
//
// Created              :     11/04/91
//
// Last Revised         :     08/11/92
//
#include <dos.h>

#define BOUNCE      10
#define WHEEL       12
#define STEP        14
#define HEIGHT      16
#define ADC         18
#define RETURN      20
#define PROMPT      22

int sys_test(void)
{                                          // start of sys_test.c
unsigned char ch;                          // keyboard character
int linenum;                               // line number variable

while(1) {                                 // loop forever
   printf("\x1B[2J");                      // clear screen
   mov_curs(4,24);                         // move cursor to line 4
   printf("HIGH SPEED PAVEMENT PROFILOMETER"); // print out test menu
   mov_curs(8,32);
   printf("System Test Menu");
   mov_curs(BOUNCE,8);
   printf("Bounce Test");
   mov_curs(WHEEL,8);
   printf("Pulsed Wheel Sensor Test");
   mov_curs(STEP,8);
   printf("Step Input Test (Height Sensor)");
   mov_curs(HEIGHT,8);
   printf("Height Sensor Test");
   mov_curs(ADC,8);
   printf("Analog-to-Digital Converter Test");
   mov_curs(RETURN,8);
   printf("Return to the Main Menu");
   mov_curs(PROMPT,4);
   printf("Move the pointer to the desired operation and press Return...");

// highlight the first menu entry

   linenum = BOUNCE;                       // initialize line number
```

```
    mov_curs(BOUNCE,4);                         // move the cursor to line 4
    printf("==>");                              // print the pointer
    mov_curs(BOUNCE,4);                         // move cursor to start

  // highlight the menu entries as the up and down cursor arrow keys are
  // pressed; when carriage return is entered, perform the highlighted function

    while((ch = getkey()) != 0x0D)              // wait for carriage return
        if(ch == 0) {                           // check for arrow key
          ch = getkey();                        // get second character
          if((ch == 0x48) || (ch == 0x50)) {    // check for up or down arrow
              printf("  ");                     // print spaces

              if(ch == 0x48)                    // check for up arrow
                linenum = (linenum == BOUNCE) ? RETURN : linenum - 2;
              else                              // check for down arrow
                linenum = (linenum == RETURN) ? BOUNCE : linenum + 2;

              mov_curs(linenum,4);              // move cursor to new line
              printf("==>");                    // print the pointer
              mov_curs(linenum,4);              // move cursor to start
              }                                 // end of if
          }                                     // end of if
                                                // end of if

    switch(linenum) {
        case BOUNCE :                           // bounce test selected
          bounce();
          break;
        case WHEEL  :                           // wheel sensor test selected
          wheel();
          break;
        case STEP   :                           // step input test selected
          step();
          break;
        case HEIGHT :                           // height sensor test selected
          height();
          break;
        case ADC    :                           // adc test selected
          adc();
          break;
        default     :                           // return to main menu selected
          return 0;
        }                                       // end of switch
    }                                           // end of while
}                                               // end of sys_test.c
```

```
// bounce.c
// bounce test data acquisition code
//
// Description          :   This program acquires data from the sensors
//                              during the bounce test
//
// Functions Included   :       bounce(), bounce_isr()
//
// External Functions   :       None
//
// Programmer           :       Greg Larson
//
// Created              :       06/22/92
//
// Last Revised         :       05/11/93
//
#include <dos.h>
#include <stdio.h>

#define  SAMPLES      528                    // number of samples per tenth

struct probuf {                              // profile data buffer
        unsigned int velcount;               // velocity count data
        unsigned int laccel;                 // left accelerometer data
        unsigned int lheight;                // left height sensor data
        unsigned int raccel;                 // right accelerometer data
        unsigned int rheight;                // right height sensor data
        };

extern struct probuf lobuf[], hibuf[];       // lo, hi profile data buffers
extern unsigned int bufptr;                  // pointer into profile buffers
extern unsigned int laccsum, raccsum;        // left, right accel partial sum
extern unsigned int lhgtsum, rhgtsum;        // left, right height partial
extern unsigned char intcount;               // sample interval count
extern unsigned char bufflag, fullflag;      // data acquisition flags

/*****************************************************************************/
int bounce(void)
{                                            // start of bounce()
void far interrupt bounce_isr();             // acquire bounce data isr
void interrupt (*oldIRQ10)();                // old IRQ10 vector

FILE *fp_raw;                                // raw sensor data file ptr
struct date date;                            // date data structure
struct time t;                               // time data structure
char processed = 0;                          // file processed flag
char far *portCdata;                         // pointer to port C data
char far *timer1con;                         // pointer to timer 1 control
char far *load1hi, *load1mi, *load1lo;       // pointer to timer 1 load
unsigned char mask;                          // mask variable for 8259

printf("\x1B[2J");                           // clear screen
bufptr = 0;                                  // initialize buffer pointer
bufflag = 0;                                 // initialize buffer flag
```

```
fullflag = 0;                                   // initialize buffer full flag
intcount = 1;                                   // initialize interval count
laccsum = 0;                                    // initialize partial sums
lhgtsum = 0;                                    //
raccsum = 0;                                    //
rhgtsum = 0;                                    //

fp_raw = fopen("RAW\\BOUNCE.RAW", "wb");        // open raw sensor file
fwrite(&processed, 1, 1, fp_raw);               // write file processed flag
fwrite(&processed, 1, 1, fp_raw);               // dummy value to align data
getdate(&date);                                 // get the date
fwrite(&date, sizeof(date), 1, fp_raw);         // write date into file

portCdata = MK_FP(0xE000, 0xD019);              // ptr to port C data reg
timer1con = MK_FP(0xE000, 0xD021);              // ptr to timer 1 control reg
load1hi = MK_FP(0xE000, 0xD027);                // ptr to timer 1 load reg
load1mi = MK_FP(0xE000, 0xD029);                // ptr to timer 1 load reg
load1lo = MK_FP(0xE000, 0xD02B);                // ptr to timer 1 load reg

*timer1con = 0xA0;                              // periodic interrupt mode
*load1hi = 0x00;                                // load high count value
*load1mi = 0x00;                                // load middle count value
*load1lo = 0xFA;                                // load low count value, 250

mask = inportb(0xA1) | 0x04;                    // get IRQ10 interrupt mask
oldIRQ10 = getvect(0x72);                       // save old interrupt vector
setvect(0x72, bounce_isr);                      // load vector with new ISR

printf("Press \'B\' to begin the bounce test...");
while(tolower(getkey()) != 'b');                // wait for b or B pressed

printf("\x1B[2J");                              // clear screen
printf("Bounce test running...");               // print message

delay(8000);                                    // wait for operator to walk

gettime(&t);                                    // get the current time
fwrite(&t, sizeof(t), 1, fp_raw);               // write time into file

*timer1con = 0xA1;                              // start one millisecond timer
*portCdata = 0xEE;                              // enable timer interrupt
outportb(0x30, 0xAE);                           // enable VMEbus interrupts
outportb(0xA1, mask&0xFB);                      // enable IRQ10 on CPU

do                                              // data acquisition loop
  if(fullflag) {                                // check for full pro buffer
        fullflag = 0;                           // reset buffer full flag
        if(bufflag)                             // check for lo buffer
          fwrite(&lobuf, SAMPLES*5*2, 1, fp_raw);        // save low buffer
        else                                    // use hi buffer
          fwrite(&hibuf, SAMPLES*5*2, 1, fp_raw);        // save high buffer
        }                                       // end of if
  while(!chkkey());                             // wait for escape key
```

```
outportb(0xA1, mask);                    // disable IRQ10 on CPU
outportb(0x30, 0x2E);                    // disable VMEbus interrupts
*portCdata = 0xFE;                       // disable timer interrupt
*timer1con = 0xA0;                       // stop one millisecond timer

setvect(0x72, oldIRQ10);                 // restore old interrupt vector

getkey();                                // flush keyboard buffer
printf("\n\n\nBounce test completed!");  // print message
delay(2000);                             // pause for 2 seconds
fclose(fp_raw);                          // close sensor raw data file

return(0);                               // normal return to test menu
}                                        // end of bounce()

/*****************************************************************************/
void far interrupt bounce_isr(void)      // acquire bounce data isr
{                                        // start of bounce_isr
int lheight, rheight;                    // left, right height data
int laccel, raccel;                      // left, right accel data

int far *llaser, *rlaser;                // pointers to laser sensors
int far *ldata, *rdata;                  // pointers to adc data reg
char far *status, *channl;               // pointers to adc registers
char far *timerstat;                     // pointer to timer status

enable();                                // enable interrupts

timerstat = MK_FP(0xE000, 0xD035);       // make ptr to timer status
*timerstat = 0x01;                       // reset interrupt request bit

llaser = MK_FP(0xE000, 0xE400);          // make ptr to left laser
rlaser = MK_FP(0xE000, 0xE800);          // make ptr to right laser

status = MK_FP(0xE000, 0xE081);          // make ptr to adc status reg
channl = MK_FP(0xE000, 0xE085);          // make ptr to adc channel reg
rdata = MK_FP(0xE000, 0xE086);           // make ptr to adc data reg
ldata = rdata;                           // this is correct!

*channl = 0x00;                          // point to channel 0
*status = 0xA0;                          // start sequential conversion

lheight = *llaser;                       // get left laser height
lheight = lheight&0x0FFF;                // mask off data bits

while(*status < 0);                      // wait for end of conversion
laccel = *ldata;                         // get left accelerometer data

rheight = *rlaser;                       // get right laser height
rheight = rheight&0x0FFF;                // mask off data bits

while(*status < 0);                      // wait for end of conversion
raccel = *rdata;                         // get right accelerometer data
```

```
laccsum = laccsum + laccel;                    // calculate partial sums
lhgtsum = lhgtsum + lheight;                   //
raccsum = raccsum + raccel;                    //
rhgtsum = rhgtsum + rheight;                   //

intcount++;                                    // increment interval count
if(intcount > 12) {                            // check for van travelled 1'
   intcount = 1;                               // re-initialize interval count

   if(bufflag) {                               // use hi buffer
         hibuf[bufptr].velcount = 0x00FA;      // velocity count, 25.4 m/sec
         hibuf[bufptr].laccel = laccsum/12;    // left accelerometer data
         hibuf[bufptr].lheight = lhgtsum/12;   // left height data
         hibuf[bufptr].raccel = raccsum/12;    // right accelerometer data
         hibuf[bufptr].rheight = rhgtsum/12;   // right height data
         }                                     // end of if
   else {                                      // use lo buffer
         lobuf[bufptr].velcount = 0x00FA;      // velocity count, 25.4 m/sec
         lobuf[bufptr].laccel = laccsum/12;    // left accelerometer data
         lobuf[bufptr].lheight = lhgtsum/12;   // left height data
         lobuf[bufptr].raccel = raccsum/12;    // right accelerometer data
         lobuf[bufptr].rheight = rhgtsum/12;   // right height data
         }                                     // end of else

   bufptr++;                                   // increment buffer pointer
   if(bufptr >= SAMPLES) {                     // check for buffer full
         bufptr = 0;                           // reset buffer pointer
         bufflag = ~bufflag;                   // point to other buffer
         fullflag = 0xFF;                      // say buffer full
         }                                     // end of if

   laccsum = 0;                                // re-initialize partial sums
   lhgtsum = 0;                                //
   raccsum = 0;                                //
   rhgtsum = 0;                                //
   }                                           // end of if

disable();                                     // disable interrupts
outportb(0xA0, 0x20);                          // send EOI to  slave 8259
outportb(0x20, 0x20);                          // send EOI to master 8259
}                                              // end of bounce_isr
```

```
// wheel.c
// pulsed wheel sensor test
//
// Description          :   This program tests the operation of the
//                              pulsed wheel sensors
//
// Functions Included   :       wheel()
//
// External Functions   :       None
//
// Programmer           :       Greg Larson
//
// Created              :       07/09/92
//
// Last Revised         :       01/18/94
//
#include <dos.h>
#include <math.h>

#define  CONVERT1     56.82                  // convert known to mph
#define CONVERT2      14204.37               // convert measured to mph

extern unsigned long odmcount, oldodmcount;  // current, old odometer count
extern float counts_mile;                    // # of odometer counts per mile
extern unsigned long sum;                    // sum of velocity counts
extern unsigned long second;                 // second counter
extern unsigned long millisec;               // millisecond counter
extern unsigned char dispflag;               // display status flag
extern unsigned char countup;                // PM increasing flag

/*****************************************************************************/
int wheel(void)
{
void far interrupt wheel_isr();              // pulsed wheel sensor isr
void far interrupt timer_isr();              // one millisecond timer isr
void interrupt (*oldIRQ9)();                 // old IRQ9 vector
void interrupt (*oldIRQ10)();                // old IRQ10 vector

char begPM[16], endPM[16];                   // beginning, ending post mile
float elapsedM, curPM;                       // elapsed, current post mile
float speed;                                 // vehicle speed, mph
char far *resetint;                          // ptr to reset interrupt bit
char far *status;                            // ptr to status port
char far *portCdata;                         // ptr to port C data register
char far *timer1con;                         // ptr to timer 1 control reg
char far *load1hi, *load1mi, *load1lo;       // ptr to timer 1 load regs
char far *timer2con;                         // ptr to timer 2 control reg
char far *load2hi, *load2mi, *load2lo;       // ptr to timer 2 load regs
unsigned char mask;                          // mask variable for 8259
char ch = 0;                                 // input from operator
unsigned int hour, min, sec;                 // elapsed time
unsigned long left;                          // leftover
int percent;                                 // percent error
float diff;                                  // known - measured
```

```
float known = 0.0, measured = 0.0;                  // known, measured values

printf("\x1B[2J");                                  // clear screen
printf("Pulsed Wheel Sensor Test\n\n");             // print header
printf("Enter the beginning post mile >");          // prompt for post mile
getmsg(begPM);                                      // get beginning post mile
printf("Enter the ending post mile >");             // prompt for post mile
getmsg(endPM);                                      // get ending post mile

countup = 0;                                        // say PM decreasing for now
if(atof(endPM) >= atof(begPM))                      // check for PM increasing
   countup = 0xFF;                                  // say PM increasing
//
// initialize the timer and interrupts
//
resetint = MK_FP(0xE000, 0xD011);                   // make ptr to reset int bit
status = MK_FP(0xE000, 0xD013);                     // make ptr to status port
portCdata = MK_FP(0xE000, 0xD019);                  // make ptr port C data reg
timer1con = MK_FP(0xE000, 0xD021);                  // make ptr to timer 1 control
load1hi = MK_FP(0xE000, 0xD027);                    // make ptr to timer 1 load hi
load1mi = MK_FP(0xE000, 0xD029);                    // make ptr to timer 1 load mi
load1lo = MK_FP(0xE000, 0xD02B);                    // make ptr to timer 1 load lo
timer2con = MK_FP(0xE000, 0xD061);                  // make ptr to timer 2 control
load2hi = MK_FP(0xE000, 0xD067);                    // make ptr to timer 2 load hi
load2mi = MK_FP(0xE000, 0xD069);                    // make ptr to timer 2 load mid
load2lo = MK_FP(0xE000, 0xD06B);                    // make ptr to timer 2 load lo

*timer1con = 0xA0;                                  // periodic interrupt mode
*load1hi = 0x00;                                    // load high count value
*load1mi = 0x00;                                    // load middle count value
*load1lo = 0xFA;                                    // load low count value, 250

*timer2con = 0x10;                                  // elapsed time mode
*load2hi = 0xFF;                                    // load high count value
*load2mi = 0xFF;                                    // load middle count value
*load2lo = 0xFF;                                    // load low count value

odmcount = 0;                                       // initialize odometer count
oldodmcount = 0;                                    // initialize old count

second = 0;                                         // initialize second count
millisec = 0;                                       // initialize millisecond count
sum = 1;                                            // ave velocity count, avoid /0

oldIRQ9  = getvect(0x0A);                           // save old IRQ9 vector
oldIRQ10 = getvect(0x72);                           // save old IRQ10 vector
setvect(0x0A, wheel_isr);                           // initialize IRQ9 vector
setvect(0x72, timer_isr);                           // initialize IRQ10 vector
mask = inportb(0xA1);                               // get interrupt mask

printf("\x1B[2J");                                  // clear screen
printf("Press \'B\' to begin the pulsed wheel sensor test...");
while(tolower(ch) != 'b')                           // wait for b or B pressed
   if(chkkey())                                     // check for input from keybrd
```

```
            ch = getkey();                                  // get the key

        *timer1con = 0xA1;                                  // start one millisecond timer
        *portCdata = 0xEE;                                  // enable timer interrupt
        outportb(0x30, 0xAE);                               // enable VMEbus interrupts
        outportb(0xA1, mask&0xFB);                          // enable IRQ10 on CPU

        printf("\x1B[2J");                                  // clear screen
        printf(" Ending PM: %7.3f  Beginning PM: %7.3f      \nCurrent PM: ",
                atof(endPM), atof(begPM));                  // print message

        // watch the raw wheel pulse signal and wait for a rising edge before
        // enabling the distance-based interrupt

        while(*status < 0);
        while(*status >= 0);                                // wait for low raw signal
        *resetint = 0x00;                                   // detect rising edge
        *resetint = 0x01;                                   // reset distance-based int
        *timer2con = 0x11;                                  // un-reset interrupt
        mask = inportb(0xA1);                               // start velocity timer
        outportb(0xA1, mask&0xFD);                          // get interrupt mask
                                                            // enable IRQ9
        mov_curs(4,1);                                      // move cursor to line 4
        printf("Pulsed wheel sensor test running...");      // print message
        mov_curs(2,13);                                     // move cursor to line 2

        do {                                                // test loop
          if(dispflag) {                                    // check for time to display
              dispflag = 0;                                 // reset time to display flag

              elapsedM = odmcount/counts_mile;              // calculate elapsed miles
              if(countup)                                   // check for PM increasing
                 curPM = atof(begPM) + elapsedM;            // calculate current post mile
              else                                          // PM is decreasing
                 curPM = atof(begPM) - elapsedM;            // calculate current post mile
              if(curPM < 0)                                 // check for negative PM
                 curPM = 0.0;                               // force curPM to 0

              hour = second/3600;                           // calculate hours
              left = second%3600;                           // get remainder
              min = left/60;                                // calculate minutes
              sec = left%60;                                // calculate seconds

              speed = (odmcount-oldodmcount)*0.05682;       // convert pulses to mph
              oldodmcount = odmcount;                       // update old odometer count

              printf("%7.3f    Elapsed M: %7.3f  Elapsed T: %2d:%02d:%02d  Speed: %4.2f",
                      curPM, elapsedM, hour, min, sec, speed);

              mov_curs(2,13);                               // move cursor to line 2
              }                                             // end of if
        } while(!chkkey());                                 // wait for escape key

        mask = inportb(0xA1) | 0x06;                        // get disable interrupt mask
```

```
outportb(0xA1, mask);                          // disable IRQ9 and IRQ10
outportb(0x30, 0x2E);                          // disable VMEbus interrupts
*portCdata = 0xFE;                             // disable timer interrupt
*timer1con = 0xA0;                             // stop one millisecond timer
*timer2con = 0x10;                             // stop velocity timer

setvect(0x0A, oldIRQ9);                        // restore old IRQ9 vector
setvect(0x72, oldIRQ10);                       // restore old IRQ10 vector

getkey();                                      // flush keyboard buffer
printf("\n\nPulsed wheel sensor test completed!\n\n");

if(countup)                                    // check for PM increasing
   known = atof(endPM) - atof(begPM);          // calculate known distance
else                                           // PM decreasing
   known = atof(begPM) - atof(endPM);          // calculate known distance
diff = (known - elapsedM < 0) ? -(known - elapsedM) : (known - elapsedM);
percent = (int)((diff/known)*100);             // calculate percent error
printf("   Known Distance    Measured Distance    %% Error\n");
printf("      %7.3f            %7.3f            %2d\n\n",
          known, elapsedM, percent);           // print message

known = ((float)odmcount/millisec)*CONVERT1;   // calculate known average speed
measured = (odmcount*CONVERT2)/sum;            // calculate measured ave speed
diff = (known - measured < 0) ? -(known - measured) : (known - measured);
percent = (int)((diff/known)*100);             // calculate percent error
printf("    Known Speed      Measured Speed     %% Error\n");
printf("      %4.2f            %4.2f            %2d",
          known, measured, percent);           // print message

printf("\n\nThe number of elapsed odometer counts is:  %ld\n\n", odmcount);

printf("Press Escape to return to the Test Menu...");
getkey();                                      // wait for an escape
return(0);                                     // normal return to test menu
}                                              // end of wheel()
/*****************************************************************************/
void far interrupt wheel_isr(void)             // pulsed wheel sensor isr
{                                              // start of wheel_isr()
char far *resetint;                            // pointer to reset int bit
char far *timer2con;                           // pointer to timer 2 control
unsigned char far *timer2mi;                   // pointer to timer 2 middle
unsigned char far *timer2lo;                   // pointer to timer 2 low

unsigned char upper, lower;                    // upper and lower 8 bits
int count;                                     // combined upper and lower

resetint = MK_FP(0xE000, 0xD011);              // make ptr to reset int bit
*resetint = 0x00;                              // reset distance-based int
*resetint = 0x01;                              // un-reset interrupt

odmcount++;                                    // increment odometer count
```

```
timer2con = MK_FP(0xE000, 0xD061);          // make ptr to timer 2 control
timer2mi  = MK_FP(0xE000, 0xD071);          // make ptr to timer 2 middle
timer2lo  = MK_FP(0xE000, 0xD073);          // make ptr to timer 2 low

*timer2con = 0x10;                          // stop timer 2
upper = *timer2mi;                          // get upper 8 bits
lower = *timer2lo;                          // get lower 8 bits
*timer2con = 0x11;                          // re-start timer 2
count = ~((upper<<8) + lower);              // combine upper and lower
sum = sum + (long)count;                            // sum counts from each sample

outportb(0xA0, 0x20);                       // send EOI to  slave 8259
outportb(0x20, 0x20);                       // send EOI to master 8259
}                                           // end of wheel_isr()
/*****************************************************************************/
void far interrupt timer_isr(void)          // one millisecond timer isr
{                                           // start of timer_isr()
char far *timerstat;                        // pointer to timer status reg

enable();                                   // enable interrupts
timerstat = MK_FP(0xE000, 0xD035);          // make ptr to timer status reg
*timerstat = 0x01;                          // reset timer interrupt

millisec++;                                 // increment millisecond count

if(millisec%1000 == 0) {                    // check for one elapsed second
   second++;                                // increment second count
   dispflag = 0xFF;                         // say time to display status
   }                                        // end of if

disable();                                  // disable interrupts
outportb(0xA0, 0x20);                       // send EOI to  slave 8259
outportb(0x20, 0x20);                       // send EOI to master 8259
}                                           // end of timer_isr()
```

```
// step.c
// height sensor step input test routine
//
// Description          :     This program tests the accuracy of the
//                            laser height sensors
//
// Functions Included   :     step()
//
// External Functions   :     None
//
// Programmer           :     Greg Larson
//
// Created              :     11/23/93
//
// Last Revised         :     01/06/94
//
#include <dos.h>
//
/*****************************************************************************/
int step(void)                                  // test laser height sensors
{                                               // start of step.c
unsigned int far *llaser, *rlaser;              // left & right laser ports
int hght;                                       // height variable
float delta;                                    // delta height value
float mm;                                       // height in millimeters
float lstart, rstart;                           // left, right start values

printf("\x1B[2J");                              // clear screen
printf("Height Sensor Step Input Test\n\n\n");  // print message
printf("Put the step blocks under the left and right height sensors and ");
printf("press Return...");

llaser = MK_FP(0xE000, 0xE400);                 // point to left laser port
rlaser = MK_FP(0xE000, 0xE800);                 // point to right laser port

do {                                            // loop..
   delay(100);                                  // wait 100 ms
   hght = *llaser;                              // get left laser height
   lstart = (float)((hght&0x0FFF)-0x0800)*31.25E-3;        // convert to mm
   hght = *rlaser;                              // get right laser height
   rstart = (float)((hght&0x0FFF)-0x0800)*31.25E-3;        // convert to mm
   } while(!chkkey());                          // ..until a key is pressed
getkey();                                       // flush keyboard buffer

printf("\x1B[2J");                              // clear screen
printf("Height Sensor Step Input Test\n\n"); // print message
printf("   Starting Value (mm)   Current Value (mm)   Change (mm)");
printf("\n\nleft:\n\nright:");

do {                                            // loop until a key is pressed
   delay(100);                                  // wait 100 ms

   mov_curs(5,10);                              // move cursor to fifth line
   hght = *llaser;                              // get left laser height
```

```
  mm = (float)((hght&0x0FFF)-0x0800)*31.25E-3; // convert to mm
  delta = mm - lstart;                         // calculate delta value
  printf("%6.2f        %6.2f        %6.2f   ", lstart, mm, delta);

  mov_curs(7,10);                              // move cursor to seventh line
  hght = *rlaser;                              // get right laser height
  mm = (float)((hght&0x0FFF)-0x0800)*31.25E-3; // convert to mm
  delta = mm - rstart;                         // calculate delta value
  printf("%6.2f        %6.2f        %6.2f   ", rstart, mm, delta);

  } while(!chkkey());                          // wait for input from keybrd
getkey();                                      // flush keyboard buffer
return(0);                                     // normal return
}                                              // end of step.c
```

```
// height.c
// height sensor test routine
//
// Description          :     This program tests the laser height sensors
//
// Functions Included   :     height()
//
// External Functions   :     None
//
// Programmer           :     Greg Larson
//
// Created              :     06/17/92
//
// Last Revised         :     08/26/92
//
#include <dos.h>
//
/*****************************************************************************/
int height(void)                                    // test laser height sensors
{                                                   // start of height.c
unsigned int far *llaser, *rlaser;                  // left & right laser ports
int hght;                                           // height variable
unsigned int exp, numave;                           // exponent, number of samples
float meters, inches;                               // height in meters, inches

printf("\x1B[2J");                                  // clear screen
printf("Height Sensor Test\n\n");                   // print message
printf("     raw data     meters     inches     samples\n\n");
printf(" left:\n\nright:");

llaser = MK_FP(0xE000, 0xE400);                     // point to left laser port
rlaser = MK_FP(0xE000, 0xE800);                     // point to right laser port

do {                                                // main loop
  mov_curs(5,11);                                   // move cursor to fifth line
  delay(100);                                       // wait 100 ms

  hght = *llaser;                                   // get left laser height
  meters = (float)((hght&0x0FFF)-0x0800)*31.25E-6;        // convert to meters
  inches = (float)((hght&0x0FFF)-0x0800)*0.0012303;       // convert to inches
  exp = ((hght&0xF000)>>12) + 1;                    // get exponent
  numave = 1<<exp;                                  // number of samples in ave

  if(meters >= 0)
       printf("%4X      %5.4f      %4.3f      %3d   ",
                hght, meters, inches, numave);      // print sensor data
  else
       printf("%4X      %5.4f      %4.3f      %3d   ",
                hght, meters, inches, numave);      // print sensor data

  mov_curs(7,11);                                   // move cursor to seventh line
  hght = *rlaser;                                   // get right laser height
  meters = (float)((hght&0x0FFF)-0x0800)*31.25E-6;        // convert to meters
  inches = (float)((hght&0x0FFF)-0x0800)*0.0012303;       // convert to inches
```

```
    exp = ((hght&0xF000)>>12) + 1;                  // get exponent
    numave = 1<<exp;                                // number of samples in ave

    if(meters >= 0)
        printf("%4X      %5.4f      %4.3f      %3d   ",
                hght, meters, inches, numave);      // print sensor data
    else
        printf("%4X      %5.4f      %4.3f      %3d   ",
                hght, meters, inches, numave);      // print sensor data

    } while(!chkkey());                             // wait for input from keybrd

getkey();                                           // flush keyboard buffer
return(0);                                          // normal return
}                                                   // end of height.c
```

```
// adc.c
// test source code for the XVME-590/3 ADC circuit card
//
// Description          :     This program will test the ADC circuit card
//
// Functions Included   :     adc()
//
// External Functions   :     None
//
// Programmer           :     Greg Larson
//
// Created              :     02/07/92
//
// Last Revised         :     01/24/94
//
#include <stdio.h>
#include <conio.h>
#include <dos.h>
//
//
int adc(void)
{
char far *status;                          // ptr to adc status register
char far *channl;                          // ptr to adc channel register
int far *data;                             // ptr to adc data register

char i;                                    // counter index
int raw[8];                                // raw adc data
float voltage;                             // adc voltage

//
printf("\x1B[2J");                         // clear screen
printf("ADC Test\n\n");                    // print test header
printf("    signal        raw data      voltage\n\n");
printf("(0)  left accel\n");
printf("(1)  right accel\n");
printf("(2)  loop excit\n");
printf("(3)      x axis\n");
printf("(4)      y axis\n");
printf("(5) angular rate\n");
printf("(6)      ground\n");
printf("(7)      +5 VDC\n");

status = MK_FP(0xE000, 0xE081);            // make ptr to status reg
channl = MK_FP(0xE000, 0xE085);            // make ptr to channel reg
data = MK_FP(0xE000, 0xE086);              // make ptr to data reg

do {                                       // main loop
   delay(200);                             // wait 200 milliseconds
   *channl = 0x00;                         // point to channel 0
   *status = 0xA0;                         // start sequential conversion

   for(i = 0; i < 8; i++) {                // do for 8 channels
         while(*status < 0);               // wait for end of conversion
```

```
        raw[i] = *data;                                 // get raw adc data
        }                                               // end of for loop

    for(i = 0; i < 8; i++) {                            // do for 8 channels
        voltage = (float)(raw[i]-0x0800)*4.8828E-3;             // convert to volts
        mov_curs(i + 5, 23);                            // move cursor
        printf("%4X        %5.2f", raw[i], voltage);// print test data
        }                                               // end of for loop

    } while(!chkkey());                                 // wait for input from keybrd

getkey();                                               // clear keyboard buffer
return(0);                                              // normal return
}                                                       // end of adc.c
```

```
// gps.c
// routines for communicating with the Magellan GPS Receiver
//
// Description          :     Checks receiver status and collects GPS messages
//
// Functions            :     GPS_messages_off(), check_GPS(), GPS_status,
//                                  PMGLH(), read_message(), checksum()
//
// External Functions   :     None
//
// Programmer           :     Greg Larson
//
// Created              :     11/24/92
//
// Last Revised         :     03/10/93
//
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <conio.h>

void GPS_status();
void GPS_messages_off();
void PMGLH();
void checksum();

extern char gpsque[];
extern unsigned int gpshead, gpstail;
extern unsigned char gpsflag;

/****************************** CHECK_GPS FUNCTION ****************************/
/*
 * check_GPS()
 *
 * This function requests the GPS status message.  If a status message is
 * not received within 2 seconds, an error message is printed.
 *
 */
unsigned char check_GPS(void)
{

time_t start_time;                          // watchdog timer
char response[4];                           // yes/no response buffer
char rcvr_status_cmd[] = "$PMGLI,00,H00,1,A,00*4B\r\n";  // status msg

GPS_messages_off();                         // turn off all GPS messages
delay(200);                                 // pause for 0.2 seconds
serial_out(rcvr_status_cmd);                // request one status message
start_time = time(NULL);                    // initialize watchdog timer

//while(!gpsflag)                           // wait for complete message
//   if(difftime(time(NULL), start_time) > 1) {      // check for timeout
//        printf("\nThe GPS receiver is not responding!");
//        printf("\nDo you want to continue without it? (Y/N) >");
```

```
//          getmsg(response);                          // get yes/no response
//          if(tolower(response[0]) == 'y' || response[0] == '\0')
//             return(0);                              // continue without GPS
//          return(0);                                 // return error
//          }

gpshead = 0;                                           // reset head pointer
gpsflag = 0x00;                                        // reset gps message flag
return(0);                                             // say no error
}                                                      // end of check_GPS

/**************************** GPS_STATUS FUNCTION ****************************/
/*
 * GPS_status()
 *
 * This function continuously checks the status of the GPS receiver and
 * displays it on the screen
 *
 */

void GPS_status(void)
{                                                      // start of GPS_status

unsigned char msg_ptr;                                 // message pointer
char message[64];                                      // message buffer

if(gpsflag != 0) {                                     // check for message received
   gpsflag = 0x00;                                     // reset gps message flag
   for(msg_ptr = 0; gpsque[gpstail] != '\r'; gpstail++)
         message[msg_ptr++] = gpsque[gpstail];         // extract message from queue

   mov_curs(4, 1);                                     // move cursor to line 4
   PMGLH(message);                                     // interpret status message
   gpshead = 0;                                        // reset head pointer
   gpstail = 0;                                        // reset tail pointer
   }                                                   // end of if
}                                                      // end of GPS_status

/*************************** GPS_MESSAGES_OFF FUNCTION ************************/
/*
 * GPS_messages_off()
 *
 * This function individually switches off all GPS output messages.  An array
 * of strings is substituted individually into message off command string
 * positions 10, 11, and 12.  The checksum is determined and substituted into
 * message off command string positions 21 and 22.
 *
 */

void GPS_messages_off(void)
{                                                      // start of GPS_messages_off

char bin_prefix[][4] = {    "A00",                     // time and date
                            "B00",                     // position, altitude, etc.
```

```
                                "B01",          // position (last fix)
                                "B02",          // altitude
                                "C00",          // ECEF position
                                "D00",          // mode
                                "E00",          // ground course and velocity
                                "F00",          // sats used
                                "F01",          // sats used including status
                                "F02",          // 5 channel satellite usage
                                "G00",          // PDOP, GDOP, error estimate
                                "H00",          // receiver status
                                "R01",          // autopilot
                                "R02",          // bearing (dest, orig wypts)
                                "R03",          // bearing, distance to wypt
                                "R04",          // magnetic variation
                                "R05",          // waypoints
                                "R06",          // time to go to waypoint
                                "R07",          // ETA to waypoint
                                "R09",          // number of waypoints
                                "S01",          // datum, terrain, units setup
                                "T01",          // almanac data
                                "T02",          // ephemeris data
                                "U01",          // sat health (alm, eph, user)
                                "U03",   };     // sat status

char off_buffer[32];                            // message off command buffer
char command[] = "$PMGLI,00,";                  // base of off command
unsigned char checksum1, checksum2;             // checksum MS char, LS char
unsigned char msglength;                        // GPS message length
unsigned char i;                                // string position

for(i = 0; i < 26; i++)       {                 // switch off all 26 messages
   strcpy(off_buffer, command);                 // copy command into buffer
   strcat(off_buffer, *(bin_prefix+i));         // add prefix to buffer
   strcat(off_buffer, ",0,A,00*");              // add more stuff

   checksum(off_buffer, &checksum1, &checksum2, &msglength);
   off_buffer[21] = checksum1;                  // install checksum MS char
   off_buffer[22] = checksum2;                  // install checksum LS char

   strcat(off_buffer, "\r\n");                  // add CR and LF

   serial_out(off_buffer);                      // turn off specified message
   }                                            // end of for
}                                               // end of GPS_messages_off

/************************** PMGLH FUNCTION ************************************/
/*
 * PMGLH()
 *
 * Receiver status messages
 *
 */

void PMGLH(char gps_message[])
```

```
{

printf("*** GPS Receiver Status ***\n\n");

printf("oscillator: ");
switch(gps_message[20]) {                    // oscillator
   case '0':
         printf("ok        \n");
         break;
   case '1':
         printf("out of tune!\n");
         break;
   default:
         printf("default!   \n");
   }

printf("   almanac: ");
switch(gps_message[28]) {                    // almanac data
   case '0':
         printf("ok     \n");
         break;
   case '1':
         printf("none!  \n");
         break;
   case '2':
         printf("old!   \n");
         break;
   default:
         printf("default!\n");
   }

printf("    memory: ");
switch(gps_message[30]) {                    // memory
   case '0':
         printf("ok                    \n");
         break;
   case '1':
         printf("lost data, need to re-initialize!\n");
         break;
   default:
         printf("default!                \n");
   }

printf("     mode: ");
switch(gps_message[32]) {                    // OEM unit status
   case '0':
         printf("INI    \n");
         break;
   case '1':
         printf("IDL    \n");
         break;
   case '2':
         printf("STS    \n");
         break;
```

```
  case '3':
        printf("ALM    \n");
        break;
  case '4':
        printf("EPH    \n");
        break;
  case '5':
        printf("ACQ    \n");
        break;
  case '6':
        printf("POS    \n");
        break;
  case '7':
        printf("NAV    \n");
        break;
  default:
        printf("default!\n");
  }
}                                          // end of PMGLH

/****************************** READ_MESSAGE FUNCTION *************************/
/*
 * read_message()
 *
 * This function extracts GPS messages from the queue and writes them into
 * a file.
 *
 */

void read_message(FILE *fp_gps)
{

unsigned char checksum1 = 0, checksum2 = 0;      // checksum MS char, LS char
unsigned char msg_ptr = 0;                       // message pointer
unsigned char msglength;                         // message length
char message[255];                               // message buffer

gpsflag = 0x00;                                  // reset gps message flag

do {                                             // do...
   gpstail = gpstail & 0xFF;                     // reset gpstail at 0x100 (256)
   message[msg_ptr++] = gpsque[gpstail];         // extract char from queue
   } while(gpsque[gpstail++] != '\n');           // ...until end of message

checksum(message, &checksum1, &checksum2, &msglength);
message[++msglength] = '\n';                     // install line feed
msglength++;                                     // increment pointer
msglength = msglength & 0xFF;                    // limit mesage length

fwrite(message, msglength, 1, fp_gps);           // write GPS data
}                                                // end of read_message

/****************************** CHECKSUM FUNCTION ******************************/
/*
```

```
* checksum()
*
* Calculates the checksum of a string by bitwise XOR'ing the bytes
* between the '$' and the '*'.  The most significant and least
* significant digits of the checksum and the string length are
* returned to the main program.
*
*/

void checksum(char string[], unsigned char *chksum1, unsigned char *chksum2,
              unsigned char *length)
{

unsigned char index;
unsigned char chksum = 0;

for(index = 1; string[index] != '*'; index++) {
   chksum = chksum ^ string[index];          // calculate checksum
   if(index >= 255) {                        // check for message
         index = 1;                          // force index to 1
         break;                              // break out of for loop
         }
   }
*length = index;                             // message length
*chksum1 = (chksum>>4) + 0x30;               // 1st digit
if(*chksum1 > 0x39)                          // check for A-F
   *chksum1 = *chksum1 + 0x07;               // add A-F bias
*chksum2 = (chksum&0x0F) + 0x30;             // 2nd digit
if(*chksum2 > 0x39)                          // check for A-F
   *chksum2 = *chksum2 + 0x07;               // add A-F bias

}                                            // end of checksum
```

```
// rs232.c
// communicates with GPS receiver
//
// Description          :     Sends commands to the GPS receiver and
//                            reads data back from it
//
// Functions Included   :     serial_init(), serial_in(), serial_out()
//
// External Functions   :     None
//
// Programmer           :     Greg Larson
//
// Created              :     07/28/92
//
// Last Revised         :     11/04/92
//
#include <stdio.h>
#include <bios.h>
#include <dos.h>

// bioscom defines
#define COM1          0
#define COM2          1
#define SETUP         0
#define SEND          1
#define RECEIVE       2
#define STATUS        3
#define BAUD9600      0xE0 | 0x00 | 0x03 | 0x00  // 9600, N, 8, 1

// serial defines
#define COM1BASE      0x3F8                 // port base address (COM1)
#define COM2BASE      0x2F8                 // port base address (COM2)

#define TXR      0    /* Transmit register (WRITE) */
#define RXR      0    /* Receive register  (READ)  */
#define IER      1    /* Interrupt Enable          */
#define IIR      2    /* Interrupt ID              */
#define LCR      3    /* Line control              */
#define MCR      4    /* Modem control             */
#define LSR      5    /* Line Status               */
#define MSR      6    /* Modem Status              */
#define DLL      0    /* Divisor Latch Low         */
#define DLH      1    /* Divisor latch High        */

#define RCVRDY        0x01
#define OVRERR        0x02
#define PRTYERR       0x04
#define FRMERR        0x08
#define BRKERR        0x10
#define XMTRDY        0x20
#define XMTRSR        0x40
#define TIMEOUT       0x80

#define DTR           0x01
```

```
#define RTS             0x02
#define MC_INT          0x08

#define CTS             0x10
#define DSR             0x20

#define RX_INT          0x01

#define SPEED           19200
#define INITCOM1        0x02

// Function prototypes

void serial_init(void);
void init_COM1(void);
char serial_in(void);
void serial_out(char*);

int comport;
int portbase;

/*****************************************************************************/
void serial_init(void)
{
comport = COM2;
portbase = COM2BASE;
bioscom(SETUP, BAUD9600, comport);
}

/*****************************************************************************/
void init_COM1(void)
{
int divisor;                                    // baud rate variable
char old;                                       // old line control reg data

divisor = 115200L/SPEED;                        // calculate baud rate data
old = inportb(COM1BASE + LCR);                  // save old line control reg
outportb(COM1BASE + LCR, (old | 0x80));         // set DLAB bit
outportb(COM1BASE + DLL, (divisor & 0x00FF));   // install new baud rate
outportb(COM1BASE + DLH, ((divisor >> 8) & 0x00FF));
outportb(COM1BASE + LCR, old);                  // restore old line control reg

outportb(COM1BASE + LCR, INITCOM1);             // set # of data, stop, parity
}
/*****************************************************************************/
char serial_in(void)
{
int rxchar;                                     // received character

rxchar = bioscom(RECEIVE, 0, COM1);             // read serial port com1:
if(rxchar&0x8A00)                               // check for error
   return(0);                                   // say error
return((char)(rxchar&0x007F));                  // return received character
}
```

```
/*****************************************************************************/
void serial_out(char *ch)
{
outportb(portbase + MCR, DTR | RTS);                    // assert handshaking lines

while(*(ch)) {                                          // check for null terminator
   while(!(bioscom(STATUS, 0, comport)&0x2000));            // wait for TX empty
   outportb(portbase + TXR, *(ch++));                   // output next character
   }
}
```

```
// mov_curs.c
// cursor addressing routines
//
// Description              :    Uses ANSI.SYS to move the cursor
//
// Functions                :    mov_curs()
//
// External Functions       :    None
//
// Programmer               :    Greg Larson
//
// Created                  :    01/12/93
//
// Last Revised             :    01/12/93
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//
void mov_curs(int linenum, int column)
{                                          // start of mov_curs()
char buffer[10] = "\x1B[";                 // start of escape sequence
char value[8];                             // buffer for itoa()

strcat(buffer, itoa(linenum, value, 10));  // install line number in buffer
strcat(buffer, ";");                       // install ';' in buffer
strcat(buffer, itoa(column, value, 10));   // install column number
strcat(buffer, "H");                       // install 'H' in buffer
printf("%s", buffer);                      // send string to screen
}                                          // end of mov_curs()
```

```
// getmsg.c
// routines for communicating with the laptop computer
//
// Description              :     Gets characters from laptop and forms them into messages
//
// Functions                :     chkkey(), laptop_in(), getkey(), getkeye(), getmsg()
//
// External Functions       :     None
//
// Programmer               :     Greg Larson
//
// Created                  :     06/23/93
//
// Last Revised             :     03/15/94
//
#include <stdio.h>
#include <conio.h>
#include <bios.h>
#include <dos.h>
//
#define BS        0x0008                    // backspace char
#define DATA      0x03F8                    // COM1 data port
#define STATUS 0x03FD                       // COM1 status port

//*****************************************************************************
unsigned chkkey(void)                       //
{                                           // start of chkkey()
extern unsigned char laptop;                // laptop in use flag

if(!laptop)                                 // check for no laptop
  return(_bios_keybrd(_NKEYBRD_READY));     // return status of keyboard
else                                        // laptop is being used
  return((int)(inportb(STATUS) & 0x01));    // return RDA status
}                                           // end of chkkey()
//*****************************************************************************
int laptop_in(void)                         //
{                                           // start of laptop_in()
int rxchar;                                 // received char

while((inportb(STATUS) & 0x01) == 0);       // wait for RDA
rxchar = (int)inportb(DATA);                // read data
return(rxchar & 0x007F);                    // mask ASCII char and return
}                                           // end of laptop_in()
//*****************************************************************************
char getkey(void)                           //
{                                           // start of getkey()
extern unsigned char laptop;                // laptop in use flag

if(!laptop)                                 // check for no laptop
  return((char)getch());                    // read from keyboard
else{                                       // laptop is being used
  while((inportb(STATUS) & 0x01) == 0);     // wait for RDA
  return(inportb(DATA));                    // read from COM1
  }                                         // end of else
```

```
}                                          // end of getkey()
//*****************************************************************************
int getkeye(void)                          //
{                                          // start of getkeye()
extern unsigned char laptop;               // laptop in use flag
int ch;                                    // input char

if(!laptop) {                              // check for no laptop
   ch = getch();                           // read keyboard
   if(ch != BS)                            // check for not backspace
         putch(ch);                        // echo char to screen
   return(ch);                             // read from keyboard
   }                                       // end of if
else                                       // laptop is being used
   return(laptop_in());                    // read from COM1
}                                          // end of getkeye()
//*****************************************************************************
char *getmsg(char *string)                 //
{                                          // start of getmsg()
int ch;                                    // char input
char count = 0;                            // char count

while((ch = getkeye()) != '\r')      {     // wait for CR
   if(ch != BS) {                          // check for not a backspace
         count++;                          // increment char count
         *(string++) = (char)ch;           // install one char at a time
         }                                 // end of if
   else if(count > 0) {                    // check for buffer not empty
         count--;                          // decrement char count
         string--;                         // back up the pointer 1 space
         putchar(BS);                      // print a backspace
         putchar(' ');                     // print a space
         putchar(BS);                      // print a backspace
         }                                 // end of else if
   }                                       // end of while
*string = '\0';                            // install terminator
printf("\n");                              // print a line feed
return(string);                            // return pointer to string
}                                          // end of getmsg()
```

```
/*-----------------------------------------------------------------*
                    SERIAL.C

   The following code shows how to take advantage of some of
   the Turbo C extensions to the C language to do asynchronous
   communications without having to write supporting assembly-
   language routines.

   This program bypasses the less-than-adequate PC BIOS
   communications routines and installs a serial interrupt
   handler. Direct access to PC hardware allows the program to
   run at faster baud rates and eliminates the need for
   the main program to continuously poll the serial port for
   data; thus implementing background communications. Data that
   enters the serial port is stored in a circular buffer.

   * Compile this program with Test Stack Overflow OFF.

 *-----------------------------------------------------------------*/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <graphics.h>
#include "serial.h"

#define VERSION 0x0101

#define FALSE          0
#define TRUE           (!FALSE)

#define NOERROR        0     /* No error          */
#define BUFOVFL        1    /* Buffer overflowed    */

#define BS             0x08                    // backspace char
#define EXIT           0x18                    // Control-X, exit char
#define ARROW          0x00                    // first key of arrow sequence
#define ESC            0x1B   /* ASCII Escape character */
#define ASCII          0x007F /* Mask ASCII characters */
#define SBUFSIZ        0x4000 /* Serial buffer size    */

int        SError       = NOERROR;
int        portbase     = 0;
void        interrupt(*oldvects[2])();

static  char  ccbuf[SBUFSIZ];
unsigned int  startbuf      = 0;
unsigned int  endbuf        = 0;


/* Handle communications interrupts and put them in ccbuf */
void  interrupt com_int(void)
{
```

```
  disable();
  if ((inportb(portbase + IIR) & RX_MASK) == RX_ID)
  {
        if (((endbuf + 1) & SBUFSIZ - 1) == startbuf)
      SError = BUFOVFL;

        ccbuf[endbuf++] = inportb(portbase + RXR);
        endbuf &= SBUFSIZ - 1;
  }

  /* Signal end of hardware interrupt */
  outportb(ICR, EOI);
  enable();
}

/* Output a character to the serial port */
int   SerialOut(char x)
{
  long int        timeout = 0x0000FFFFL;

  outportb(portbase + MCR,  MC_INT | DTR | RTS);

  /* Wait for Clear To Send from modem */
  while ((inportb(portbase + MSR) & CTS) == 0)
    if (!(--timeout))
      return (-1);

  timeout = 0x0000FFFFL;

  /* Wait for transmitter to clear */
  while ((inportb(portbase + LSR) & XMTRDY) == 0)
    if (!(--timeout))
      return (-1);

  disable();
  outportb(portbase + TXR, x);
  enable();

  return (0);
}

/* Output a string to the serial port */
void   SerialString(char *string)
{
  while (*string)
    SerialOut(*string++);
}

/* This routine returns the current value in the buffer */
int   getccb(void)
{
  int             res;

  if (endbuf == startbuf)
```

```
    return (-1);

  res = (int) ccbuf[startbuf++];
  startbuf %= SBUFSIZ;
  return (res);
}

/* Install our functions to handle communications */
void  setvects(void)
{
  oldvects[0] = getvect(0x0B);
  oldvects[1] = getvect(0x0C);
  setvect(0x0B, com_int);
  setvect(0x0C, com_int);
}

/* Uninstall our vectors before exiting the program */
void  resvects(void)
{
  setvect(0x0B, oldvects[0]);
  setvect(0x0C, oldvects[1]);
}

/* Turn on communications interrupts */
void  i_enable(int pnum)
{
  int          c;

  disable();
  c = inportb(portbase + MCR) | MC_INT;
  outportb(portbase + MCR, c);
  outportb(portbase + IER, RX_INT);
  c = inportb(IMR) & (pnum == COM1 ? IRQ4 : IRQ3);
  outportb(IMR, c);
  enable();
}

/* Turn off communications interrupts */
void  i_disable(void)
{
  int          c;

  disable();
  c = inportb(IMR) | ~IRQ3 | ~IRQ4;
  outportb(IMR, c);
  outportb(portbase + IER, 0);
  c = inportb(portbase + MCR) & ~MC_INT;
  outportb(portbase + MCR, c);
  enable();
}

/* Tell modem that we're ready to go */
void  comm_on(void)
{
```

```
  int             c, pnum;

  pnum = (portbase == COM1BASE ? COM1 : COM2);
  i_enable(pnum);
  c = inportb(portbase + MCR) | DTR | RTS;
  outportb(portbase + MCR, c);
}

/* Go off-line */
void  comm_off(void)
{
  i_disable();
  outportb(portbase + MCR, 0);
}

void  initserial(void)
{
  endbuf = startbuf = 0;
  setvects();
  comm_on();
}

void  closeserial(void)
{
  comm_off();
  resvects();
}

/* Set the port number to use */
int   SetPort(int Port)
{
  int             Offset, far *RS232_Addr;

  switch (Port)
  { /* Sort out the base address */
   case COM1 : Offset = 0x0000;
           break;
   case COM2 : Offset = 0x0002;
                   break;
   default   : return (-1);
  }

  RS232_Addr = MK_FP(0x0040, Offset);  /* Find out where the port is. */
  if (*RS232_Addr == NULL) return (-1);/* If NULL then port not used. */
  portbase = *RS232_Addr;             /* Otherwise set portbase      */

  return (0);
}

/* This routine sets the speed; will accept funny baud rates. */
/* Setting the speed requires that the DLAB be set on.       */
int   SetSpeed(int Speed)
{
  char            c;
```

```
    int             divisor;

    if (Speed == 0)         /* Avoid divide by zero */
       return (-1);
    else
          divisor = (int) (115200L/Speed);

    if (portbase == 0)
       return (-1);

    disable();
    c = inportb(portbase + LCR);
    outportb(portbase + LCR, (c | 0x80)); /* Set DLAB */
    outportb(portbase + DLL, (divisor & 0x00FF));
    outportb(portbase + DLH, ((divisor >> 8) & 0x00FF));
    outportb(portbase + LCR, c);        /* Reset DLAB */
    enable();

    return (0);
}

/* Set other communications parameters */
int   SetOthers(int Parity, int Bits, int StopBit)
{
    int             setting;

    if (portbase == 0)                                  return (-1);
    if (Bits < 5 || Bits > 8)                     return (-1);
    if (StopBit != 1 && StopBit != 2)                   return (-1);
    if (Parity != NO_PARITY && Parity != ODD_PARITY && Parity != EVEN_PARITY)
                                                        return (-1);

    setting  = Bits-5;
    setting |= ((StopBit == 1) ? 0x00 : 0x04);
    setting |= Parity;

    disable();
    outportb(portbase + LCR, setting);
    enable();

    return (0);
}

/* Set up the port */
int   SetSerial(int Port, int Speed, int Parity, int Bits, int StopBit)
{
    if (SetPort(Port))              return (-1);
    if (SetSpeed(Speed))              return (-1);
    if (SetOthers(Parity, Bits, StopBit)) return (-1);

    return (0);
}

/*  Control-Break interrupt handler */
```

```
int   c_break(void)
{
   i_disable();
   fprintf(stderr, "\nStill online.\n");

   return(99);
}

main()
{
/* Communications parameters */
int      port    = COM1;
int      speed   = 19200;
int      parity  = NO_PARITY;
int      bits    = 7;
int      stopbits = 1;

int      done  = FALSE;
char       c;

int gdriver = VGA;                          // use VGA driver
int gmode = VGAHI;                          // use high resolution mode

int cee[18] = {120, 8, 312, 8, 299, 72, 203, 72, 183, 168, 279, 168,
                266, 232, 74, 232, 120, 8};
int tee[34] = {328, 8, 424, 8, 408, 88, 504, 88, 491, 152, 395, 152, 379, 232,
               395, 232, 408, 168, 488, 168, 462, 296, 274, 296, 299, 152,
               203, 152, 216, 88, 312, 88, 328, 8};

if(SetSerial(port, speed, parity, bits, stopbits) != 0) {
   fprintf(stderr, "Serial Port setup error.\n");
   return (99);
   }

initserial();                               // open serial port
ctrlbrk(c_break);                           // install Control-C handler
//
// print the Caltrans logo and software version number
//
initgraph(&gdriver, &gmode, "");            // initialize graphics mode
setbkcolor(BLACK);                          // black background
setcolor(7);                                // light gray foreground
fillpoly(9, cee);                           // draw a "C"
fillpoly(17, tee);                          // draw a "T"

gotoxy(18, 16);                             // move cursor to line 17
printf("CALTRANS");                         // print message

gotoxy(1, 20);                              // move cursor to line 20
printf("State of California\n");            // print header
printf("Department of Transportation\n");
printf("Division of New Technology, Materials & Research\n");
printf("Office of Electrical & Electronics Engineering\n\n");
printf("High Speed Pavement Profilometer Software\n");
```

```
printf("Version 00.01    03/03/94\n\n");
printf("Press Return to continue...");
getch();                                        // wait for CR to continue

printf("\x1B[=3h");                             // 80 X 25 color text mode
_setcursortype(_NOCURSOR);                      // turn off cursor
printf("Starting terminal mode, press Ctrl-X to exit...");
//
// read keys and display them; read the serial port and display the data
//
do {                                            // do...
  if(kbhit())                                   // check for key pressed
        switch (c = getch()) {                  // handle each char
          case EXIT  :                          // check for Control-X
                done = TRUE;                    // say exit
                break;                          // break
          case ARROW :                          // check for arrow, function key
                SerialOut(c);                   // send out char
                c = getch();                    // get next key, fall through
//              SerialOut(c);                   // send out char
//              break;                          // break
          case BS    :                          // check for backspace
                SerialOut(c);                   // send out backspace
                break;                          // break
          default    :                          // any other key
                SerialOut(c);                   // send out char
                putchar(c);                     // echo to local screen
          }                                     // end of switch
  if((c=getccb()) != -1)                        // check for char from COM1
        putchar(c);                             // send char to local screen
  } while(!done && !SError);                    // ...until done or error
//
// Exit and check for errors
//
switch(SError) {
  case NOERROR: fprintf(stderr, "\nbye.\n");
        closeserial();
        _setcursortype(_NORMALCURSOR);
        return (0);
  case BUFOVFL: fprintf(stderr, "\nBuffer Overflow.\n");
        closeserial();
        return (99);
  default:    fprintf(stderr, "\nUnknown Error, SError = %d\n", SError);
        closeserial();
        return (99);
  }                                             // end of switch
}                                               // end of serial()
```

```
// process.c
// data reduction and processing source code
//
// Description          :   This program reduces and processes raw sensor
//                              data into slope profile files, and processes
//                              slope profile data into elevation profile and
//                              IRI files.
//
// Functions Included   :       process(), prnline(), getraw(), getslp(),
//                              getnav(), getfile()
//
// External Functions   :       raw(), slope(), cnvnav()
//
// Programmer           :       Greg Larson
//
// Created              :       09/23/93
//
// Last Revised         :       01/13/94
//
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <dir.h>
#include <dos.h>
#include <string.h>

#define NORMAL      0x07
#define INVERSE     0x70

#define RAW_DATA    10
#define SLP_DATA    12
#define BAT_DATA    14
#define NAV_DATA    16
#define EXIT_DOS    18
#define PROMPT      20
//
void prnline();                     // print a line of the menu
int getraw();                       // get the raw data file name
int getslp();                       // get the slope file name
int getbat();                       // batch process raw data
int getnav();                       // get the navigation file name
int getfile();                      // get file name
/*****************************************************************************/
void main(void)
{                                   // start of data_pro()
extern int raw();                   // process raw sensor data
extern int slope();                 // process slope sensor data
extern int cnvnav();                // convert navigation data

unsigned char ch;                   // input char
unsigned char linenum;              // menu line number
```

```
int gdriver = VGA;                          // use VGA driver
int gmode = VGAHI;                          // use high resolution mode

int cee[18] = {120, 8, 312, 8, 299, 72, 203, 72, 183, 168, 279, 168,
               266, 232, 74, 232, 120, 8};
int tee[34] = {328, 8, 424, 8, 408, 88, 504, 88, 491, 152, 395, 152, 379, 232,
               395, 232, 408, 168, 488, 168, 462, 296, 274, 296, 299, 152,
               203, 152, 216, 88, 312, 88, 328, 8};
//
// print the Caltrans logo and software version number
//
clrscr();                                   // clear screen
initgraph(&gdriver, &gmode, "");            // initialize graphics mode
setbkcolor(BLACK);                          // black background
setcolor(7);                                // light gray foreground
fillpoly(9, cee);                           // draw a "C"
fillpoly(17, tee);                          // draw a "T"

gotoxy(18, 16);                             // move cursor to line 17
printf("CALTRANS");                         // print message

gotoxy(1, 20);                              // move cursor to line 20
printf("State of California\n");            // print header
printf("Department of Transportation\n");
printf("Division of New Technology, Materials & Research\n");
printf("Office of Electrical & Electronics Engineering\n\n");
printf("High Speed Pavement Profilometer Data Processing Software\n");
printf("Version 00.01    01/13/94\n\n");
printf("Press return to continue...");
getch();                                    // wait for CR to continue
printf("\x1B[=3h");                         // color text, 80 columns X 25
_setcursortype(_NOCURSOR);                  // turn off cursor
//
// print the data reduction and processing menu
//
while(1) {                                  // loop forever
   clrscr();                                // clear screen
   gotoxy(24,4);                            // move cursor to line 4
   printf("HIGH SPEED PAVEMENT PROFILOMETER"); // print menu
   gotoxy(23,8);
   printf("Data Reduction and Processing Menu");
   gotoxy(8,RAW_DATA);
//   printf("Process Raw Sensor Data");
   gotoxy(8,SLP_DATA);
//   printf("Process Slope Profile Data");
   gotoxy(8,BAT_DATA);
   printf("Batch Process Raw Sensor Data");
   gotoxy(8,NAV_DATA);
   printf("Convert Navigation Binary Data File to Text");
   gotoxy(8,EXIT_DOS);
   printf("Exit to DOS");
   gotoxy(4,PROMPT);
   printf("Move the highlight bar to the desired operation ");
   printf("and press Return...");
```

```
//
// highlight the first menu entry
//
// gotoxy(8,RAW_DATA);                          // move the cursor to line 10
   gotoxy(8,BAT_DATA);
   textattr(INVERSE);                           // set for inverse video
   cprintf("Process Raw Sensor Data                ");
   gotoxy(8,BAT_DATA);
// gotoxy(8,RAW_DATA);                          // move cursor to start
//
// highlight the menu entries as the up and down cursor arrow keys are
// pressed; when carriage return is entered, perform the highlighted function
//
// linenum = RAW_DATA;                          // initialize line number
   linenum = BAT_DATA;
   while((ch = getch()) != 0x0D)                // wait for a <CR>
         if(ch == 0) {                          // check for arrow key
            ch = getch();                       // get second character
            if((ch == 0x48) || (ch == 0x50)) {  // check for up or down arrow
                  textattr(NORMAL);             // set for normal video
                  prnline(linenum);             // re-print current line

                  if(ch == 0x48)                // check for up arrow
                     linenum = (linenum == BAT_DATA) ? EXIT_DOS : linenum - 2;
//                   linenum = (linenum == RAW_DATA) ? EXIT_DOS : linenum - 2;
                  else                          // check for down arrow
                     linenum = (linenum == EXIT_DOS) ? RAW_DATA : linenum + 2;

                  gotoxy(8,linenum);            // move cursor to new line
                  textattr(INVERSE);            // set for inverse video
                  prnline(linenum);             // re-print new line
                  gotoxy(8,linenum);            // move cursor to start
                  }                             // end of if
            }                                   // end of if(ch == 0)

   textattr(NORMAL);                            // set for normal video
   clrscr();                                    // clear screen
   switch(linenum) {                            // check operation selected
         case RAW_DATA :
            getraw();                           // process raw data selected
            break;
         case SLP_DATA :
            getslp();                           // process slope data selected
            break;
         case BAT_DATA :
            getbat();                           // batch process data selected
            break;
         case NAV_DATA :
            getnav();                           // convert nav data selected
            break;
         case EXIT_DOS :
            clrscr();                           // clear screen
            exit(0);                            // exit to DOS selected
         }                                      // end of switch
```

```
  }                                                    // end of while(1)
}                                                      // end of process()
/*****************************************************************************/
void prnline(int linenum)                    // start of prnline()
{
switch(linenum) {                                      // determine which line to print
  case RAW_DATA :                                      //
        cprintf("Process Raw Sensor Data                 ");
        break;                                         //
  case SLP_DATA :                                      //
        cprintf("Process Slope Profile Data              ");
        break;                                         //
  case BAT_DATA :                                      //
        cprintf("Batch Process Raw Sensor Data             ");
        break;                                         //
  case NAV_DATA :                                      //
        cprintf("Convert Navigation Binary Data File to Text");
        break;                                         //
  default     :                                        //
        cprintf("Exit to DOS                         ");
  }                                                    // end of switch
}                                                      // end of prnline()
/*****************************************************************************/
int getraw(void)
{                                                      // start of getraw()
int done;                                              // directory search done flag
struct ffblk ffblk;                                    // file finder data structure
char path[] = "RAW\\";                                 // raw data directory

printf("Available raw sensor data files:\n\n");        // print header
done = findfirst("RAW\\*.RAW", &ffblk, 0);             // search directory
if(done) {
  printf("   No raw sensor data files are available!");
  delay(3000);                                         // pause
  return(0);                                           // return to process menu
  }                                                    // end of if

if(!getfile(&done, &ffblk, &path))                     // get file name
  raw(ffblk.ff_name);                                  // process raw data file
return(0);                                             // exit
}                                                      // end of getraw()
/*****************************************************************************/
int getslp(void)
{                                                      // start of getslp()
int done;                                              // directory search done flag
struct ffblk ffblk;                                    // file finder data structure
char path[] = "SLOPE\\";                               // slope data directory

printf("Available slope profile data files:\n\n");
done = findfirst("SLOPE\\*.SLP", &ffblk, 0);           // search directory
if(done) {
  printf("   No slope profile data files are available!");
  delay(3000);                                         // pause
  return(0);                                           // return to process menu
```

```
   }                                          // end of if

if(!getfile(&done, &ffblk, &path))            // get file name
   slope(ffblk.ff_name);                      // process slope data file
return(0);                                    // exit
}                                             // end of getslp()
/*****************************************************************************/
int getnav(void)
{                                             // start of getnav()
int done;                                     // directory search done flag
struct ffblk ffblk;                           // file finder data structure
char path[] = "RAW\\";                        // raw data directory

printf("Available navigation data files:\n\n");
done = findfirst("RAW\\*.NAV", &ffblk, 0);    // search directory
if(done) {
   printf("   No navigation data files are available!");
   delay(3000);                               // pause
   return(0);                                 // return to process menu
   }                                          // end of if

if(!getfile(&done, &ffblk, &path))            // get file name
   cnvnav(ffblk.ff_name);                     // process navigation data file
return(0);                                    // exit
}                                             // end of cnvnav()
/*****************************************************************************/
int getfile(int *done, struct ffblk *ffblk, char *path)
{                                             // start of getfile()
unsigned char filecnt;                        // number of files found
unsigned char i, ch;                          // loop index, input char
unsigned char column, linenum;                // cursor column, line number
unsigned char processed;                      // file already processed flag
char file_name[84][13];                       // array of file names
char buffer[20];                              // scratch buffer
FILE *fileptr;                                // pointer to data file
//
// find each file name and pad it with spaces
//
for(filecnt = 0; !(*done) && filecnt < 84; filecnt++) {        // count .XXX files
   strcpy(file_name[filecnt], (*ffblk).ff_name);   // copy file name
   for(i = 0; file_name[filecnt][i] != '\0'; i++);  // find null terminator
   i--;                                       // compensate for loop count
   while(i++ < 11)                            // pad with spaces
         file_name[filecnt][i] = ' ';         // install a space
   file_name[filecnt][12] = '\0';             // install null terminator
   *done = findnext(ffblk);                   // search for files
   }
//
// print out all the file names
//
for(i = 0; i < filecnt; i++) {                // print out file names
   linenum = wherey();                        // get current line number
   if(linenum == 24) linenum = 3;             // reset to top of column
   switch(i/21) {                             // 21 file names per column
```

```
        case 1 :                                    // check for second column
          gotoxy(21,linenum);                       // move cursor to second column
          break;                                    // break
        case 2 :                                    // check for third column
          gotoxy(41,linenum);                       // move cursor to third column
          break;                                    // break
        case 3 :                                    // check for fourth column
          gotoxy(61,linenum);                       // move cursor to fourth column
        }                                           // end of switch
    strcpy(buffer, path);                           // install path in buffer
    strcat(buffer, file_name[i]);                   // add file name to path
    fileptr = fopen(buffer, "rb");                  // open file for reading
    fread(&processed, 1, 1, fileptr);               // get file processed flag
    fclose(fileptr);                                // close file
    if(processed)                                   // check for file processed
        printf("* %s\n", file_name[i]);             // print processed file name
    else                                            // un-processed file
        printf("  %s\n", file_name[i]);             // print un-processed file name
    }                                               // end of for
gotoxy(1,25);                                       // move cursor to last line
printf("Move the highlight bar to the desired file name and press return...");
//
// highlight the first file name
//
gotoxy(3,3);                                        // move cursor to line 3
textattr(INVERSE);                                  // set to inverse video
cprintf("%s", file_name[0]);                        // re-print the first file name
gotoxy(3,3);                                        // move cursor to line 3
linenum = 3;                                        // initialize line number
//
// highlight each file name as the cursor moves over it
//
i = 0;                                              // initialize file name index
while((ch = getch()) != 0x0D) {                     // wait for a <CR>
    if(ch == 0x1B) {                                // check for escape
        textattr(NORMAL);                           // set to normal video
        return(1);                                  // say escape pressed
        }                                           // end of if
    if(ch == 0) {                                   // check for arrow key
        ch = getch();                               // get second character
        if((ch == 0x48) || (ch == 0x50)) {          // check for up or down arrow
          textattr(NORMAL);                         // set to normal video
          cprintf("%s", file_name[i]);              // re-print current line

          if(ch == 0x48) {                          // check for up arrow
              if(linenum == 3) linenum = 23;        // top line, wrap around
              else linenum--;                       // otherwise, move up one line

              if(i == 0) {                          // check for first file
                i = filecnt - 1;                    // go to last file
                linenum = filecnt%21 + 2;           // calculate line number
                if(filecnt%21 == 0)                 // check for multiple of 21
                      linenum = 23;                 // move cursor to last line
                }                                   // end of if
```

```
            else i--;                          // not first file
            }                                  // end of if

        else {                                 // check for down arrow
            if(linenum == 23) linenum = 3;     // bottom line, wrap around
            else linenum++;                    // otherwise, move down one line

            if(i == filecnt - 1) {             // check for last file
              i = 0;                           // go to first file
              linenum = 3;                     // move cursor to top line
              }                                // end of if
            else i++;                          // not last file
            }                                  // end of if

        switch(i/21) {                         // 21 file names per column
            case 0 :                           // check for first column
              column = 3;                      // cursor position
              break;                           // break
            case 1 :                           // check for second column
              column = 23;                     // cursor position
              break;                           // break
            case 2 :                           // check for third column
              column = 43;                     // cursor position
              break;                           // break
            case 3 :                           // check for fourth column
              column = 63;                     // cursor position
            }                                  // end of switch

        gotoxy(column,linenum);                // move cursor
        textattr(INVERSE);                     // set to inverse video
        cprintf("%s", file_name[i]);           // re-print file name
        gotoxy(column,linenum);                // move cursor
        }                                      // end of if
      }                                        // end of if(ch == 0)
  }                                            // end of while
textattr(NORMAL);                              // set to normal video
clrscr();                                      // clear screen
strcpy((*ffblk).ff_name, file_name[i]);        // copy the selected file name
return(0);                                     // return file name
}                                              // end of getfile()
/*****************************************************************************/
int getbat(void)
{                                              // start of getbat()
int done;                                      // directory search done flag
struct ffblk ffblk;                            // file finder data structure
unsigned char i, j, ch;                        // loop indices, input char
unsigned char column, linenum;                 // cursor column, line number
unsigned char processed;                       // file already processed flag
char file_name[84][13];                        // array of file names
char tag_name[84][13];                         // array of tagged file names
unsigned char filecnt;                         // number of files found
unsigned char tagcnt = 0;                      // number of tagged files
char name[10], buffer[20];                     // file name, scratch buffers
FILE *fp_report;                               // pointer to report file
```

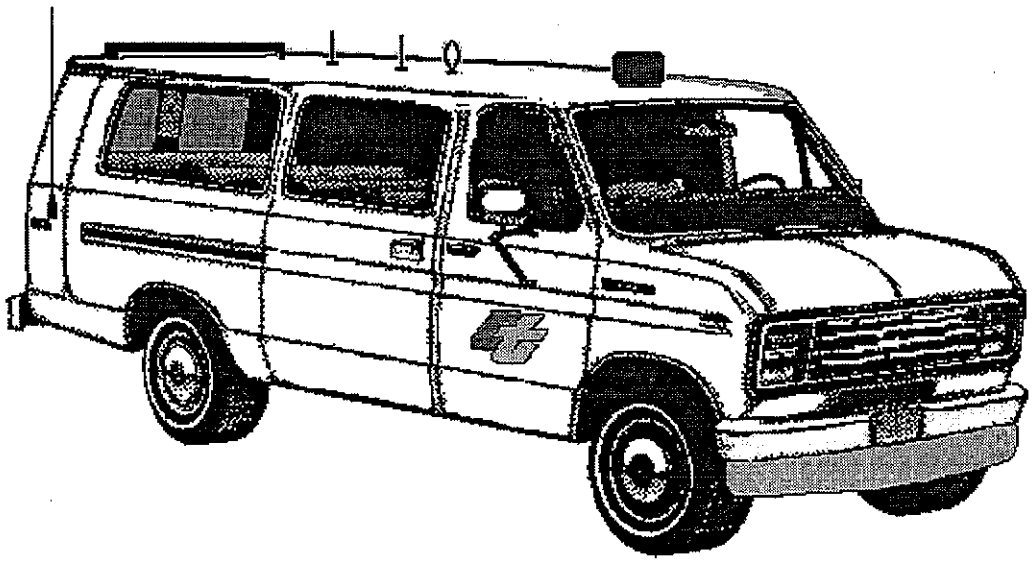
```
FILE *fp_raw;                                   // pointer to raw data file
//
// check for any files available
//
printf("Available raw sensor data files:\n\n");  // print header
done = findfirst("RAW\\*.RAW", &ffblk, 0);      // search directory
if(done) {
   printf("   No raw sensor data files are available!");
   delay(3000);                                 // pause
   return(0);                                   // return to process menu
   }                                            // end of if
//
// find each file name and pad it with spaces
//
for(filecnt = 0; !done && filecnt < 84; filecnt++) {  // count .RAW files
   strcpy(file_name[filecnt], ffblk.ff_name);   // copy file name
   for(i = 0; file_name[filecnt][i] != '\0'; i++);  // find null terminator
   i--;                                         // compensate for loop count
   while(i++ < 11)                              // pad with spaces
         file_name[filecnt][i] = ' ';           // install a space
   file_name[filecnt][12] = '\0';               // install null terminator
   done = findnext(&ffblk);                     // search for files
   }                                            // end of for
//
// print out all the file names
//
for(i = 0; i < filecnt; i++) {                  // print out file names
   linenum = wherey();                          // get current line number
   if(linenum == 24) linenum = 3;               // reset to top of column
   switch(i/21) {                               // 21 file names per column
         case 1 :                               // check for second column
            gotoxy(21,linenum);                 // move cursor to second column
            break;                              // break
         case 2 :                               // check for third column
            gotoxy(41,linenum);                 // move cursor to third column
            break;                              // break
         case 3 :                               // check for fourth column
            gotoxy(61,linenum);                 // move cursor to fourth column
         }                                      // end of switch
   strcpy(buffer, "RAW\\");                     // install path in buffer
   strcat(buffer, file_name[i]);                // add file name to path
   fp_raw = fopen(buffer, "rb");                // open file for reading
   fread(&processed, 1, 1, fp_raw);             // get file processed flag
   fclose(fp_raw);                              // close file
   if(processed)                                // check for file processed
         printf("* %s\n", file_name[i]);        // print processed file name
   else                                         // un-processed file
         printf("  %s\n", file_name[i]);        // print un-processed file name
   }                                            // end of for
gotoxy(1,25);                                   // move cursor to last line
printf("Press return to tag a file; Press escape to start processing...");
//
// highlight the first file name
//
```

```
gotoxy(3,3);                                        // move cursor to line 3
printf("%s<===", file_name[0]);                     // re-print the first file name
gotoxy(3,3);                                        // move cursor to line 3
linenum = 3;                                        // initialize line number
//
// highlight the tagged files and wait for an Esc to begin processing
//
i = 0;                                              // initialize file name index
while((ch = getch()) != 0x1B) {                     // wait for an <Esc>
  if(ch == 0x0D) {                                  // check for <CR>
        textattr(INVERSE);                          // set to inverse video
        cprintf("%s", file_name[i]);                // tag file name
        printf("    ");                             // cover up pointer
        strcpy(tag_name[tagcnt++], file_name[i]);   // save tagged file name

        if(linenum == 23) linenum = 3;              // bottom line, wrap around
        else linenum++;                             // otherwise, move down one line

        if(i == filecnt - 1) {                      // check for last file
          i = 0;                                    // go to first file
          linenum = 3;                              // move cursor to top line
          }                                         // end of if
        else i++;                                   // not last file

        switch(i/21) {                              // 21 file names per column
          case 0 :                                  // check for first column
                column = 3;                         // cursor position
                break;                              // break
          case 1 :                                  // check for second column
                column = 23;                        // cursor position
                break;                              // break
          case 2 :                                  // check for third column
                column = 43;                        // cursor position
                break;                              // break
          case 3 :                                  // check for fourth column
                column = 63;                        // cursor position
          }                                         // end of switch

        gotoxy(column,linenum);                     // move cursor
        textattr(NORMAL);                           // set to normal video
        printf("%s<===", file_name[i]);             // re-print file name
        gotoxy(column,linenum);                     // move cursor
        }                                           // end of if
  if(ch == 0) {                                     // check for arrow key
        ch = getch();                               // get second character
        if((ch == 0x48) || (ch == 0x50)) {          // check for up or down arrow
          printf("%s    ", file_name[i]);           // re-print current line

          if(ch == 0x48) {                          // check for up arrow
                if(linenum == 3) linenum = 23;      // top line, wrap around
                else linenum--;                     // otherwise, move up one line

                if(i == 0) {                        // check for first file
                  i = filecnt - 1;                  // go to last file
```

```
                    linenum = filecnt%21 + 2;        // calculate line number
                    }                                // end of if
                  else i--;                          // not first file
                  }                                  // end of if

            else {                                   // check for down arrow
                  if(linenum == 23) linenum = 3;     // bottom line, wrap around
                  else linenum++;                    // otherwise, move down one line

                  if(i == filecnt - 1) {             // check for last file
                    i = 0;                           // go to first file
                    linenum = 3;                     // move cursor to top line
                    }                                // end of if
                  else i++;                          // not last file
                  }                                  // end of if

            switch(i/21) {                           // 21 file names per column
                  case 0 :                           // check for first column
                    column = 3;                      // cursor position
                    break;                           // break
                  case 1 :                           // check for second column
                    column = 23;                     // cursor position
                    break;                           // break
                  case 2 :                           // check for third column
                    column = 43;                     // cursor position
                    break;                           // break
                  case 3 :                           // check for fourth column
                    column = 63;                     // cursor position
                  }                                  // end of switch

            gotoxy(column,linenum);                  // move cursor
            printf("%s<====", file_name[i]);         // re-print file name
            gotoxy(column,linenum);                  // move cursor
            }                                        // end of if
          }                                          // end of if(ch == 0)
    }                                                // end of while
fp_report = fopen("DATA\\REPORT.TXT", "wt");         // overwrite old report file
fclose(fp_report);                                   // close report file
//
// process the selected raw data files
//
for(i = 0; i < tagcnt; i++) {                        // process each tagged file
    for(j = 0; tag_name[i][j] != '.'; j++)           // mask off extension
          name[j] = tag_name[i][j];                  // install one char at a time
    name[j] = '\0';                                  // install terminator

    strcpy(buffer, name);                            // install file name in buffer
    strcat(buffer, ".RAW");                          // add extension to file name
    if(raw(buffer))                                  // process raw data file
          continue;                                  // error in processing

    strcpy(buffer, name);                            // install file name in buffer

    strcat(buffer, "L.SLP");                         // add extension to file name
```

Appendix C

CALTRANS HIGH-SPEED PAVEMENT PROFILER VAN



California Department of Transportation

DATA ACQUISITION and GENERAL OPERATIONS

*** * * CAUTION * * ***

DO NOT LOOK DIRECTLY AT THE LASER BEAM LENSES
THE LASER BEAMS MAY CAUSE PERMANENT EYE INJURY
WHEN SERVICING THE VEHICLE MAKE SURE LASERS ARE TURNED OFF

© Manual compiled 1994, Charles P. Hoffman, California Department of Transportation

PROFILE VAN COMPUTER START-UP PROCEDURES

1. Start vehicle or connect external 110 volt extension cord.

2. Turn on power switch at rear of CPU housing. (If connected to 110 volt external power source, turn on power strip located on the floor at the rear of the CPU housing.) The red LED light(s) on the front of the CPU housing should now be on.

3. Plug the 12 volt adapter cable into the dashboard cigarette lighter outlet. Turn on the VGA monitor.

4. Turn on the switch on the front of the CPU housing, this starts the computer boot-up sequence. The computer will boot-up to the VGA monitor.

5. If profiling data is going to be collected or systems tests performed, turn on both laser keys inside the front cover of the CPU housing. Make sure both lasers are within the operating ranges indicated by the green LED lights. If the yellow LED lights remain on, it is possible that the front bumper is over a curb or other obstruction, back the van up so that the lasers can contact ground level.

NOTE: If the lasers are not going to be used, such as when downloading data, leave them off as they use a lot of power and will create an unnecessary load on the auxiliary battery.

6. The VGA monitor should now show that the Profiler computer is running and in the C:\PRO directory.

7. Once the Profile computer is on-line, the operator may conduct all further operations from the driver's seat.

8. Insert the laptop computer into the mounting bracket and attach the restraining strap. Plug in the power adapter cable, serial cable, and expanded keyboard cables. Make sure the external power symbol is showing on the laptop LCD display, it is sometimes necessary to twist the power adapter plug in the cigarette lighter socket to make a good connection. Also make sure that the cables from the extended keyboard are plugged into the proper receptacles. (They both have the same plug design.)

9. Turn on the laptop computer, the computer will automatically select "DOS Prompt" from the boot-up options. You may select to start in "Windows without PCM" (Phoenix Card Management, modem operations.), but it is suggested that the data collection programs be run from the DOS prompt to avoid hardware interrupt problems.

10. Start the "Serial" program on the laptop computer from the Windows icon or by typing "S" at the DOS prompt in the C:\PRO directory. When the Caltrans screen appears, press the return key as instructed. **Important:** If you are going to use the laptop computer to

control the Profile computer, the "Serial" program on the laptop computer must be started first.

11. Using the keyboard for the Profile van computer, which should be in the C:\PRO directory, type "P" to start the Profile program. The laptop computer screen should now display the Profile program options.

PROFILE.EXE PROGRAM - MAIN MENU

DATA ACQUISITION

NOTE: The van should be operated long enough for the tires to reach normal operating temperatures before attempting data acquisition.

1. Select "Data Acquisition" from the Profile main menu and fill in the requested information.

2. The file name must not be more than 7 characters long, the reason for this being that as the "raw" data is processed, the Process program appends an 8th character to the file name to distinguish left and right wheelpaths. i.e., file name SAC1001.RAW becomes several files named SAC1001L.??? and SAC1001R.??? in several directories. After processing, the files stored in the C:\PRO\IRI directory will be used by headquarters to determine the "IRI" (International Roughness Index) scores for the Pavement Condition Survey.

3. Enter the beginning and ending Post Mile numbers and the beginning and ending County Odometer numbers from the highway log. The Post Mile and County Odometer numbers will not always match. The Post Mile numbers will be used by headquarters to relate the collected data to their database files. The County Odometer numbers will assist you in determining your location while collecting data. At this time, the Profile program does not have the ability to incorporate equations in determining mileage, thus it is necessary to use the County Odometer numbers to determine the actual distance you will be traveling.

4. Proceed to the point where you wish to begin collecting data and press the "B" key to begin data collection. Try to maintain a speed above 20 miles per hour (below 20 mph the data is not usable). When you reach the end of the selected data acquisition section, press the "ESC" (Escape) key to end data collection. The computer will continue to collect data until the last 1/10 mile "buffer" is filled and will then return you to the Profile main menu screen.

5. During data collection, pressing the various "F" keys will add comment lines to the data file. In addition, it is suggested that a cassette recorder be used to supplement the use of the "F" keys, and to record any other information pertaining to data collection. After the raw files have been processed, the IRI files may be edited to include any comments you may wish to add. At times it will not be possible to press the "F" keys at the exact point where a comment applies, such as at a bridge departure where the point is also the same

point at which the pavement type changes. It is not possible to press two "F" keys at the same exact point, but by using the cassette recorder you could make a note of the location and event and add the comment to the IRI files after they have been processed.

6. The raw data is stored in the C:\PRO\RAW directory on the Profile van computer.

7. After collecting the data, the files in the above directory should be downloaded to the corresponding directory on the laptop computer by using the Laplink program.

NOTE: If the files are comparatively small, they may be processed on the Profile van computer. The time it takes to transfer small files to the laptop and then process them on the laptop is about the same as if the processing is done on the Profile van computer. Then, at the end of the day you can just transfer the processed IRI files to the laptop. (See directions for processing files later in this manual.) Also, if you are finished with an area or route and are relocating to a site that will take you several miles from the area, it is a good idea to process the data before leaving your current location. That way, if a problem does arise with the processed IRI files, you wouldn't have to waste time and mileage backtracking to an area you thought you were finished with.

8. Exit the Profile program by selecting "Exit to DOS" from the main menu. Then, hold down the "CTRL" key and press "X" to exit the Serial program on the laptop computer. This puts both computers back under the control of their own keyboards.

9. Start LAPLINK 5 on both computers, either from the DOS prompt in the C:\LL5 directory or from the LL5 icon in Windows. Transfer the files from the C:\PRO\RAW directory on the Profile van computer to the C:\PRO\RAW directory on the laptop computer.

DATA COLLECTION-QUICK START

1. Select "Data Acquisition" from main menu.
2. Enter file name and other information.
3. Proceed to starting point and press "B" to begin data collection.
4. Press "F" keys as necessary to enter comments.
5. Upon reaching ending point, press "Escape" key to end data collection.

DISPLAY VAN SPEED AND GPS DATA

This routine is used to display the distance between two points and the speed of the van.

1. To view the elapsed miles and speed of the van, move the cursor to "Display Van Speed and GPS Data" on the main menu. When you reach the point where you want to start measuring distance and/or speed, press the "ENTER" key and the program will display the desired information. To quit the program press the "ESCAPE" key and you will be brought back to the main menu.

SYSTEM TESTS

BOUNCE TEST

The Profiler computer has been fitted with hardware and software that remove the up and down bouncing of the van itself from the data collected. This gives a true profile of the actual pavement surface. The "Bounce" test allows the operator to determine that the hardware and software are functioning properly.

1. Make sure the vehicle is in park and the parking brake is set to prevent as much backward or forward movement as possible. Place the flat steel plates used in the "Step" test under each laser sensor to insure that the sensors are looking at a smooth surface. Start the test by selecting "System Test" from the main menu and "Bounce Test" from the test menu. Press "B" to begin the test, exit the vehicle and push up and down on the font bumper of the van for about 30 seconds. Then, get back into the van and press the "ESCAPE" key to terminate the test. The computer will save the test data in a file called "BOUNCE" in the C:\PRO\RAW directory.

2. The data collected from the "Bounce" test can be processed in the same way as other raw files. After processing, there will be 2 files named "BOUNCEL", and "BOUNCER". The files can be viewed with the Windows Notepad or on any text editor. The IRI shown should be very small (< 50), this shows that the computer is functioning properly.

PULSED WHEEL SENSOR TEST (ODOMETER CALIBRATION)

NOTE: The odometer should be recalibrated anytime there is a change in conditions, such as a flat tire being changed, after adding air to the tires, or drastic climatic changes. It should be checked at the beginning of every work week and when it becomes obvious during data collection that the odometer readings do not coincide with actual postmile numbers.

1. The odometer calibration number is kept in a file named "ODOMETER.TXT" in the C:\PRO directory. The number can be viewed and/or changed by opening the file in Windows "NOTEPAD" or the DOS "Editor".

2. A test area **MUST** be less than 3 miles long, but try to get as close to the 3 mile limitation as possible. The longer the test area, the higher the accuracy of the odometer. Do not use postmile markers as starting or ending points, they are seldom placed at exact postmile locations. Use permanent structures, such as bridges, for postmile starting and ending points. Determine the exact distance between the 2 locations by figuring the difference in postmile numbers to 3 decimals (0.000).

3. Start the program by selecting it from the main menu and then enter the information requested. Enter 0.000 as the beginning postmile and the number you obtained from the above calculation as the ending postmile. Drive to the exact starting point and stop, press the "B" key to begin the test and drive to the exact ending point and stop. Press the

"ESCAPE" key to end the test. The screen will now show the number arrived at by the test program. Take that number and divide it by the same number you used as the ending postmile number, this is the number you will use in the "ODOMETER.TXT file.

IMPORTANT: Make at least 3 passes through test area and compare the results before changing the file.

EXAMPLE:

FROM P.M. 5.555 TO P.M. 7.999		7.999 - 5.555	= 2.444
BEGINNING POSTMILE	= 0.000	ENDING POSTMILE	= 2.444
TEST RESULT NUMBER	= 159593	159593 / 2.444	= 65300
ODOMETER.TXT FILE	= 65300		

4. The number can be changed at any time but will affect the odometer calibration dramatically, so care should be taken to keep track of the numbers used and the dates they were changed.

ODOMETER CALIBRATION-QUICK START

1. Select a test location below, but as close to, 3 miles long, between 2 permanent structures.
2. Figure the distance between the two locations.
3. Select "Pulsed Wheel Sensor Test" from the test menu.
4. Enter 0.000 as the starting postmile and the length of the test as the ending postmile.
5. Proceed to starting point, stop, press "B" to begin test and proceed.
6. Stop at ending point and press "Escape" to end test.
7. Calculate odometer number and change the ODOMETER.TXT file if necessary.

STEP INPUT TEST (HEIGHT SENSOR BLOCKS)

Used to determine whether the lasers are functioning properly. Measures the vertical distance between the "steps" on the test blocks. Make sure vehicle is in park and parking brake set before starting test.

1. Place the "STEP" blocks under the lasers, using the optical viewer to align the beams on the steps. Start the program by selecting "System Test" from the main menu and then selecting "Step Input Test" from the test menu. Make a note of the numbers shown on the screen for each wheelpath.

2. Move the blocks so that the lasers are on the next step and make a note of the numbers on the screen. The differences between the numbers for each wheelpath should be the same as the number imprinted on the side of the "STEP" blocks. By moving the blocks so that the beams are moved from step to step, the differences in height can be noted to confirm that the lasers are operating within their proper ranges. Press the "ESCAPE" key to end the test.

HEIGHT SENSOR TEST (GROUND LEVEL)

Similar to the "STEP" test, except that it is used mainly to level the van so that both lasers are at approximately the same distance from the ground.

1. Select a smooth, level area to conduct the test, preferably a concrete pad or driveway. Make sure the van is in park with the parking brake set to prevent backward and forward motion. Start the program by selecting "System Test" from the main menu and then selecting "Height Sensor Test" from the test menu.

2. The test should be performed with the driver in position in the driver's seat and the van configured for actual data collection. When testing the height sensors, it is not necessary that both readings be exactly the same. The sensors have a range of 5 inches, 2.5 inches above and 2.5 inches below ground level. Ideally, readings of 0.000 for each sensor would show that the sensors are exactly on target. If the numbers on the centimeter range are within the 0.050 area it's as close to perfect as possible. If the numbers for the two sensors differ greatly, the difference can be corrected by adjusting the front and/or rear air shocks and airbags.

ANALOG TO DIGITAL CONVERTER TEST

Checks the system for proper operation and voltage requirements. If one of the systems is not operating or receiving proper voltage, it could interfere with proper data collection. If these procedures do not solve the problem, contact Greg Larson at the Lab. (916) 227-7097

1. Start the program by selecting "System Test" from the main menu and then selecting "Analog to Digital Converter Test" from the test menu.

2. The test screen will show whether all systems are operating and the voltages available. If a line on the screen shows nothing, or is staying at a constant number, there is something malfunctioning.

3. If the test shows that there is a malfunction, check for loose connections in the rear of the computer.

PROCESS.EXE PROGRAM

1. After downloading the "RAW" data files as outlined earlier in the manual, they are to be processed on the laptop computer by using the "PROCESS" program. Start the program from the Icon in Windows, or by typing "PROCESS" at the DOS prompt in the C:\PRO directory.

2. Select "Batch Process Raw Sensor Data" from the menu. The files may be selected one at a time or in batches by pressing the "ENTER" key at each file selected. After selecting the files, press the "ESCAPE" key to begin the automatic processing.

3. The files created by the processing will be saved in the directory C:\PRO\IRI as files with the same names as the raw files but with the "L" and "R" suffixes appended to them to denote the left and right wheelpaths, and with the extension "IRI".

4. The "IRI" files should be copied to a floppy disk and the information forwarded to headquarters. A copy of the files should also be kept by the van operator as a backup copy.

5. A database and spreadsheet in 'Microsoft Works for Windows' is kept, showing the information about the routes for which data has been collected. Copies of these files are updated by the operator daily. At the end of the workweek they are also updated to the computer at headquarters either by modem or by copying from a floppy disk to the directory C:\JOHN.

FILE MANAGEMENT

Because of the number and size of the files created by the data collection and processing procedures, it is necessary that the processed files be deleted from the hard disks of both the laptop and van computers on a regular basis.

1. Once the raw files have been processed and the IRI files saved on the hard disk and backed up on floppy disks, the subdirectories of C:\PRO may be "cleaned up". The files that should be deleted are located in the following subdirectories:

C:\PRO\RAW	ALL FILES MAY BE DELETED
" \DATA	ALL FILES MAY BE DELETED
" \SLOPE	ALL FILES MAY BE DELETED
" \ELEV	ALL FILES MAY BE DELETED
" \IRI	ALL FILES MAY BE DELETED (After backing up to disk.)

Appendix D

Data Record Format

Setup Data File (TEST.SET)

```
 4/07/94
File Name: TEST
 Operator: Chuck Hoffman
 District: 3
   County: Sacramento
    Route: US Highway 50
     Lane: 2
Direction: eastbound
 Pavement: rigid
 Begin PM: 4.000
Begin ODM: 0.000
   End PM: 19.000
  End ODM: 15.000
 Comments: partly cloudy, 74 degrees
```

Keyboard Data File (TEST.KEY)

```
8:19:08.28
0:01:23.654     1.293  Rigid to Flex
0:01:33.756     1.450  Bridge Approach
0:03:21.860     3.121  Bridge Departure
0:03:26.356     3.193  Loop
0:03:31.756     3.275  Flex to Rigid
0:03:37.866     3.369  Bridge Approach
0:03:39.565     3.396  Bridge Departure
0:04:03.177     3.761  Rigid to Flex
0:04:42.916     4.368  Bridge Approach
0:04:45.110     4.402  Bridge Departure
0:04:46.458     4.423  Flex to Rigid
0:05:13.491     4.835  Bridge Approach
0:05:16.113     4.875  Bridge Departure
0:05:51.431     5.420  Bridge Approach
0:05:53.576     5.453  Bridge Departure
0:05:57.713     5.516  Bridge Approach
0:06:47.670     6.284  Bridge Departure
0:08:40.871     8.049  Flex to Rigid
0:08:58.150     8.315  Rigid to Flex
0:09:07.683     8.475  Loop
0:09:18.052     8.609  Flex to Rigid
0:09:29.659     8.798  Rigid to Flex
0:10:30.747     9.646  Speed below 20mph
0:11:00.150     9.723  Speed above 20mph
0:11:12.931     9.838  Speed below 20mph
0:11:58.226     9.879  Speed above 20mph
0:12:08.277     9.969  Escape
8:31:21.08
```

format:

```
0:01:23.654     1.293  Rigid to Flex
```

0:01:23.654 - elapsed time in hours, minutes, seconds, and milliseconds

1.293 - elapsed distance traveled in miles

Rigid to Flex - message from the keyboard (initiated by pressing a function key)

Partial GPS Data File (TEST.GPS)

```
 8:19:08.28   4/07/94
$GPGGA,151945,3833.29,N,12124.13,W,1,4,003,124,M,-026,M*6C
$GPGGA,151946,3833.30,N,12124.12,W,1,4,003,120,M,-026,M*62
$GPGGA,151947,3833.30,N,12124.10,W,1,4,003,117,M,-026,M*65
$GPGGA,151948,3833.31,N,12124.08,W,1,4,003,116,M,-026,M*63
$GPGGA,151949,3833.31,N,12124.07,W,1,4,003,114,M,-026,M*6F
$GPGGA,151950,3833.31,N,12124.05,W,1,4,003,115,M,-026,M*64
$GPGGA,151950,3833.31,N,12124.05,W,1,4,003,115,M,-026,M*64
$GPGGA,151952,3833.32,N,12124.02,W,1,4,003,109,M,-026,M*6F
$GPGGA,151953,3833.33,N,12124.00,W,1,4,003,107,M,-026,M*63
$GPGGA,151954,3833.33,N,12123.98,W,1,4,003,103,M,-026,M*66
$GPGGA,151955,3833.33,N,12123.97,W,1,4,003,100,M,-026,M*6B
$GPGGA,151956,3833.34,N,12123.95,W,1,4,003,097,M,-026,M*62
$GPGGA,151957,3833.34,N,12123.94,W,1,4,003,096,M,-026,M*63
$GPGGA,151958,3833.35,N,12123.92,W,1,4,003,094,M,-026,M*69
$GPGGA,151959,3833.35,N,12123.90,W,1,4,003,091,M,-026,M*6F
$GPGGA,151959,3833.35,N,12123.90,W,1,4,003,091,M,-026,M*6F
$GPGGA,152002,3833.36,N,12123.85,W,1,4,003,083,M,-026,M*6F
$GPGGA,152002,3833.36,N,12123.85,W,1,4,003,083,M,-026,M*6F
$GPGGA,152004,3833.37,N,12123.82,W,1,4,003,079,M,-026,M*6A
$GPGGA,152005,3833.38,N,12123.81,W,1,4,003,076,M,-026,M*68
$GPGGA,152006,3833.38,N,12123.79,W,1,4,003,072,M,-026,M*68
$GPGGA,152007,3833.38,N,12123.77,W,1,4,003,070,M,-026,M*65
$GPGGA,152008,3833.39,N,12123.75,W,1,4,003,070,M,-026,M*69
$GPGGA,152009,3833.39,N,12123.74,W,1,4,003,068,M,-026,M*60
$GPGGA,152010,3833.40,N,12123.72,W,1,4,003,065,M,-026,M*6D
$GPGGA,152011,3833.40,N,12123.70,W,1,4,003,063,M,-026,M*68
$GPGGA,152011,3833.40,N,12123.70,W,1,4,003,063,M,-026,M*68
$GPGGA,152013,3833.41,N,12123.67,W,1,4,003,059,M,-026,M*64
$GPGGA,152014,3833.41,N,12123.65,W,1,4,003,057,M,-026,M*6F
$GPGGA,152014,3833.41,N,12123.65,W,1,4,003,057,M,-026,M*6F
$GPGGA,152016,3833.42,N,12123.62,W,1,4,003,052,M,-026,M*6C...

format:
$GPGGA,152016,3833.42,N,12123.62,W,1,4,003,052,M,-026,M*6C<CRLF>
       1      2       3 4        5 6 7 8   9  10 11  12 13 14

    1: 6 char, UTC (coordinated universal time) time of last fix
       (hhmmss, where h = hours, m = minutes, and s = seconds)
    2: 7 char, latitude (DDMM.HH, where D = degrees, M =
       minutes, and H = hundredths of minutes)
    3: 1 char, latitude, N = north, S = south
    4: 8 char, longitude (DDDMM.HH)
    5: 1 char, longitude, W = west, E = east
    6: 1 char, availability, 1 = available, 0 = not available
       (last position fix more than ten seconds ago)
    7: 1 char, number of satellites being used (3 or 4)
    8: 3 char, HDOP (Horizontal Dilution of Precision)
 9-10: 3 char, antenna height above sea level, M = meters
11-12: 4 char, geoidal height, M = meters
   13: 2 char, checksum (ASCII encoded hexadecimal)
   14: 2 char, Carriage Return and Line Feed (non-printable)
```

Partial Navigation Data File (TEST.NAV in Hexadecimal)

```
CA07070413081C08460000008B0D460B
F407C1030000FE0CB60B040895070000
8B0D450B04086A0B0000770D4C0B0208
3F0F0000810D450BFD0713130000780D
4B0B0108E81600007D0D440B0A08BB1A
0000760D4E0B02088D1E0000780D440B
000884220000620D650BFF072F260000
710D530BFE07FF2900005A0D6A0BFF07...
```

```
format:
    /        header          \   /            sample 1
   |CA 07 07 04   13 08 1C 08| |46 00 00 00   8B 0D 46 0B
       \ /           sample 2            \   /
    F4 07|C1 03   00 00 FE 0C   B6 0B 04 08| |95 07 00 00
    sample 3          \ /            sample 4            \
    8B 0D 45 0B   04 08|6A 0B   00 00 77 0D   4C 0B 02 08|
    /            sample 5            \ /          sample 6
   |3F 0F 00 00   81 OD 45 0B   FD 07|13 13   00 00 78 0D
             \   /           sample 7            \ /
    4B 0B 01 08| |E8 16 00 00   7D 0D 44 0B   0A 08|BB 1A
        sample 8            \   /            sample 9
    00 00 76 0D   4E 0B 02 08| |8D 1E 00 00   78 0D 44 0B
       \ /           sample 10           \   /
    00 08|84 22   00 00 62 0D   65 0B FF 07| |2F 26 00 00
    sample 11         \ /           sample 12           \
    71 0D 53 0B   FE 07|FF 29   00 00 5A 0D   6A 0B FF 07|

header:
    07CA ==> 1994 - year
      07 ==>    7 - day                    file date: 4/07/94
      04 ==>    4 - month

      13 ==>   19 - minute
      08 ==>    8 - hour                   file time: 8:19:08.28
      1C ==>   28 - hundredth of second
      08 ==>    8 - second

sample 1:
00000046 ==>   70 - odometer counts

    0D8B ==> 3467 - x axis signal (flux gate compass)
    0B46 ==> 2886 - y axis signal (flux gate compass)

    07F4 ==> 2036 - solid-state angular rate sensor signal
```

Partial Processed Navigation Data File (TEST.PRN)

```
8:19:08.28   4/07/94
    70, 3467, 2886, 2036
   961, 3326, 2998, 2052
  1941, 3467, 2885, 2052
  2922, 3447, 2892, 2050
  3903, 3457, 2885, 2045
  4883, 3448, 2891, 2049
  5864, 3453, 2884, 2058
  6843, 3446, 2894, 2050
  7821, 3448, 2884, 2048
  8836, 3426, 2917, 2047
  9775, 3441, 2899, 2046
 10751, 3418, 2922, 2047
 11727, 3445, 2893, 2046
 12846, 3432, 2908, 2054
 13677, 3460, 2884, 2051
 14652, 3429, 2910, 2051
 15626, 3466, 2878, 2050
 16597, 3451, 2890, 2050
 17569, 3480, 2865, 2052
 18542, 3455, 2883, 2052
 19517, 3471, 2870, 2045
 20494, 3445, 2895, 2047
 21511, 3468, 2880, 2047
 22454, 3481, 2875, 2046
 23433, 3535, 2838, 2058
 24413, 3543, 2841, 2049
 25410, 3511, 2872, 2052
 26370, 3424, 2963, 2051
 27385, 3373, 3002, 2047
 28325, 3371, 3010, 2050
 29303, 3379, 2998, 2048
 30319, 3390, 2992, 2049
 31258, 3363, 3010, 2051
 32237, 3354, 3024, 2050
 33215, 3339, 3029, 2050
 34193, 3332, 3047, 2049
 35171, 3308, 3048, 2048
 36151, 3299, 2995, 2047
 37130, 3324, 3023, 2049
 38108, 3319, 3021, 2046
 39088, 3321, 3019, 2055...
```

format:

```
    70, 3467, 2886, 2036

    70 - elapsed odometer counts

  3467 - x axis signal (flux gate compass)
  2886 - y axis signal (flux gate compass)

  2036 - solid-state angular rate sensor signal
```

Partial Raw Data File (TEST.RAW in Hexadecimal)

```
0000CA07070413081C08130110089208
0B08E40929010C0889081807DB091401
5C087E085A06B7091601B007E408BD07
E809F800440791082708C70916019D06...
```

```
format:
   /            header              \ /          sample 1
  |00 00 CA 07   07 04 13 08   1C 08|13 01   10 08 92 08
             \   /           sample 2            \ /
   0B 08 E4 09| |29 01 0C 08   89 08 18 07   DB 09|14 01
        sample 3          \   /          sample 4
   5C 08 7E 08   5A 06 B7 09| |16 01 B0 07   E4 08 BD 07
      \ /            sample 5            \   /sample 6...
   E8 09|F8 00   44 07 91 08   27 08 C7 09| |16 01 9D 06
```

```
header:
  0000 - file processed flag; changes to 0xFFFF after processing

  07CA ==> 1994 - year
    07 ==>    7 - day                    file date: 4/07/94
    04 ==>    4 - month

    13 ==>   19 - minute
    08 ==>    8 - hour                   file time: 8:19:08.28
    1C ==>   28 - hundredth of second
    08 ==>    8 - second

sample 1:

Vehicle Velocity
  0113 ==>  275; 6350/275 = 23.090909 meters/second

Left Wheelpath Acceleration (meters/second**2)
  0810 - 0800 = 0010 ==> 16, 16 X 1.9154E-3 = 0.030646

Left Wheelpath Height
  0892 - 0800 = 0092 ==> 146, 146 X 31.25E-6 = 0.004562 meters

Right Wheelpath Acceleration (meters/second**2)
  080B - 0800 = 000B ==> 11, 11 X 1.9154E-3 = 0.021069

Right Wheelpath Height
  09E4 - 0800 = 01E4 ==> 484, 484 X 31.25E-6 = 0.015125 meters
```

Partial Converted Data Files (Hexadecimal)

Velocity (TEST.VEL):

```
2EBAB8413333BB41D70EB841DBBBB641
DBBBBE41DBBBB641D70EB841D70EC041...
```

format:

```
 / sample 1\   / sample 2\   / sample 3\   / sample 4\
|2E BA B8 41| |33 33 BB 41| |D7 0E B8 41| |DB BB B6 41|
 / sample 5\   / sample 6\   / sample 7\   / sample 8\
|DB BB BE 41| |DB BB B6 41| |D7 0E B8 41| |D7 0E C0 41|
```

sample 1:

41B8BA2E (floating point format) ==> 23.090909 meters/second

Left Wheelpath Acceleration (TESTL.ACC):

```
290EFB3C9F4ABC3C2D72343ED9E81CBE
665EB8BE51122EBF302411BE1023C43D...
```

format:

```
 / sample 1\   / sample 2\   /sample 3 \   / sample 4\
|29 0E FB 3C| |9F 4A BC 3C| |2D 72 34 3E| |D9 E8 1C BE|
 / sample 5\   / sample 6\   / sample 7\   / sample 8\
|66 5E B8 BE| |51 12 2E BF| |30 24 11 BE| |10 23 C4 3D|
```

sample 1:

3CFB0E29 ==> 0.030646 meters/second**2

Left Wheelpath Height (TESTL.HGT):

```
0681953BBA498C3B2506813BD578E93B
E17A943B4C37893B986E923BC1CA213B...
```

format:

```
 / sample 1\   / sample 2\   / sample 3\   / sample 4\
|06 81 95 3B| |BA 49 8C 3B| |25 06 81 3B| |D5 78 E9 3B|
 / sample 5\   / sample 6\   / sample 7\   / sample 8\
|E1 7A 94 3B| |4C 37 89 3B| |98 6E 92 3B| |C1 CA 21 3B|
```

sample 1:

3B958106 ==> 0.004562 meters

Partial Slope and Elevation Profile Data Files (Hexadecimal)

Slope Profile (TESTL.SLP):

```
00000000B8AF703A18929D3A84AD25BC
B0B8163C5CCBF63AC0D568B90BE7E63B
8527AEBBA25DDA3AAD7785BB46A7F339
008FA3BA80570E3B1466F63AB878333B...
```

format:

```
/   flag  \   /sample 1 \   /sample 2 \   /sample 3 \
|00 00 00 00| |B8 AF 70 3A| |18 92 9D 3A| |84 AD 25 BC|
/sample 4 \   /sample 5 \   /sample 6 \   /sample 7 \
|B0 B8 16 3C| |5C CB F6 3A| |C0 D5 68 B9| |0B E7 E6 3B|
/sample 8 \   /sample 9 \   /sample 10\   /sample 11\
|85 27 AE BB| |A2 5D DA 3A| |AD 77 85 BB| |46 A7 F3 39|
/sample 12\   /sample 13\   /sample 14\   /sample 15\
|00 8F A3 BA| |80 57 0E 3B| |14 66 F6 3A| |B8 78 33 3B|
```

flag:

00000000 - file processed flag

sample 1:

3A70AFB8 (floating point format) ==> 0.000918 meters/meter

Elevation Profile (TESTL.ELV):

```
7A9F5BBD901F5EBDE8FE60BD73AC55BD
228A62BD4F4B66BD806D67BDD2B271BD
46816CBD5A0C70BD7D676CBD1F706EBD
3D556EBDE38572BD305F76BDA5527BBD...
```

format:

```
/sample 1 \   /sample 2 \   /sample 3 \   /sample 4 \
|7A 9F 5B BD| |90 1F 5E BD| |E8 FE 60 BD| |73 AC 55 BD|
/sample 5 \   /sample 6 \   /sample 7 \   /sample 8 \
|22 8A 62 BD| |4F 4B 66 BD| |80 6D 67 BD| |D2 B2 71 BD|
/sample 9 \   /sample 10\   /sample 11\   /sample 12\
|46 81 6C BD| |5A 0C 70 BD| |7D 67 6C BD| |1F 70 6E BD|
/sample 13\   /sample 14\   /sample 15\   /sample 16\
|3D 55 6E BD| |E3 85 72 BD| |30 5F 76 BD| |A5 52 7B BD|
```

sample 1:

BD5B9F7A ==> -0.053619 meters

Appendix E

VENDOR ADDRESSES

Laser Height Sensors and Optocator Interface Circuit Cards

Selcom AB
Selective Electronic Group
P. O. Box 250
Valdese, NC 28690
(704) 874-4102

Accelerometers

Lucas Schaevitz Inc.
7905 North Route 130
Pennsauken, NJ 08110-1489
(609) 662-8000

Equipment Console and Computer Chassis

ELMA Electronic Inc.
41440 Christy Street
Fremont, CA 94538
(415) 656-3400

GPS Receiver

Magellan Systems Corp.
960 Overland Court
San Dimas, CA 91773
(714) 394-5015

Tape Backup Unit

Colorado Memory Systems Inc.
800 South Taft Avenue
Loveland, CO 80537
(303) 635-1502

CPU, DIO, ADC Circuit Cards and Mass Storage Unit

Xycom Inc.
750 North Maple Road
Saline, MI 48176
(313) 429-4971

DC/DC Converter and AC/DC Power Supply

BICC-VERO Electronics Inc.
1000 Sherman Avenue
Hamden, CT 06514
(203) 288-8001

Keyboard

Marquardt Switches Inc.
2711 Route 20 East
Cazenovia, NY 13035
(315) 655-8050

Video Monitor

Philips Consumer Electronics Company
112 Polk Street
Greeneville, TN 37743
(615) 636-5802

Wheel Sensors

Electro Corporation
1845 57th Street
Sarasota, FL 34243
(813) 355-8411

Magnetic Flux Gate Compass

Etak Inc.
1430 O'Brien Drive
Menlo Park, CA 94025
(415) 328-3825