WHOI-92-05

Copy 1

Woods Hole Oceanographic Institution

WOODS HOLE OCEANOGRAPHIC INSTITUTION 1930

A Data Processing Module for Acoustic Doppler Current Meters

by

Albert J. Plueddemann
Andrea L. Oien
Robin C. Singer
Stephen P. Smith

January 1992

Technical Report

Funding was provided by the Office of Naval Research under Contract No. N00014-89-J-1288.

WHOI-92-05

A Data Processing Module for Acoustic Doppler Current Meters

by

Albert J. Plueddemann, Andrea L. Oien, Robin C. Singer, Stephen P. Smith

Woods Hole Oceanographic Institution
Woods Hole, Massachusetts 02543

January 1992

Technical Report

Funding was provided by the Office of Naval Research under Contract No. N00014-89-J-1288.

Approved for Distribution:

[illegible]

James Luyten, Chairman
Department of Physical Oceanography

Abstract

This report describes the development of a Data Processing Module (DPM) designed for use with an RD Instruments Acoustic Doppler Current Meter (ADCM). The DPM is a self-powered unit in its own pressure case and its use requires no modification to the current meter. The motivation for this work was the desire for real-time monitoring and data transmission from an ADCM deployed at a remote site. The DPM serves as an interface between the ADCM and a satellite telemetry package consisting of a controller, an Argos Platform Transmit Terminal, and an antenna. The DPM accepts the data stream from the ADCM, processes the data, and sends out the processed data upon request from the telemetry controller. The output of the ADCM is processed by eliminating unnecessary data, combining quality control information into a small number of summary parameters, and averaging the remaining data in depth and time. For the implementation described here, eight data records of 719 bytes each, output from the ADCM at 15 minute intervals, were processed and averaged over 2 hr intervals to produce a 34 byte output array.

Keywords: Satellite telemetry, Acoustic Doppler Current Profiler, Argos.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Background and motivation

The desirability of data telemetry from remote, unmanned sites such as deep ocean buoys has been recognized for some time, and several programs at the Woods Hole Oceanographic Institution (Frye and Owens, 1991) and elsewhere have helped to develop this capability. Much of the work to date has concentrated on the telemetry of a limited set of data or status parameters, with little or no data processing or compression. Although more sophisticated systems are being developed (Frye and Owens, 1991; Irish *et al.*, 1991), in some cases the telemetered information from a complex sensor is only sufficient to provide an indication of instrument status. As instrumentation becomes more complex, and as information from multiple instruments is combined, the data rate exceeds that which can be transmitted via conventional means (e.g., Service Argos). By developing a telemetry interface module with data processing capability, it is possible to recover an intelligently composed subset of information from high data rate instrumentation systems deployed on a drifting or moored platform.

This report describes the development of a Data Processing Module (DPM) for use with acoustic Doppler current meters (ADCMs). ADCMs produce prodigious amounts of data in comparison to traditional oceanographic instrumentation like the meteorological sensors and single point current meters discussed by Frye and Owens (1991). During a deployment where a high degree of temporal and spatial resolution is required, the ADCM may generate as much as 1 Kbyte of data per min. Internal recording capacity of up to 40 Mbyte allows this data to be archived, but the low throughput of satellite telemetry systems like Argos (approximately 1 byte/min) make it impossible to transmit the complete data set. In order to be practical for real-time telemetry, the raw data must be processed to create a reduced set of variables or data parameters to be transmitted.

An initial effort to obtain real-time data from an ADCM via satellite was guided by McPhaden at the Pacific Marine Environmental Laboratory (McPhaden *et al.*, 1990; 1991). The result was the PROTEUS mooring, consisting of a downward-looking ADCM mounted in the bridle of a surface buoy, and connected to a processor which transmitted averaged velocity profiles at 24 hr intervals. Although benefiting from their work, we felt that the design requirements (described below) were different enough to warrant a completely independent implementation. The PROTEUS mooring and the DPM are similar in that both provide an interface to the ADCM and do some pre-processing of ADCM data in preparation for satellite telemetry. The principal difference is that on the PROTEUS mooring one microprocessor handled both ADCM data processing and telemetry while the DPM processes the data and offloads it to an external telemetry controller. The design of the DPM as a self-contained, addressable module allows a telemetry controller to collect and transmit data from many different sensors by interrogating each in turn.

The development of the DPM was geared towards a particular initial application, an Arctic data buoy. A recent deployment of an Arctic Environmental Drifting Buoy (AEDB) developed by S. Honjo of WHOI (Honjo *et al.*, 1990) demonstrated the feasibility of a drifting buoy for making velocity and temperature measurements below the Arctic ice pack. The AEDB was deployed in August of 1987 in the pack ice north of Svalbard and drifted for 255 days while collecting data on ice and water temperature, subsurface currents, and particle fluxes. Although the prototype buoy was designed with telemetry capability, the data stream was restricted to buoy position, temperature, and various status parameters. Information from the sub-surface instruments was not available until recovery.

A second-generation Arctic drifter, the Ice-Ocean Environmental Buoy (IOEB), has been developed to succeed the AEDB. The IOEB incorporates a new buoy hull design and a meteorological package in addition to sub-surface instrumentation similar to that deployed on the original buoy. Plans for the IOEB call for the data from both surface and sub-surface sensors to be made available to an Argos satellite transmitter housed in the surface floatation element. This strategy allows the status of the buoy to be monitored more closely during the deployment and will give immediate access to the data regardless of the fate of the drifter. Each IOEB will carry an ADCM, and both ADCMs will be equipped with a DPM to allow the sub-surface current data to be relayed via satellite to a shore based station along with surface meteorological data and buoy position. The purpose of the DPM is to serve as the interface between the ADCM and an Argos telemetry system on the IOEB and to provide a manageable subset of processed ADCM data for transmission.

1.2 Design requirements

The DPM packaging specification called for a self-powered, stand-alone unit in its own pressure case. In a typical deployment, the DPM would be attached to ADCM load cage (Fig. 1) or on the mooring line within a few meters of the ADCM. The power requirement was a battery supply sufficient for deployments of 6 to 9 months. Underwater cabling would provide the communications link between the ADCM and the DPM, and between the DPM and a telemetry controller. The communication requirements were set by the input and output devices; the DPM was designed to process ADCM data in a manner completely transparent to the instrument itself (i.e. requiring no modifications to the ADCM) and to communicate with a generic telemetry controller using the software protocol associated with the Serial ASCII Instrumentation Loop (SAIL; IEEE, 1985).

From the point of view of the DPM there are three important characteristics of the ADCM: The communication protocol, the data stream, and the sample interval. For the application described here, the ADCM was configured to send a binary data stream via EIA-423 at 1200 baud (8 bits, no parity) every 15 minutes. The ADCM data stream, also known as an ensemble, consists of an average over a sequence of many acoustic pulses. For the IOEB application, individual pulses are transmitted once per second, with the data from 40 pulses making up one ensemble. At the end of each ensemble interval, the instrument records the data stream to EPROM memory and transmits the same data through the serial port. The sample interval and serial port enable are preset; the instrument sends out the data strings at fixed intervals based on its own clock and cannot be interrogated through the serial port while in the operational mode. The serial data stream contains a variety of configuration parameters in leader and header arrays, plus data arrays containing velocity, echo amplitude, and data quality information for each bin of each beam. Details of the characteristics of the RD Instruments self-contained ADCM are described in the manufacturer's documentation (RD Instruments, 1991a). A general familiarity with ADCM technical information, data formats, and terminology is assumed throughout this report.

For the application on the IOEB, the DPM was not to communicate directly to an Argos Platform Transmit Terminal (PTT), but rather to a telemetry system consisting of a controller, PTT, and antenna. The controller interrogates the DPM over an EIA-485 loop at 9600 baud using the SAIL software protocol (the SAIL/485 implementation is similar to that described by Park *et al.*, [1991]). Data requests from the controller are made once per hour. Upon receiving a valid SAIL address and a data offload command, the DPM echoes its address and then sends an ASCII-Hex data stream to the controller. Since the timing between the ADCM, the DPM and the controller is arbitrary, the DPM must be able to service a SAIL data request at any time, even when actively communicating with the ADCM or processing data.

The difference in ADCM data output and Argos PTT throughput determines the required data reduction. The 719 byte data stream and 15 min ensemble interval chosen for the IOEB implementation give an effective data rate of about 3 kbytes/hr from the ADCM. The maximum throughput for Argos is in the range of 60 bytes/hr, giving a target for data reduction of at least a factor of 50. For the IOEB deployment, a throughput of only 17 bytes/hr was available for the ADCM data, so that data reduction by about a factor of 170 was necessary. A set of processing routines written in the C programming language, and used previously for laboratory analysis of ADCM data, was implemented on the DPM microcontroller for the purpose of data reduction.

Section two of this report provides a general description of the DPM, with the discussion separated into sub-sections on hardware, communication and control, and software. Four appendices provide more detailed information about the DPM and its use. Appendix A describes a procedure for testing the DPM in the lab and Appendix B describes the deployment procedure. Appendix C is a complete listing of all software used with the DPM. Appendix D provides technical information in the form of tables and figures.

2 Description of the DPM

2.1 Hardware implementation

The DPM hardware layout is sketched schematically in Figure 2. The heart of the electronics is an Intel 87C51FC microcontroller with 32k of external RAM, an external, opto-isolated UART for EIA-423 communication with the ADCM, and an EIA-232 to EIA-485 converter for communication with a telemetry controller. A "watchdog" timer circuit implemented in hardware is used to reset the microcontroller in the event of firmware or communication errors. The power system consists of two battery packs and a switching regulator. The principal system components are discussed in turn below.

The Intel 87C51FC microcontroller was chosen for the DPM application for a number of reasons, the most significant of these being that all the necessary development tools were available to ensure that 'C' code for ADCM processing, developed for mini-computers, could easily be ported to the 87C51. In the addition to this the controller has many other desirable features such as: low power consumption, an idle mode, 32 kbytes of internal EPROM, 256 bytes of internal RAM, an internal UART, and 3 internal 16 bit timers. To keep power consumption low, the microcontroller is clocked by a 2.4576 MHz crystal and the UART crystal is 1.8432 MHz. As currently configured, the DPM uses approximately 23 kbytes of external RAM for data storage, so a 32 kbyte part was used. Since the microprocessor is running at a relatively low clock rate, a 150 ns, low power RAM was selected.

The external National Semiconductor NSC858 UART was selected because of its low power consumption and pin controllable power down mode. In this application the UART is left powered down for the majority of the time to conserve power. The port is set up to receive data only, and is shut down for 14 minutes of the 15 minute period between ADCM sampling intervals. This part was abruptly discontinued by National Semiconductor in early 1991; there is no pin-for-pin compatible replacement. Other similar UARTs are available, but their use would require both hardware and software modifications.

The DPM communicates with a telemetry controller via an EIA-485 link that uses SAIL software protocol. This was accomplished by using a Maxim RS-485 transceiver in conjunction with the microcontroller's internal UART. The Maxim part was selected because of its very low power consumption (1.3 mW typ.) and guaranteed EIA-485 performance. This part on the DPM is always enabled so that the module will respond to its SAIL address at any time.

The watchdog timer circuitry in the DPM is used to provide a power-up reset pulse and to reset the microcontroller if program execution fails. When power is initially applied to the DPM, pin 9 (reset) of the 87C51 is held high for approximately 100 ms, after which it is brought abruptly to ground. This provides the negative going edge (after the supply has stabilized) that is required to properly reset the microcontroller. The timing for the watchdog is generated by a low frequency R-C oscillator that is divided down to approximately 32 minutes (greater than two sampling periods for the ADCM). If the microcontroller does not regularly reset the clock divider, indicating a firmware error condition caused by either a lack of incoming ADCM data or a glitch in program execution, a power-up reset pulse will occur.

RD Instruments warns of a corrosion problem that occurs when ADCMs are used with an external serial device. To avoid this, the ADCM data lines must be electrically isolated from the external device. The design requirements of the DPM dictated use of a micro power isolator capable of data rates up to 9600 baud. A quick look at readily available off-the-shelf components (their power consumption in particular) led to the decision to build an isolator from discrete parts. A spectrally matched, high speed infra-red LED and photo diode were used in conjunction with a discrete current limiting circuit and a micro power operational amplifier to make the isolator. Tests showed that although the circuit could be made to operate at 9600 baud data rates, it was much more tolerant of changes in the EIA-423 levels and to temperature fluctuations when biased for 1200 baud operation. An added advantage of this 1200 baud configuration was that the isolator performed well over such a wide range of signal levels that it could be driven directly from a serial port on a PC. Since high baud rates were not required to handle the 719 bytes of ADCM data at 15 minute intervals, the more robust and versatile 1200 baud configuration was implemented.

The DPM is equipped with two, 7 "D" cell alkaline battery packs. This provides a nominal 10.5 V source with a 28 ampere-hour capacity. De-rating the batteries to 66% of capacity to accommodate their degradation at low temperatures and to allow for some safety factor leaves the DPM with a working capacity of 18.5 ampere-hours. Design goals were to provide the DPM with a service life expectancy of approximately 9 months given the duty cycle appropriate for the IOEB deployment.

The function of the voltage regulator is to convert the battery voltage to a constant 5 volt supply for the DPM. The Maxim MAX638EPA switching regulator was chosen for its high conversion efficiency and small size (low associated parts count). Bench tests showed that the configuration used in the DPM would function at 75% to 92% efficiency over the full range of expected operating conditions. The wide range of efficiency is due to load conditions that vary from 2–30 mA, and from an input (battery) voltage range that varies from 11–6.5 V (6.5 is the minimum input voltage allowed for regulator operation).

2.2 Communication and control

The DPM communicates serially with the ADCM over an optically isolated EIA-423 link and with a telemetry controller via EIA-485. The 1200 baud EIA-423 communications link is accomplished in the DPM by an NSC858 UART which provides a data ready pulse to the 87C51 microcontroller's external interrupt 1 pin. The 87C51 on-chip serial port services the 9600 baud EIA-485 communication link. Both channels use 8 bits and no parity.

A flow chart of DPM communication and control is shown in Figure 3. The DPM is initially powered up by use of an external control line (a shorting plug) or may experience a power-up reset due to the watchdog timer. In normal operation the DPM resets the watchdog timer every 15 minutes, after receipt of each ensemble from the ADCM. This prevents the timer from reaching its 32 minute trigger. In the event that the timer is not reset during a 32 minute period, the watchdog circuit will provide a pulse to reset the DPM. Upon reset, the DPM restarts the firmware, reinitializing all variables and zeroing the output buffers. Thus, a data stream of all zeros from the DPM in response to a SAIL query indicates that a reset has occurred.

In order to save power, the 87C51FC microcontroller is put into a low power idle mode whenever it is not processing data or servicing serial, external or timer interrupts. The microcontroller exits idle mode when it receives an interrupt, so the telemetry controller can address the DPM over the EIA-485 link at any time. The NSC858 UART is turned off by the microcontroller directly after receipt of a complete 719 byte ensemble from the ADCM. While it is off, characters sent by the ADCM would not trigger an external interrupt and therefore not be received by the DPM. However, the UART is turned back on 14 minutes after it is turned off, in response to the microcontroller's internal timer 1 interrupt routine. Since ensembles are sent every 15 minutes by the ADCM, all of the ADCM data is received.

A communications interrupt may be either the EIA-423 data stream from the ADCM or an EIA-485 SAIL command from a telemetry controller. If incoming ADCM data has the proper character count (719 bytes), it is sent to an "unpacking" routine where the packed binary data stream is decoded. An incomplete ensemble (at least 1 byte, but less than 719 bytes) causes a timeout in the communications routine and is counted as a bad ensemble. Ensembles sent to the unpacking routine which do not have the correct checksum, or do not contain the expected header values, are rejected and counted as bad ensembles. Otherwise, the "good ensemble" counter is incremented and the data is stored for later processing.

When the total number of ensembles received (the sum of the good and bad ensemble counters) equals eight, representing two hours of data from the ADCM, the DPM processes the data and stores a 68 character ASCII-Hex data array in one of two output buffers for transmission to the telemetry controller. The double buffering scheme is used to ensure that an existing output array, which has not yet been sent to the controller, will not be corrupted by newly processed data. Within each buffer the output array is arranged in two halves, an "even half" containing data for the even depth bins of the ADCM profile, and an "odd half" containing data for the odd depth bins (the details of the output array contents are discussed in Section 2.3).

Two telemetry controllers, with independent PTTs and Argos antennae, are used on the IOEB to provide a robust data transmission scheme. Each controller interrogates the DPM at 2 hour intervals, but their timing is staggered so that the DPM receives a request for data approximately once per hour. A SAIL data request consists of an attention character (#), a two character address, and a data offload command (R). The DPM responds to a data request with an echo of the address and offload command followed by 34 ASCII-Hex characters of data from the most recently filled output buffer. The two controllers use different addresses (40 and 41) to interrogate the DPM. The DPM considers either of the two addresses valid, sending the even half of the output array in response to a data request which uses the even address (#40R) and the odd half in response to one which uses the odd address (#41R). Thus, transmission of the full DPM output array is split over two independent telemetry systems. The data in the two halves of the output array are arranged so that either half alone provides useful information.

2.3 Data processing

The DPM processing routines were developed from programs used to analyze ADCM data from the Arctic Environmental Drifting Buoy deployment (Plueddemann, 1991). There are two principal processing tasks, "unpacking" the binary ADCM data stream for each ensemble and reducing the data after eight ensembles have been unpacked. For the IOEB application the ADCM data stream is 719 bytes long and contains a header and leader, plus velocity, echo intensity, percent good, and status information for each beam (Fig. 4). Spectral width is not recorded. The unpacking step consists of decoding the packed binary ADCM data stream and filling a floating point array with the decoded, scaled data. The majority of the data reduction is accomplished by eliminating non-essential data and averaging the remaining data in depth and time. Some additional benefit is gained from the creation of summary error and status parameters and judicious scaling based on expected data values.

Upon receiving a 719 byte ensemble from the ADCM, the controlling program passes the array to the unpacking routine. The first step in the unpacking routine is to compute the checksum for the complete ensemble and decode the header. The checksum computed in the unpack routine is compared to the checksum sent with the ensemble. The size of each of the data arrays is extracted from the header (Fig. 5) and checked against the expected array sizes. Any errors found during these checks result in a flag being set to indicate a communication error. The associated data ensemble is counted as a "bad ensemble", it is not stored and will not be included in the averaging step. Ensembles which pass these checks are processed further; the leader data (Fig. 6) is extracted and stored (except for the CTD and bottom track variables, since these functions are not used), and the four data arrays are decoded and stored.

After eight ADCM ensembles have been received, the controlling program calls a sequence of routines that perform several processing steps along with error checking and averaging. The first processing step is to document the status of ADCM operation using information from the leader and the percent good array. The Built In Test (BIT status; RDI, 1991a) code from the leader is used to set two flags, one for beam frequency errors and one for transmitter current errors. The percent good information is combined into a single good/no-good status bit for each averaged bin. Data in a given bin is generally considered to be of poor quality if the percent good value is less than 25. The status bit is set if percent good values less than 25 occur in more than ten percent of the samples in the depth-time averaging interval.

The next processing step is time averaging of the leader data. This consists of a simple arithmetic average over the number of unpacked ensembles in the storage arrays. Under normal conditions 8 ensembles will have been unpacked and stored at the end of a two hour period. If communication errors have occurred, there may be fewer than 8 ensembles to process. There are 14 leader values included in the averaging step: time in decimal days, number of ADCM bins, ensemble number, BIT status, x-axis tilt, y-axis tilt, heading, temperature, high voltage level, transmit current level, low voltage level, and the standard deviations of x-tilt, y-tilt, and heading.

The major processing task involves manipulation of the velocity and echo amplitude data, recorded by the ADCM in beam coordinates, to produce depth-time averaged arrays in earth coordinates. For the IOEB application a 16 m transmit pulse was used and 40 eight-meter bins were recorded. Note that since the transmit pulse sets the fundamental vertical resolution of the measurements, the eight meter bins represent oversampling by a factor of two. The depth averaging implemented for the IOEB deployment is a three bin average of the first 30 bins, resulting in 10 averaged bins. Time averaging is over the 2 hr interval represented by the sequence of 8 ensembles. Before the averaging step, however, several other processing tasks are executed. First, the tilt data is used to interpolate the slant velocity and echo amplitude for each beam onto standard depths. Next, the four beams of slant velocity are combined into two horizontal velocities and two vertical velocity estimates. The heading data is used to rotate the horizontal velocities into earth coordinates. The mean of the two vertical velocities and the mean of the four beams of echo amplitude are computed during the averaging. Thus, the output of this processing step is 4 ten-bin arrays containing depth-time averaged values of east velocity, north velocity, vertical velocity, and echo amplitude.

The final step in the processing is to pack the status flags plus the averaged leader and velocity data into an output buffer for transmission to a telemetry controller. As discussed above, there are two telemetry controllers on the IOEB which request data from the DPM using two different SAIL addresses. Between the two controllers the DPM is interrogated once per hour and the full output array, representing a two hour average, is sent in two halves. It was decided that the hourly transmissions would consist of a header plus status and velocity data for half of the depth bins. The header is repeated for each transmission, but alternating even and odd depth bins are sent in response to the alternating SAIL addresses. A combination of a count bit which alternates between 0 and 1, and an even (0) and odd (1) bin flag are used to keep track of what has been sent (i.e., four successive transmissions would have a [count, even/odd bin] sequence of

[0,0] [0,1] [1,0] [1,1]). This information is useful for putting the half-arrays back together in the proper order, particularly if occasional transmissions are missed. The repeated header and alternating even-odd bin sequence is similar to the scheme described by McPhaden *et al.* (1990) and ensures that usable data spanning the desired depths (albeit with poorer resolution) will be received even if one of the telemetry systems malfunctions.

Due to the limited space (135 bits) allotted to the ADCM for each hourly transmission from the IOEB (Fig. 7), the averaged data had to be reduced further before going into the output buffer. This was accomplished by choosing not to transmit the echo amplitude array and restricting the output header to a subset of the averaged leader data. The floating point horizontal velocity data is scaled and converted into 8-bit integers, the vertical velocity into 4-bits. The first half of the 272 bit output array (Fig. 8) consists of a dummy bit, count bit, even/odd bin bit, even-bin status array (5 bits), error flag array (4 bits), temperature (8 bits), number of ensembles in the average (4 bits), tilt standard deviation (6 bits), heading standard deviation (6 bits), even-bin east velocity array (40 bits), even-bin north velocity (40 bits), and even-bin vertical velocity (20 bits). The second half of the output array (Fig. 8) contains the same count bit, the opposite even/odd bin bit, the same error, temperature, ensemble, and instrument motion data, and the odd-bin status, east velocity, north velocity, and vertical velocity arrays.

The output data is packed into an ASCII-Hex array with two characters per 8-bit word. Thus, it takes 272 bits to store the 68 ASCII-Hex characters. A pointer, set by examining the incoming SAIL address, determines whether the even or odd half of the buffer will be sent to the telemetry controller each hour. Upon receipt by the controller, the 34 ASCII-Hex characters are unpacked, the dummy bit is eliminated, and the remaining 135 bits are added to the data stream for the appropriate PTT (Fig 7).

Acknowledgements

Many hours of useful advice were provided by E. Hobart throughout the project. M. McPhaden kindly provided technical details of the PROTEUS development. The initial effort on this project was supported by seed money from the Woods Hole Oceanographic Institution in the form of a grant from the Vetlesen Fund. Continued work leading to the completion of a field-ready version of the DPM was supported by the Office of Naval Research, Code 1122AR, under Grant No. N00014-89-J-1288.

References

Frye, D. E. and W. B. Owens, 1991. Recent developments in ocean data telemetry at Woods Hole Oceanographic Institution, *IEEE Journal of Oceanic Engineering*, **16**(4), 350–359.

Honjo, S., R. Krishfield and A. Plueddemann, 1990. The Arctic Environmental Drifting Buoy (AEDB): Report of field operations and results, Woods Hole Oceanographic Institution, Woods Hole, MA, Technical Report WHOI–90–2, 128 pp.

IEEE Computer Society, 1985. IEEE standard serial ASCII instrumentation loop (SAIL) shipboard data communication, IEEE, New York.

Irish, J. D., K. E. Morey, G. J. Needell and J. D. Wood, 1991. A current meter with intelligent data system, environmental sensors, and telemetry, *IEEE Journal of Oceanic Engineering,* **16**(4), 319–328.

McPhaden, M. J., H. B. Milburn, A. I. Nakamura and A. J. Shepherd, 1990. PROTEUS - Profile Telemetry of Upper Ocean Currents, *Proc. MTS 1990 Conference, Marine Technological Society,* 353–357.

McPhaden, M. J., H. B. Milburn, A. I. Nakamura and A. J. Shepherd, 1991. *PROTEUS - Profile Telemetry of Upper Ocean Currents, Sea Technology,* February Issue, 10–19.

Park, M. M., R. C. Singer, A. J. Plueddemann and R. A. Weller, 1991. High-speed, real-time data acquisition for vector measuring current meters, *IEEE Journal of Oceanic Engineering*, **16**(4), 360–367.

Plueddemann, A. J., 1991. Internal wave observations from the Arctic Environmental Drifting Buoy, *Journal of Geophysical Research,* submitted.

RD Instruments, 1991a. Self-Contained Acoustic Doppler Profiler Technical Manual, RD Instruments, San Diego, CA, 330 pp.

RD Instruments, 1991b. Deployment Program User's Manual, RD Instruments, San Diego, CA, 34 pp.

Appendices

A. Test procedure

A test procedure meant to be used in verifying the operation of the DPM prior to field deployment is described below. Two IBM compatible PCs, an ammeter, and various test cables are necessary for the complete test (Fig. 9). The ammeter replaces the DPM shorting plug and is used to check current draw by the UART and microcontroller. The procedure can be performed without the ammeter if current checks are not desired. The PCs simulate the ADCM and telemetry controller. The result of the test is a sequence of DPM output records which can be compared to a file containing the expected output. A RMK-7 to DB-25 test cable is needed to connect the EIA-423 side of the DPM to the PC simulating the ADCM. A program called OVERNITE.C (see Appendix C) is run on this PC to send simulated ADCM data transmissions to the DPM. The program accesses a data file called DPMCCS6.BIN containing a sequence of previously recorded ADCM binary data ensembles which have been modified to test a variety of DPM features. A RMG-3BCL connector and cable are used to connect the EIA-485 side of the DPM to an Acromag EIA-485 to EIA-232 converter box. A second cable with two DB-25 connectors attaches the Acromag box to the serial port (COM1) of the PC simulating the telemetry controller. This PC runs a program called TT.C (see Appendix C) which requests processed data records from the DPM using SAIL commands.

The VSG-2BCL connector on the top end cap of the DPM is used to power the module. A dummy plug is used to cover this connector when the DPM is not in use. The RED color-coded shorting plug turns the DPM on by connecting the 10.5 VDC battery packs in the DPM to the input of the switching regulator. After making the initial connection with an ammeter in place of the shorting plug, the

DPM should settle out, within 20 seconds, to a current drain of 2.3 mA $\pm$ 0.3 mA. At this point the DPM UART is on and waiting for data. The DPM will stay in this state until it receives a serial stream from the ADCM (or equivalent simulation). The ADCM serial data enters the DPM via the XSK-7BCL connector. The XSG-3BCL connector is the EIA-485 connection between the DPM and the telemetry controller or controller simulator.

ADCM operation is simulated by connecting the RMK-7 to DB-25 test cable from the DPM to the serial port (COM1) of a PC and running the test program OVERNITE.C. The test program will ask for a data file to use as input. The file DPMCCS6.BIN should be available in the same directory as OVERNITE.C and should be specified as the input file. The number of ensembles should be set to 144 and the time between ensembles to 15 minutes. If a mistake is made in specifying input parameters for OVERNITE.C, reboot the computer, reset the DPM by removing and re-connecting the shorting plug (or ammeter connection), and start again. When OVERNITE.C is running successfully, a message will be sent to the screen as each simulated ADCM data ensemble is sent.

Immediately after receiving a valid ADCM data ensemble, the current draw from the DPM will rise to 5.5 mA $\pm$ 0.5 mA for a few seconds while the DPM unpacks and stores the data in RAM. After receiving and unpacking the data, the DPM goes into an idle mode in which it will respond to EIA-485 SAIL requests from the telemetry controller, but will not accept data from the ADCM. The NSC858 UART is powered down in this state and the microcontroller is idle. The current drawn by the DPM will drop to 1.2 mA $\pm$ 0.3 mA. The idle mode will continue for 14 minutes after which the UART is turned back on and the DPM is ready and waiting for EIA-423 data from the ADCM. The current level will increase back to the original 2.3 mA $\pm$ 0.3 mA until another valid ADCM ensemble is received and the data collection cycle begins again. This cycle will continue unless data is not received from the DPM at the expected 15 minute interval (e.g., the ADCM is disconnected or inoperative and data transmissions stop). If no ADCM ensembles are received, the DPM will wait in the ready state (NSC858 UART on) for EIA-423 data and the microprocessor will be reset every 32 minutes by the watchdog timer.

Any time after the DPM is turned on (using the shorting plug or an ammeter in place of the shorting plug), the module can be addressed via EIA-485 SAIL commands. A 50 foot test cable with a RMG-3BCL connector on one end is provided for this purpose. The other end of the cable should be connected to an Acromag 485/232 converter box. The EIA-232 side of the Acromag box is then connected to the serial port (COM1) of a PC running the telemetry controller simulation program TT.C. (Note that TT.C is not necessary for a simple simulation of the telemetry controller — a terminal emulation program running on the PC with serial communication settings of 9600 baud, no parity, 8 data bits, 1 stop bit can be used to send SAIL commands by hand). It should be started at least 5 minutes, but less than 15 minutes after OVERNIGHT.C for proper results. The TT.C program will request a data file name to which it will log the DPM responses. TT.C will send the first command (without the attention character #) to the DPM within a minute after the interrogation loop is started by selecting a transmission interval. An interval of 60 minutes should be selected. The DPM will respond to the SAIL data offload commands #40R and #41R with an echo of the command (without the attention character #) followed by 34 characters of data and an ETX (ASCII 03) to end the transmission. The data will be all zeros until eight ensembles have been received and processed. The receipt of eight ensembles will take two hours from the time of the first ADCM ensemble. Since the DPM output array is in two halves, transmitted once per hour, the response to the first two SAIL requests will contain zeros.

The processing steps initiated upon receipt of the 8th ADCM ensemble take approximately four minutes to complete. During this time the current drain at the DPM will be 6 mA $\pm$ 0.5 mA. Once the first set of eight ensembles has been processed, the DPM will respond to the SAIL offload commands by sending the processed data. If at any time after this the DPM responds to a data request with a string of zeros, it is an indication that the microprocessor has been reset by the watchdog timer. A listing of the expected DPM output when using the simulated ADCM ensembles in the file DPMCCS6.BIN is given in Figure 10 and in the file DPMCCS6.OUT. The contents of the file created by TT.C during the test procedure should be compared to this listing.

B. Deployment procedure

1. The ADCM and DPM should be installed in the load cage (see Fig. 1) and the cable from the telemetry controller should be accessible at the location of the DPM.

2. Download the desired configuration parameters to the ADCM using the Deployment Configuration Files provided (e.g., I198.DPF) and the RD Instruments Deployment Program (RD Instruments, 1991b). Upon completion of the deployment procedure, the ADCM will be running and sending serial data every 15 minutes. The first ensemble will be sent immediately following the last entry in the deployment sequence. Since the DPM is not connected at this time, the first ensemble received by the DPM will be 15 minutes later.

3. Remove the three dummy plugs from the DPM and store them in the packing crate. Locate the RED color-coded shorting plug in the packing crate. Attach the DPM XSK-7BCL connector to the ADCM XSL-20BCR

I/O connector using the two meter RMK-7FS to XSL-20CCP cable packed with the DPM. Attach the DPM XSG-3BCL connector to the telemetry controller cable.

4. Power up and reset the DPM by connecting the RED color-coded shorting plug to the VSG-2BCL connector on the end cap. The DPM will now be running and waiting for the next ensemble from the ADCM. Note that the first ensemble will not have been received by the DPM (see (2)), but it is assumed that (3) and (4) are completed within 15 min of starting the ADCM, so that the second ensemble will be received.

5. The DPM can be interrogated by the telemetry controller at any time after power-up. The first non-zero data array from the DPM will be obtained after receipt and processing of eight ADCM ensembles, or 2 hrs after receipt of the first ensemble. Since the first ADCM record is not received by the DPM, this will occur approximately 2 hrs 15 min after start-up of the ADCM.

C. Program listings

Four C-language programs associated with the use of the DPM are listed on the following pages.

DPM.C is the main communication and processing program, written in Franklin C, which runs on the Intel 87C51FC microcontroller in the DPM. The compiler used was Franklin C, version 3.07, the assembler was Franklin Assembler version 4.4, and the linker was Franklin Linker L51, version 2.7. A companion program, PC_DPM.C, was written in Microsoft Quick-C and run on an IBM compatible PC. PC_DPM processes data in the same fashion as DPM.C, but reads from and writes to disk files on the PC rather than communicating to the ADCM or the telemetry controller. This version was used during development and testing, but is not reproduced here.

OVERNITE.C and TT.C are used in the deployment simulation procedure and allow the DPM to be exercised in the absence of the other instrumentation to be used in the deployment. OVERNITE.C simulates the operation of the ADCM by taking a file of binary ADCM data and sending it serially to the DPM at a user specified interval. TT.C simulates the telemetry controller by sending alternating SAIL data offload commands (#40R and #41R) to the DPM at an adjustable interval. The data received in response is stored in a file and printed to the screen.

DPMSATOUT.C unpacks the output data array sent to the telemetry controller, and was used during development and testing of the DPM. The program takes groups of 34 ASCII hex characters representing alternating halves of the output data array, combines the appropriate pairs, and then decodes the data.

```
/* DPM.C            */
/* by Robin Singer  */
/* May 1, 1991      */

/* Franklin C compiler version 3.07  */
/* Franklin Assembler version  4.4   */
/* Franklin Linker (L51) version 2.7 */

/* Main routine for ADCP DPM */

/* The DPM is a data processing module which processes ADCP  */
/* ensembles and provides an ASCII Hex string in response to */
/* a SAIL request over an EIA-485 channel.  It runs on an    */
/* Intel 87C51FC microcontroller with 32k of external RAM,   */
/* an NSC 858 UART for EIA-422 communication with the ADCP,  */
/* a 32 minute hardware deadman timer, and an EIA-485 con-   */
/* verter on the microcontroller's serial lines.  Double     */
/* buffering is used and odd and even layer data is sent in  */
/* response to different SAIL addresses.  The microcontroller*/
/* is clocked by a 2.4576Mhz crystal and the UART crystal is */
/* 1.8432Mhz.                                                */

/* The terms record and ensemble are used interchangeably */

#define  TRUE       1
#define  FALSE      0
#define  MAXBYTE    719  /* number of bytes in adcp ensemble */
#define  ENSEMBLE   1    /* value for use with timer 1 flag */
#define  UART       0    /*     "                           */
#define  NBEAM      4    /* number of sonar beams */
#define  NRECA      8    /* number of adcp strings to collect before   */
                         /* processing and moving to the output buffer */
#define  MAXBINS    40   /* max # of bins per record */
#define  MAXLDR     14   /* max # of values of leader to store*/
#define  MAXENS     10   /* max. no. of ensembles to store */
#define  AVGLDR     14   /* # of points in averaged leader */
#define  AVGBINS    10   /* # of depth bins after averaging */
#define  NTYPE      22   /* # of data types (leader+vel+amp) */

#include   <reg51f.h>
#include   <math.h>


bit   attention,  addressed,  offload, oddeven, intschk, nosleep; /* flags */
bit   tlurang, tlerang, altflag, acflag, buffbit, digl;
unsigned char ddata[MAXBYTE], * data ddptr; /* incoming ADCP data buffer/ptr */
unsigned char nproc;   /* count of good adcp data ensembles sent */
unsigned char chcount;   /* count of ASCII hex chars to send via SAIL */
unsigned char unp_err;   /* number of errors from unpack routine */
unsigned char badrec;    /* number of short or bad records */
data int itcount;  /* variable to count timer 1 iterations */
char  * outbuff, buff0[74], buff1[74];  /* processed data output buffers/ptr */


/* declaration of arrays and structures  */

typedef struct stored   /* unpacked data structure */
{
  float ldr[MAXLDR];              /* subset of leader data */
  float vel[NBEAM][MAXBINS];      /* velocity array */
  float amp[NBEAM][MAXBINS];      /* echo amplitude array */
  float gd[NBEAM][MAXBINS];       /* percent good array */
  unsigned char st[16][MAXBINS]; /* bit status array */
} stored;


stored stor[NRECA];      /* array of structures of unpacked data */

typedef struct averaged  /* record-averaged data structure */
{
  float ldr[AVGLDR];
  float jan[3][AVGBINS];
  float amp[AVGBINS];
  unsigned char sb[AVGBINS];
  unsigned char error;
} averaged;
averaged avg;

extern  void    ADSINIT(void);
extern  void    NSCINIT(void);
extern   bit    checkaddr(void);
extern  void    sendptt(char * buffer);
extern  void    process(char * frombuff, char * tobuff);
extern  void    goodnight(void);
extern  void    ttdelay(void);
extern  void    uonidle(void);
extern  void    aclock(void);
extern  void    etimer(void);
extern  void    unp_1_(unsigned char irec, unsigned char *e);
extern  void    pc_leader(unsigned char nrec);
extern  void    janus_echo(unsigned char nrec);
extern  void    repack(unsigned char *bfptr,bit count,unsigned char nrp);
extern  void    uartoff(void);
extern  void    uarton(void);
extern  void    err(unsigned char nproc);
extern  void    notdead(void);
extern  void    prepack(unsigned char *bfptr);

main()
{
  attention = FALSE;   /* initialize SAIL bit flags */
  addressed = FALSE;
  offload   = FALSE;
  nosleep   = FALSE;   /* sleep after loop unless partial record timeout */
  tlurang   = TRUE;    /* start out with UART enabled */
  tlerang   = FALSE;   /* haven't used timer 1 for ensemble time yet */
  oddeven   = 0;
  ddptr     = ddata;   /* point to start of ddata buffer */
  nproc     = 0;       /* initialize number of strings from adcp */
  buffbit   = 0;       /* 1st time send from buff0, repack into buff1 */
  badrec    = 0;       /* initialize bad record counter */
  avg.error = 0x00;    /* initialize global error flag */
  unp_err   = 0;       /* initialize unpack error indicator */
  NSCINIT();           /* initialize UART */
  ADSINIT();           /* initialize 8751 serial communication */
  prepack(buff0);
  prepack(buff1);      /* zero the ASCII hex output buffers */
  notdead();           /* initialize the deadman timer */
  buff0[0] = '4';      /* set up buffers to echo address and offload cmd */
  buff0[1] = '0';
buff0[2] = 'R';
buff1[0] = '4';
buff1[1] = '0';
buff1[2] = 'R';
buff0[37] = '4';
buff0[38] = '1';
buff0[39] = 'R';
buff1[37] = '4';
buff1[38] = '1';
buff1[39] = 'R';
while(1)
{
   intschk = TRUE; /* before sleeping we must prove that */
                   /* we've been through this whole loop */
                   /* interrupt routines set intschk to FALSE */

 /* Have we received the first data byte from the adcp ? */

   if(ddptr==&ddata[1])    /* If so, expect 716 more 1200 baud chars */
   {                       /* which should take about 6 seconds. */
      acflag = ENSEMBLE;   /* so call the alarm clock routine with acflag */
      aclock();            /* set to ENSEMBLE (rings in about 40 sec.) */
   }                       /* if it rings, tlerang gets set to TRUE */
                           /* in the timer 1 interrupt routine */

  /* Have we either received a whole ensemble string from the    */
  /* adcp or timed out after receiving only a partial ensemble ? */

   if((ddptr == &ddata[MAXBYTE]) || ((ddptr>&ddata[0])&&(tlerang)))
   {
      if(acflag==ENSEMBLE)
         TCON &= 0xBF;           /* disable timer 1 */
      ddptr = ddata;             /* reinitialize the incoming data buffer */
      nosleep = FALSE;           /* we'll sleep unless this was a timeout */
      unp_err = FALSE;           /* reinitialize error flag */
      unp_1_(nproc,&unp_err);    /* unpack the data from the adcp */
      if((!tlerang)&&(unp_err==0))
         nproc++;                /* increment good ensemble counter */
      else
      {
         badrec +=1;             /* or bad ensemble counter */
         if(tlerang)
         {
            tlerang = FALSE;     /* reset this flag */
            nosleep = TRUE;      /* we'll stay on till things get right */
         }
      }
      notdead();                 /* reset the deadman timer */

   /* get ready to turn UART off until another ensemble expected */
   /* by setting up the UART wakeup timer */

      tlurang = FALSE;     /* UART alarm rang flag off */
      acflag = UART;
      aclock();            /* alarm clock on for 14 minutes -  when it */
                           /* rings the ISR sets tlurang & puts UART on */

  /* Have we received a full suite of adcp ensembles to process ? */

      if ((nproc + badrec) == NRECA)
      {
         uartoff();          /* uart off while we process */
         err(nproc);         /* store error/status info for repack */
         pc_leader(nproc);   /* compute average leader values */
         janus_echo(nproc);  /* compute average janus velocities */
                             /* and beam-avg echo amplitudes */
         outbuff = buffbit ? buff0 : buff1; /* where to put processed data */
         repack(outbuff+3,buffbit,nproc);   /* as ASCII Hex chars for ARGOS */
         buffbit=(!buffbit);  /* switch buffers - the new data is ready */
         nproc = 0;       /* reinitialize good ensemble counter */
         badrec = 0;      /* reinitialize bad ensemble counter */
         avg.error = 0x00;/* reinitialize global error flag */
         ddptr = ddata;   /* reinitialize the incoming data buffer */
         tlerang = FALSE; /* in case we were out of sync with ADCP */
      }

   }

   if(intschk)   /* If we've been through the whole loop */
   {
      /* If UART alarm clock rang or a timeout occurred */
      if((tlurang)||(nosleep))
         uonidle();    /* idle with the UART on */
      else
         goodnight();  /* otherwise, low power - idle with the UART off */
   }
  }
}
```

```
/* dpmfns.c */
/* by Robin Singer   */
/* May 15, 1991  */

/* Franklin C compiler version 3.07  */
/* Franklin Assembler version  4.4   */
/* Franklin Linker (L51) version 2.7 */


extern data int itcount; /* iteration counter for UART sleep interval */
extern int  tcount;  /* iteration counter for ensemble receive timer */
extern unsigned char * data ddptr;
extern unsigned char ddata[];


#define TRUE      1
#define FALSE     0
#define PDLAY     100   /* power down delay to wait for stop bit */
#define DMDLAY    450   /* deadman timer reset delay */
#define BUFFLEN   34    /* length of SAIL output data string  */
sbit            UON = 0x91;      /* NSC UART on pin */
sbit            DEADMAN = 0x90; /* Deadman Timer (4060) Reset Line */

#include        <reg51f.h>
#include        <math.h>


/* power down NSC and then put 8751 into idle mode */

void    goodnight(void)
{
   unsigned char n;

   for(n=0;n<PDLAY;n++)       /* delay a bit */
       ;
   UON = FALSE;    /* power down the uart by clearing P1.1 */
   ddptr = ddata;  /* when we wake up we will be ready for a new enemble */
        PCON |= 0x01;   /* go into idle */
}

/* power down NSC but leave 8751 on */

void  uartoff(void)
{
   unsigned char n;

   for(n=0;n<PDLAY;n++)
       ;
   UON = FALSE;
}

/* leave UART on but put micro into idle */

void    uonidle(void)
{
   unsigned char n;

   for(n=0;n<PDLAY;n++)       /* delay a bit */
       ;
   PCON |= 0x01;   /* go into idle */
}

/* UART alarm clock routine */
/* sets up the timer to wake up the NSC UART in time to listen */
/* for the next ADCP ensemble */
/* also used for timeout clock in case a partial ensemble or    */
/* stray characters arrive at the UART */

void  aclock(void)
{
   TMOD  =  0x10;  /* timer 1 to timer mode 1 (16 bits) */
   TH1   =  0x00;  /* 16 bits at 2.4M gives .3 sec */
   TL1   =  0x01;
   TCON  |= 0x40;  /* set the timer 1 run control bit to turn timer 1 on */
   itcount  =  0;
   IE |= 0x08;     /* enable timer 1 interrupt */

 /* The Timer 1 ISR (mttimint.a51) will increment itcount and reset the */
 /* timer unless: (1) acflag equals 0 and 2625 iterations (14 minutes)  */
 /* has passed, in which case it turns the uart on and sets the tlurang */
 /* flag to TRUE ...or... (2) acflag equals 1 and 144 iterations (about */
 /* 40 seconds) have passed in which case it sets the tlerang flag.     */

}

void notdead(void) /* prevents hardware reset by resetting 4060 */
{
   int  delay;

   DEADMAN = 1;   /* send reset to deadman circuit (4060) */
   for(delay=DMDLAY; delay>0; delay--)
       ;
   DEADMAN = 0;              /* end of deadman reset pulse */

}

/* Initialize output buffers with SAIL echo and zeroes */

void prepack(unsigned char *bufptr)
{
   unsigned char n;

   *bufptr++ = '4';
   *bufptr++ = '0';
   *bufptr++ = 'R';
   for(n=0;n<BUFFLEN;n++)
      *bufptr++ = 0x30;
   *bufptr++ = '4';
   *bufptr++ = '1';
   *bufptr++ = 'R';
   for(n=0;n<BUFFLEN;n++)
      *bufptr++ = 0x30;
}
```

```
/*    dapro.h
      This header file contains symbolic constants and external data
      structure declarations that are used in the functions called by
      the dapro/dpm program.
*/

#define MAXBYTE    719     /* max number of bytes per record(serial input)*/
#define NRECA        8     /* number of records to accumulate */
#define MAXBINS     40     /* number of depth bins per record */
#define MAXLDR      14     /* max number of values of leader to store */
#define NBEAM        4     /* number of sonar beams used in calc */
#define AVGLDR      14     /* number of leader values averaged and stored */
#define AVGBINS     10     /* number of depth bins afer averaging */
#define NBINA        3     /* no. of bins averaged(MAXBINS div. by AVGBINS)*/
                /* for dt and dz functions */
#define NTYPE       22     /* no. of data types(leader+velocity+amplitude)*/
#define SD_FACTOR    3     /* multiple of standard dev. for threshold set */

extern unsigned char ddata[MAXBYTE];        /* input data buffer */

typedef struct stored                      /* stored data strucure */
{
        float ldr[MAXLDR];                 /* subset of leader data */
        float vel[NBEAM][MAXBINS];         /* velocity array */
        float amp[NBEAM][MAXBINS];         /* echo amplitude array */
        float gd[NBEAM][MAXBINS];          /* percent good array */
        unsigned char st[16][MAXBINS];     /* bit status array */
} stored;

typedef struct averaged    /*struct of data averaged over NRECA # of records*/
{
        float ldr[AVGLDR];                 /* averaged leader array */
        float jan[3][AVGBINS];             /* averaged janus velocity array */
        float amp[AVGBINS];                /* averaged echo amplitude array */
   unsigned char sb[AVGBINS];           /* calc status using percent good */
   unsigned char error;                 /* error code per output cycle */
} averaged;

extern stored stor[NRECA];                 /* array of unpacked records */

extern averaged avg;                       /* averaged data array */

extern float dt_thresh[NTYPE];             /* dt threshold storage buffer */
extern float dz_thresh[2*NBEAM];           /* dz threshold storage buffer */

extern float eab[4][10];                   /* beam and bin averaged echo amp */
```

```
/*      unp_1_.h
 *
 *      include file for function unp_1_.c
 */

/* constants and scale factors */
#define N_LO     1               /* nibble is the least significant */
#define N_HI     0               /* nibble is the most significant */
#define RES_16  65536.0          /* number of 16 bit counts */
#define RES_12  4096.0           /* number of 12 bit counts */
#define RES_8   256.0            /* number of 8 bit counts */
#define DEG     360.0            /* conversion factor for degrees */
#define VLTSC_H 0.17             /* high voltage scale factor */
#define VLTSC_L 0.05             /* low voltage scale factor */
#define AMPSC   0.01             /* transmit current scale factor (low power)*/
#define AMP_DB  0.45             /* echo amplitude conversion factor */
#define SIGMA   0.1              /* std dev scale factor for pitch and roll */
#define SIGMA_H 1.0              /* std dev scale factor for heading */
#define H       24.0             /* scale for converting hours to decimal day */
#define M       60.0             /* scale for converting sec,min to dec day */
#define VEL_SC  0.25   /* velocity scaling factor for cm/sec*/
#define AMP_DB  0.45  /* echo amplitude conversion factor to dB */
#define ZEROB   0

/* variable and array sizes */
#define MAXBYTE          719

/* change MAXENS in unp_1_.h to NRECA, MAXENS eliminated */

#define BINHD_SIZ       14      /* size of binary header */
#define CHCKSUM_SIZ     2       /* size of checksum */
#define LONG_LD         63      /* no. bytes in long leader */
#define SHORT_LD        49      /* no. bytes in short leader */
#define VEL_SZ          240     /* no. bytes of velocity  data per record */
#define SPW_SZ          0       /* no. bytes of spectral width data */
#define AMP_SZ          160     /* no. bytes of echo amplitude data */
#define GD_SZ           160     /* no. bytes of percent good data */
#define ST_SZ           80      /* no. bytes of status data per record */


/* deployment dependent parameters */
char year[5];                    /* starting year */


/* input array sizes */

int maxbytes;               /* bytes in the record */
unsigned short lead_sz;     /* byte size of leader */
unsigned short vel_sz;      /* byte size of velocity data */
unsigned short spw_sz;      /* byte size of spectral width data */
unsigned short amp_sz;      /* byte size of echo amplitude data */
unsigned short gd_sz;       /* byte size of % good data */
unsigned short st_sz;       /* byte size of status data */

/* bit manipulation subroutines and working variables */

double ppow(double base,double n);
unsigned short comb(unsigned char msb, unsigned char lsb);
unsigned short combn(int which, unsigned char by, unsigned char nibble);
int splitb(unsigned char by, unsigned char *lsn, unsigned char *msn);
```

```
short signb(unsigned short ival);
unsigned char lsn,msn;
unsigned short ival;
unsigned short ius;
short is;


/* function and variables for time manipulation */
int julian(int m, int d, int y);        /* function to return julian day */
float date;             /* time/date converted to decimal julian day */
float dtime;            /* time converted to decimal day */
int julday;             /* julian day */
int oldyear;            /* number of julian days in previous year */


/* leader variables */  /* check occurances and data types */
char month[3];          /* date and time string variables*/
char day[3];
char minute[3];
char hour[3];
char second[3];
char hundsc[3];                     /* hundredths of seconds */
float timbp;                        /* time between pings */
unsigned short pens; /* rec */      /* pings per ensemble */  /* short -> char */
unsigned short nbin;                /* bins per ping */       /* short -> char */
double blen;                        /* bin length */
unsigned short tint;                /* transmit interval */
unsigned short delay;               /* delay */
unsigned short ens;  /* rec */      /* ensemble number */     /* locate occur */
unsigned short status_b;            /* bit status */
unsigned short snt;                 /* signal to noise threshold */
unsigned short pgdt;                /* percent good threshold */
float tiltx;                        /* x-axis tilt (pitch) */
float tilty;                        /* y-axis tilt (roll) */
float head;                         /* heading */
float temp;                         /* temperature (adcp) */
float vhi;                          /* high voltage input */
float xmit;                         /* transmit current */
float vlow;                         /* low voltage input */
float sdx;                          /* std deviation of pitch */
float sdy;                          /* std deviation of roll */
float sdh;                          /* std deviation of heading */
```

```
#pragma ot(3)

/* Revising Andrea's final code to incorporate it with */
/* the final microcontroller code - May 10, 1991 - rcs */

/*      unp_1_.c  is a modified version of
        unpack.c file and function from the dpm prototype program

        It unpacks one, hence the _1_, record of binary data stored in
        a one dimensional array called ddata.

        The RDI self-contained Acoustic Doppler Current Meter is set up
        to put out binary ddata to characterize an event of  ? minutes.
        The data is read in as bytes and sent to various routines for
        'unpacking' which involves various byte and nibble manipulations.
        The code is generalized to extract data from variable size input
        records with the required array size information read from the
        first 14 bytes of the record.  Data is presumed to have the long
        leader.

        After unpacking and scaling, a storage buffer containing leader
        data and velocity, echo amplitude, percent good pings, and status
        bits for each bin of each beam is filled.  If percent good pings
        is not recorded, spectral width data is substituted.  Time is
        stored as decimal yearday.  The RDI data record does not contain
        the year of deployment, so this must be handled within the
        program.

        Bit and byte variable names have been eliminated from the functions
        called by this function.

*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "unp_1_.h"
#include "dapro.h"


void unp_1_( unsigned char irec, unsigned char *e)
                                /*irec is storage buffer record index */
                                /* e is a temporary error indication */

{
        unsigned char  i;       /* loop variable - max value < nbin */
        unsigned char  ibeam;   /* beam loop variable */
        unsigned short j;       /* byte location for velocity */
        unsigned short k;       /* byte location for echo amp */
        unsigned short l;       /* byte location for % good */
        unsigned short m;       /* byte location for status */
        unsigned short n;       /* byte location for spectral width*/
        unsigned short s;       /* loop var for sum */
        unsigned short sum;     /* program calculated checksum */
        unsigned short check;   /* value of checksum unpacked from input */

        void dec_long (unsigned char irec);

        strcpy(year,"1991");
        oldyear = 0;
        *e = 0;                         /* error indicator set to 0 */
        j = 0;
        k = 0;
        l = 0;
        m = 0;
        n = 0;

        /* decode binary header to determine total no. bytes in input
        *  record (maxbytes) and no. bytes of each data type */
        maxbytes = comb(ddata[0],ddata[1]) + CHCKSUM_SIZ;
        lead_sz = comb(ddata[2],ddata[3]);
        vel_sz = comb(ddata[4],ddata[5]);
        spw_sz = comb(ddata[6],ddata[7]);
        amp_sz = comb(ddata[8],ddata[9]);
        gd_sz = comb(ddata[10],ddata[11]);
        st_sz = comb(ddata[12],ddata[13]);


        /* check to see if checksum is equal to program calculated sum */
        /* calculate new checksum */
        sum = 0;
        for (s = 0; s < (MAXBYTE-2); s++)
        {
            sum = sum + ddata[s];
            if (sum == 65535)
               sum = 0;
        }
check = ((ddata[MAXBYTE-2] * 256) + ddata[MAXBYTE-1]);
if (check != sum)
{
        *e += 1;
}


        /* check for correct header values */
        /* if not correct increment temporary error variable e */


        if ( maxbytes != MAXBYTE )
                *e += 1;

        if ( lead_sz != LONG_LD )
                *e += 1;

        if ( vel_sz != VEL_SZ )
                *e += 1;

        if ( spw_sz != SPW_SZ )
                *e += 1;

        if ( amp_sz != AMP_SZ )
                *e += 1;

        if ( gd_sz != GD_SZ )
                *e += 1;

        if ( st_sz != ST_SZ )
        *e += 1;

    if (*e > 0)
    {

        if  ( (0x4 & avg.error) == 0 )
            avg.error = avg.error + 4;
        return;
    }


    /* unpack leader */
    dec_long(irec);


    if (nbin != MAXBINS)
    {                              /* rcs modification */
          nbin = MAXBINS;
          avg.error |= 0x08;
}

    /* loop through depth bins unpacking velocity, echo amp,
    *  percent good pings, and status */
    for ( i = 0; i < nbin; i++)
    {

        /* compute byte locations for this bin */
        j = (i*6)       /* velocity, 6 bytes per bin */
          + (BINHD_SIZ + lead_sz);
        k = (i*4)       /* echo amplitude, 4 bytes per bin */
          + (BINHD_SIZ + lead_sz + vel_sz + spw_sz);
        l = (i*4)       /* percent good, 4 bytes per bin */
          + (BINHD_SIZ + lead_sz + vel_sz + spw_sz + amp_sz);
        m = (i*2)       /* status, 2 bytes per bin */
          + (BINHD_SIZ + lead_sz + vel_sz + spw_sz + amp_sz + gd_sz);
        n = (i*4)       /* spectral width, 4 bytes per bin */
          + (BINHD_SIZ + lead_sz + vel_sz);

        /* unpack velocity (cm/s) */
          splitb(ddata[j+1],&lsn,&msn);
          stor[irec].vel[0][i] =
                VEL_SC * signb(combn(N_LO,ddata[j],msn));
          stor[irec].vel[1][i] =
                VEL_SC * signb(combn(N_HI,ddata[j+2],lsn));
          splitb(ddata[j+4],&lsn,&msn);
          stor[irec].vel[2][i] =
                VEL_SC * signb(combn(N_LO,ddata[j+3],msn));
          stor[irec].vel[3][i] =
                VEL_SC * signb(combn(N_HI,ddata[j+5],lsn));


        /* unpack echo amplitude (dB) */
          for ( ibeam = 0; ibeam < 4; ibeam++ )
          {
            stor[irec].amp[ibeam][i] =
                AMP_DB * comb(ZEROB,ddata[k+ibeam]);
          }

        /* unpack percent good (%) */
          for ( ibeam = 0; ibeam < 4; ibeam++ )
          {
            stor[irec].gd[ibeam][i] =
                (float) comb(ZEROB,ddata[l+ibeam]);
          }


        /* if spectral width is recorded, substitute for % good
           (% good and spec width should not occur together)
           THIS SHOULD NOT HAPPEN
           SPW_SZ should equal 0 for this deployment because
           instrument is sending radial-beam velocities
        */
        if ( gd_sz == 0 && spw_sz != 0 )
        {
          for ( ibeam = 0; ibeam < 4; ibeam++ )
          {
            stor[irec].gd[ibeam][i] =
               2.* VEL_SC * comb(ZEROB,ddata[n+ibeam]);
          }
        }


        /* unpack status bits */
          splitb(ddata[m],&lsn,&msn);
          /* mask for first bit, first beam */
          stor[irec].st[0][i] = lsn & 001;
          /* shift and mask for 2nd bit, 1st beam */
          stor[irec].st[1][i] = (lsn >> 1) & 001;
          /* etc. */
          stor[irec].st[2][i] = (lsn >> 2) & 001;
          stor[irec].st[3][i] = (lsn >> 3) & 001;
          stor[irec].st[4][i] = msn & 001;       /* second beam */
          stor[irec].st[5][i] = (msn >> 1) & 001;
          stor[irec].st[6][i] = (msn >> 2) & 001;
          stor[irec].st[7][i] = (msn >> 3) & 001;
          splitb(ddata[m+1],&lsn,&msn);
          stor[irec].st[8][i] = lsn & 001;       /* third beam */
          stor[irec].st[9][i] = (lsn >> 1) & 001;
          stor[irec].st[10][i] = (lsn >> 2) & 001;
          stor[irec].st[11][i] = (lsn >> 3) & 001;
          stor[irec].st[12][i] = msn & 001;      /*fourth beam */
          stor[irec].st[13][i] = (msn >> 1) & 001;
          stor[irec].st[14][i] = (msn >> 2) & 001;
          stor[irec].st[15][i] = (msn >> 3) & 001;

    }

}
/*********************************************************************/

/* function dec_long           decodes the long leader (63 bytes) */
```

```
void dec_long(unsigned char irec)
                              /*irec is the storage buffer record index */

{
        unsigned int i;    /* byte location for beginning of leader ??? */
        unsigned char j;   /* j < lead_sz (63)  */

        /* loop through leader bytes (index j) */
        for ( j = 1; j < lead_sz; j++)      /* ?? lead_sz not visible ??? */
        {
                /* offset to proper byte location in ddata (index i) */

                i = j + BINHD_SIZ - 1;

                switch (j)
                {

                case 1: /* date */
                        /* get mo,dy,hr,min,sec from ddata buffer */
                        sprintf(month,"%b2.2x",ddata[i]);
                        sprintf(day,"%b2.2x",ddata[i+1]);
                        sprintf(hour,"%b2.2x",ddata[i+2]);
                        sprintf(minute,"%b2.2x",ddata[i+3]);
                        sprintf(second,"%b2.2x",ddata[i+4]);
                        /* compute date in decimal julian days */
                        julday = julian(atoi(month),atoi(day),atoi(year));
                        dtime = atoi(hour)/H
                          + atoi(minute)/(H*M)
                          + atoi(second)/(H*M*M);
                        date = dtime + julday + oldyear;

                        /* check for new year */
                        if ( oldyear == 0)
                        {
                                if ( strcmp(month,"12") == 0
                                  && strcmp(day,"31") == 0)
                                        oldyear = julday;
                        }
                        break;
                case 6: /* time between pings (decimal seconds) */
                        sprintf(minute,"%b2.2x",ddata[i]);
                        sprintf(second,"%b2.2x",ddata[i+1]);
                        sprintf(hundsc,"%b2.2x",ddata[i+2]);
                        timbp = atoi(minute)*60
                          + atoi(second)
                          + atoi(hundsc)/100.0;
                        break;
                case 9: /* pings per ensemble */
                        pens = comb(ddata[i],ddata[i+1]);
                        break;
                case 11:        /* bins per ping */
                        nbin = comb(ZEROB,ddata[i]);
                        break;
                case 12:        /* bin length (meters) */
                        ius = comb(ZEROB,ddata[i]);
                        if( ius > 5 )  ius = 0;
                        blen = ppow((double)2.0,(double)ius);
                        break;
                case 13:        /* transmit interval (meters) */
                        tint = comb(ZEROB,ddata[i]);
                        break;
                case 14:        /* delay after transmit (nearest meter) */
                        delay = comb(ZEROB,ddata[i]);
                        break;
                case 16:        /* ensemble number */
                        ens = comb(ddata[i],ddata[i+1]);
                        /* printf("ensemble #      = %6d\r",ens); */
                        break;
                case 18:        /* built-in test status */
                        status_b = comb(ZEROB,ddata[i]);
                        break;
                case 20:        /* signal-to-noise threshold */
                        snt = comb(ZEROB,ddata[i]);
                        break;
                case 21:        /* percent good threshold */
                        pgdt = comb(ZEROB,ddata[i]);
                        break;
                case 22:        /* "pitch" (deg) */
                        tiltx = DEG*comb(ddata[i],ddata[i+1])/RES_16;
                        if(tiltx > 180.0)
                          tiltx = tiltx - DEG;
                        break;
                case 24:        /* "roll" (deg) */
                        tilty = DEG*comb(ddata[i],ddata[i+1])/RES_16;
                        if(tilty > 180.0)
                          tilty = tilty - DEG;
                        break;
                case 26:        /* heading (deg) */
                        ius = comb(ddata[i],ddata[i+1]);
                        head = DEG*((short)ius)/RES_16;
                        break;
                case 28:        /* temperature (deg C) */
                        temp = 45. - 50.
                          * (int) comb(ddata[i],ddata[i+1]) / RES_12;
                        break;
                case 30:        /* high voltage input (volts) */
                        vhi = VLTSC_H*comb(ZEROB,ddata[i]);
                        break;
                case 31:        /* transmit current (amps) */
                        xmit = AMPSC*comb(ZEROB,ddata[i]);
                        break;
                case 32:        /* low voltage input (volts) */
                        vlow = VLTSC_L*comb(ZEROB,ddata[i]);
                        break;
                case 56:        /* pitch std deviation (deg) */
                        sdx = SIGMA*comb(ZEROB,ddata[i]);
                        break;
                case 57:        /* roll std deviation (deg) */
                        sdy = SIGMA*comb(ZEROB,ddata[i]);
                        break;
                case 58:        /* heading std deviation (deg) */
                        sdh = SIGMA_H*comb(ZEROB,ddata[i]);
                        break;
                }
        }

        /* fill the storage array with desired leader variables */
```

```
stor[irec].ldr[0] = date;
stor[irec].ldr[1] = (float) nbin;
stor[irec].ldr[2] = (float) ens;
stor[irec].ldr[3] = (float) status_b;   /*   bit;   */
stor[irec].ldr[4] = tiltx;
stor[irec].ldr[5] = tilty;
stor[irec].ldr[6] = head;
stor[irec].ldr[7] = temp;
stor[irec].ldr[8] = vhi;
stor[irec].ldr[9] = xmit;
stor[irec].ldr[10] = vlow;
stor[irec].ldr[11] = sdx;
stor[irec].ldr[12] = sdy;
stor[irec].ldr[13] = sdh;
```

```
/* err.c              Andrea Olen                     5-9-91
 * The err function is called in dapro and pc_dpm to fill part of a global
 * error variable in the average structure.  The average structure holds
 * the output values packed in repack.  The status bits in the output
 * stream are calculated here from the % good ADCP data and stored in
 * the average structure also.
 */



#include <stdio.h>
#include "dapro.h"
#include <math.h>


void err(unsigned char nrec)

{

   float stat;
   unsigned char i,j,irec,ibin;
   unsigned char avbin,av,bdcount;



/* calculate status bit for each average bin */

/* initializations */
av = 0;         /* counts the number of bins to be averaged together */
avbin = 0;      /* current avg bin whose status is being calculated  */
bdcount = 0;    /* incremented everytime percent good < 25 in the avg bin */
/* initialize status bit values in avg structure */
for (i=0;i<AVGBINS;i++)
avg.sb[i] = 0;
if(nrec==0)
   return;


/* loop for counting percent good < 25 over NBINA bins, NRECA records and
 * 4 beams(NBEAM).  If bdcount >= (NBINA*nrec*NBEAM*.10) then the status bit
 * for that avg bin is set to 1.*/

   for (ibin = 0;ibin < (NBINA * AVGBINS);ibin++)
   {
       av = av+1;
       for (irec = 0;irec<nrec;irec++)
       {
           for (i = 0;i<NBEAM;i++)
          {
               /* printf(" %3.2f",stor[irec].gd[i][ibin]); */
               if (stor[irec].gd[i][ibin] < 25.0)
                  bdcount = bdcount + 1;

          }
          /* printf("\n"); */
       }
       if (av == NBINA)
       {
       /*printf("bdcount = %d = %1.2fratio\n",bdcount,((float)bdcount/96));*/
          if (bdcount >= (.10*NBINA*nrec*NBEAM))
             avg.sb[avbin] = 1;
          else avg.sb[avbin] = 0;
          av = 0;                   /* reset bin counter */
          avbin = avbin + 1;        /* increment index for status array */
          bdcount = 0;              /* reset good counter */
       }
   }


          /* set error codes from status byte info stored in leader array */
          /* instrument receiver errors from the status byte */

          for (irec=0;irec<nrec;irec++)
          {
              stat = stor[irec].ldr[3];
              if (((16 < stat) && (stat < 80)) &&
                   ((stat != 32) || (stat != 48) || (stat != 64)))
              {
                 if ( (0x01 & avg.error) == 0)
                    avg.error += 1;
               }

          /* instrument transmitter errors: very low, low and high current */
              if ((80 < stat) && (stat < 83))
                 {
                   if ( (0x02 & avg.error) == 0)
                      avg.error += 2;
                 }
          }

}
```

```
/*      function pc_leader.c

*       This routine accepts an array of stored ADCM data set up by
*       function unp_1_.c and deglitched by pc_dtfix.c.  Each data
*       type in the leader is averaged over the number of records
*       (ensembles) in the storage array.  An averaged data array
*       is filled with the resulting values.
*/

#include <stdio.h>
#include <math.h>
#include "dapro.h"


void pc_leader( unsigned char nrec )
                                /* nrec is number of records in storage */

{
        unsigned char irec;             /* record counter */
        unsigned char j;                /* leader data type index */

   /* initialize average leader buffer */

   for(j=0;j<MAXLDR;j++)
      avg.ldr[j] = 0.0;

   /* leave function if nrec==0 */

   if(nrec==0)
      return;

   /* loop through each data type in leader */
        for (j = 0; j < MAXLDR; j++)
        {

          /* sum stored records for each data type */
          for (irec = 0; irec < nrec; irec++)
          {
            avg.ldr[j] = avg.ldr[j] + stor[irec].ldr[j];
          }

          /* compute means */
          avg.ldr[j] = avg.ldr[j] / (float) nrec;

        }
}
```

```
/*  Revising Andrea's final Janus_ec.c program for use in the   */
/*  final microcontroller code - May 10, 1991 - rcs             */

/*  A. Plueddman  A. Oien                               May 9, 1991
 *  janus_echo is a function used in the dapro and pc_dpm programs.  It
 *  combines the janus and echo functions of the dpm prototype program,
 *  eliminating some redundant calculations and loops.
 *  Separately these routines do the following:
 *      Janus:
 *      This routine accepts an array of stored ADCM data set up by
 *       function unpack.c and computes janus horizontal and vertical
 *       velocities.  Heading correction and tilt correction are made
 *       for each record using compass and inclination data from
 *       leader.  Pitch and roll error angles are set equal to zero.
 *       Magnetic declination is set equal to zero and soundspeed
 *       correction factor is set equal to one since actual values
 *       will be unkown for a drifter deployment.
 *
 *       Janus East and North velocities plus combined vertical
 *       velocity (average of the two janus w's) are averaged over
 *       the number of stored records and the number of bins
 *       specified by NBINA.  Results are stored in averaged data
 *       array.
 *
 *      Echo:
 *      This routine accepts an array of stored ADCM data set up by
 *      the funtion UNP_1_.c and computes the four beam average echo
 *      amplitude.  Prior to averaging, the amplitude for each beam is
 *      normalized by the average of the last four depth bins.  The
 *      normalized amplitude is then averaged over the number of stored
 *      records and the number of bins specified by the NBINA. Results
 *      are stored in averaged data array.
*/

#include <stdio.h>
#include <math.h>
#include "dapro.h"

#define SSCOR   1.000      /* sound speed cbrrection factor */
#define STHETO   0.866      /* sin of beam angle from horiz */
#define CTHETO   0.500     /* cos of beam angle from horiz */
#define PERR     0.0               /* pitch error angle */
#define RERR    0.0        /* roll error angle */


void janus_echo(unsigned char nrec)     /* nrec == no. of records processed */


{

        unsigned char nbin;              /* no. depth bins */
        unsigned char ibin;              /* depth bin counter */
        unsigned char irec;              /* record (ensemble) counter */
        unsigned char ibeam;             /* beam index */
        unsigned char i;                 /* general purpose counter */
        unsigned char j;                 /* depth bin index */
        unsigned char k;                 /* averaged depth bin index */
        unsigned char bad;               /* counter for heading check */
        unsigned char end;               /* ends loop-good heading found */
        signed short head[NRECA];        /* headings forced to be in range */
```

```
        float z0scale[NBEAM];    /* scales for obs to std depths */
        float z0[MAXBINS];       /* observed depth buffer */
        float v0[MAXBINS];       /* obs depth slant vel buffer */
        float a0[MAXBINS];       /* obs depth echo amp buffer */
        float z[MAXBINS];        /* standard depth buffer */
        float v[MAXBINS];        /* std depth slant vel buffer */
        float a[MAXBINS];        /* std depth echo amp buffer */
        float nlevel[NBEAM];     /* "noise level" for normalization */
        float uscale,wscale;     /* scale factors for janus vel */
        float decl,hd;           /* declination and heading */
        float sinhd,coshd;       /* sin and cos of heading */
        double phi,rho;          /* pitch, roll angle buffers */
        double ju,jv,jw1,jw2;    /* janus velocity buffers */
        double arg;              /* math function argument */


        int ierr1;               /* function error flag */
        int ierr2;               /* function error flag */

        int nf,nl,jerr;          /* lintp function error flags */

    extern int lintp(float *x0,float *y0,unsigned char n0,float *x,float *y,unsigned char
n,int *nf,int *nl,int *ierr);
    /* linear interpolation function */

        float pi;
        pi = 4.0 * atan(1.0);

        /* initialize parameters */
        decl = 0.0;     /* declination angle, assumed unknown */
        nbin = (int) stor[0].ldr[1];    /* number of depth bins */
        /* check for correct nbin value */
        if (nbin != MAXBINS)
        {
           nbin = MAXBINS;


        }


    /* initialize averaging buffers */

    for(k=0;k<AVGBINS;k++)
    {
       avg.amp[k] = 0.0;
       for(i=0; i<3; i++)
       {
          avg.jan[i][k] = 0.0;
       }
    }

    /* leave janus_echo if nrec==0 */

    if(nrec==0)
       return;

        /* check for out of range headings and replace with the nearest
         * record in range. If there aren't any good headings among the
         * records, "bad" will be equal to the number of records processed.
* Headings are all set to zero and the processing error flag is set.
*/
bad = 0;

/* initialize headings with readings from leader  & find no. bad */
for (irec = 0; irec < nrec; irec++)
{
   head[irec] = stor[irec].ldr[6];
   if ( (stor[irec].ldr[6] < -180.0) || (stor[irec].ldr[6] >180.0) )
      bad += 1;
}

/* substitute nearest record with heading in range */
if ( (bad > 0) && (bad < nrec) )
{
  bad = 0;
  for(irec=0;irec<nrec;irec++)
  {
    end = 0;
    if (( head[irec] < -180.0 ) || ( head[irec] > 180.0 ))
    {
     bad +=1;
     for (i = 1; i < nrec;i++)
     {
        if (((irec-i)>=0) && ((irec-i)<nrec) && (end==0))
        {
          if ((stor[irec-i].ldr[6] > -180.0) &&
                    (stor[irec-i].ldr[6] < 180.0))
          {
             head[irec] = (signed short) stor[irec-i].ldr[6];
             end = 1;
          }
        }
        if ((end == 0) && ((irec+i) < nrec))
        {
          if ((stor[irec+i].ldr[6] > -180.0) &&
                   (stor[irec+i].ldr[6] < 180.0))
          {
             head[irec] = (signed short) stor[irec+i].ldr[6];
             end = 1;
          }
        }
     }
    }
  }
}

/* all headings in this sample period out of range, subst. zero */
if (bad == nrec)
{
  for(irec=0;irec<nrec;irec++)
  {
    head[irec] = 0;
  }
  if ( (avg.error & 0x08) == 0 )
     avg.error += 8;
}

/* display velocities in stor.vel */
/*  printf("stor[0].vel[0][0] is %8.2f",stor[0].vel[0][0]); */
/*  for (irec = 0; irec < nrec; irec++)
    {
       printf("\nirec = %d stor.vel[0][0] = %4.2f .vel[1][0] = %4.2f",
              irec,stor[irec].vel[0][0],stor[irec].vel[1][0]);
    }  */

    /* loop through the stored records, transform four beams of
    *  slant velocity to janus u,v,w1,w2 for each bin */
    for (irec = 0; irec < nrec; irec++ )
    {
      /* set up tilt angles for this data point
      *  include correction for un-gimballed pitch axis */
      phi = (double) ((stor[irec].ldr[4] + PERR) * pi / 180.);
      rho = (double) ((-stor[irec].ldr[5] + RERR) * pi / 180.);

      /* check to see that phi is between +/- 20 deg(pi/180) radians */
      /* if not set to 0 and set processing error flag */

      if ((phi < -0.35) || (phi > 0.35))
      {
         phi = 0;
         if ((avg.error & 0x08)==0)
            avg.error += 8;
      }

      /* same error check for angle rho */

      if ((rho < -0.35) || (rho > 0.35))
      {
         rho = 0;
         if ((avg.error & 0x08) == 0)
             avg.error += 8;
      }


      arg = sqrt (1.0 - ((sin(phi)*sin(rho))*(sin(phi)*sin(rho))));
      phi = asin( (sin(phi)*cos(rho)) / arg );

      /* compute scale factor for each beam to transform
      *  depths in tilted frame to depths in fixed frame */
      z0scale[0] = (float) ( cos(rho)*cos(phi) * STHETO
        - sin(rho)*cos(phi) * CTHETO );
      z0scale[1] = (float) ( cos(rho)*cos(phi) * STHETO
        + sin(rho)*cos(phi) * CTHETO );
      z0scale[2] = (float) ( cos(rho)*cos(phi) * STHETO
        - sin(phi) * CTHETO );
      z0scale[3] = (float) ( cos(rho)*cos(phi) * STHETO
        + sin(phi) * CTHETO );

      /* perform translation correction on slant velocities
          and echo aplitudes */
      for (ibeam = 0; ibeam < NBEAM; ibeam++)
      {
        /* set up arrays for interpolation, depth positive */
        for (j = 0; j < nbin; j++)
        {
          z0[j] = j * z0scale[ibeam];       /* observed depths */
          z[j] = j * STHETO;                /* standard depths */
  /* vel at observed depths */
  v0[j] = stor[irec].vel[ibeam][j];
  /* amplitude at observed depths */
  a0[j] = stor[irec].amp[ibeam][j];
 }

 /* do linear interp for v[j] = vel at standard depths,
 *  replace vel at obs depths with vel at std depths.
 *  do linear interp for a[j] = amp at standard depths,
 *  replace amp at obs depths with amp at std depths. */

 ierr1 = lintp(z0,v0,nbin,z,v,nbin,&nf,&nl,&jerr);
 if (jerr < (nbin/2))
 {
    for (j = 0; j < nbin; j++)
        stor[irec].vel[ibeam][j] = v[j];
 }
 else if ((avg.error & 0x08) == 0)
        avg.error += 8;

 ierr2 = lintp(z0,a0,nbin,z,a,nbin,&nf,&nl,&jerr);
 if (jerr < (nbin/2))
 {
    for (j = 0; j < nbin; j++)
     stor[irec].amp[ibeam][j] = a[j];
 }
 else if ((avg.error & 0x08) == 0)
        avg.error += 8;

}     /* end translation correction loop */

/* conversion from slant vel to janus vel scales like
*  a/(2*cos(theta0)) for u,v and a/(2*sin(theta0)) for w
*  where theta0 = 60 deg is the beam angle from horiz
*  and a = ss/1536 is soundspeed correction factor. */
uscale = SSCOR / (2.0 * CTHETO);
wscale = SSCOR / (2.0 * STHETO);

/* compute janus velocities and do rotation correction */
for (j = 0; j < nbin; j++)
{
  /* combine slant velocities to form janus velocities */
  ju = (double) ( uscale *
      (stor[irec].vel[0][j] - stor[irec].vel[1][j]) );
  jv = (double) ( uscale *
      (stor[irec].vel[3][j] - stor[irec].vel[2][j]) );
  jw1 = (double) ( wscale *
      (stor[irec].vel[0][j] + stor[irec].vel[1][j]) );
  jw2 = (double) ( wscale *
      (stor[irec].vel[2][j] + stor[irec].vel[3][j]) );

  /* rotation correction follows RD's conventions for
  *  pitch and roll angles, and the implicit assumption
  *  that the janus velocities are representative of the
  *  component velocities for each beam */
  stor[irec].vel[0][j] = (float)
      ( ju * cos(rho)
      + jw1 * sin(rho) );
  stor[irec].vel[1][j] = (float)
      ( jv * cos(phi)
      + ju * sin(rho)*sin(phi)
      - jw2 * sin(phi)*cos(rho) );
  stor[irec].vel[2][j] = (float)
      ( jw1 * cos(phi)*cos(rho)
      - ju * sin(rho)*cos(phi)
      + jv * sin(phi) );
  stor[irec].vel[3][j] = (float)
      ( jw2 * cos(phi)*cos(rho)
      - ju * sin(rho)*cos(phi)
      + jv * sin(phi) );

}

/* correct heading for magnetic declination, correct
*  janus horizontal velocities for heading */
hd = head[irec] + decl;        /* add decl to heading */
arg = (double) (hd * pi / 180.0);
sinhd = (float) sin(arg);  coshd = (float) cos(arg);

for (j = 0; j < nbin; j++)
{
  ju = stor[irec].vel[0][j];
  jv = stor[irec].vel[1][j];
  stor[irec].vel[0][j] = ju * coshd + jv * sinhd;
  stor[irec].vel[1][j] = jv * coshd - ju * sinhd;
}

   /*   display current record
             for (ibeam = 0; ibeam < NBEAM; ibeam++) {
               printf(" irec %d  ibeam %d ",irec,ibeam);
               for (j = 0; j < 5; j++) {
                 printf(" %10.4f",stor[irec].vel[ibeam][j]);
               }
               printf("\n");
             } */

}  /* end of record processing loop */


/* echo amplitude processing routine: noise level */

/* estimate "noise level" by averaging last four bins of each
 * beam for all of the stored records */
for (ibeam = 0; ibeam < NBEAM; ibeam++)
{
    nlevel[ibeam] = 0.0;
    for (irec = 0; irec < nrec; irec++)  /* ?? use i ?? */
    {
        for (j = (nbin-4); j < nbin; j++)
        {
            nlevel[ibeam] = nlevel[ibeam] + stor[irec].amp[ibeam][j];
        }
    }
    nlevel[ibeam] = nlevel[ibeam] / (float) (4 * nrec);
}

/* average velocity and echo amplitude routines combined */
/* average janus velocities and amplitudes over records and bins */
```

```
    /* initialize counters, start loop */
    k = 0;
    ibin = 0;
    for (j = 0; j < (NBINA * AVGBINS); j++ )
    {
      ibin = ibin + 1;

      /* average over stored records for this depth bin */
      for (irec = 0; irec < nrec; irec++)
      {
          /* janus east velocity */
          avg.jan[0][k] = avg.jan[0][k] + stor[irec].vel[0][j];
          /* janus north velocity */
          avg.jan[1][k] = avg.jan[1][k] + stor[irec].vel[1][j];
          /* combined vertical vel */
          avg.jan[2][k] = avg.jan[2][k]
               + stor[irec].vel[2][j] + stor[irec].vel[3][j];

          /* average over beams as well as records for
             amplitude average of this depth bin */
          for (ibeam = 0; ibeam < NBEAM; ibeam++)
          {
              avg.amp[k] = avg.amp[k] + stor[irec].amp[ibeam][j];
          }
      }

      /* check for end of depth averaging cycle */
      if (ibin == NBINA)
      {

          /* compute the average velocities */
          avg.jan[0][k] = avg.jan[0][k] / (float) (nrec*ibin);
          avg.jan[1][k] = avg.jan[1][k] / (float) (nrec*ibin);
          avg.jan[2][k] = avg.jan[2][k] / (float) (2*nrec*ibin);

          /* compute the average amplitude */
          avg.amp[k] = avg.amp[k] / (float) (nrec*ibin*NBEAM);

          /* reset counters */
          ibin = 0;
          k = k + 1;

      }
    }      /* end of bin loop for velocities and amplitudes */
}
```

```
/* A. Olen */

/* repack.c -- function for dpm, pc_dpm and dapro.c programs --   2-25-91
     This function packs velocity data, error message data and bin status
     data from the data processing routines into 60 ascii hex characters
     pointed to by bfptr.  bfptr points to signed characters elements.
     The least significant nibble of each element is an ascii hex character.
     There are bit, nibble and byte variables coded into output.

     The output created by repack is used to serially send ascii hex
     data to the unit that transmits to argos.

     See repack.memo to decode data stream.
*/

#include <stdio.h>
#include "dapro.h"
#include <math.h>


void repack(unsigned char *bfptr, bit count, unsigned char nrp)

                         /* nrp is number of records processed */


{

   unsigned char four_to_1(unsigned char *four);
   signed char f_to_c(float f1);
   unsigned char f_to_n(float f0);
   unsigned char c_hexa(unsigned char ch);
   void b_to_n(signed char byt,unsigned char *l,unsigned char *m);
   unsigned char k[3],out4bit[4];
   int round(float f);
   float t1,sdrp;
   signed char vel;
   unsigned char send,t2,sdh,lsn,msn,i,j,index,errout,ibin,ibeam,irec;



/* stuff counter, even/odd, and status bit (bin0/1) into output buf pointer*/

out4bit[0] = 0;
out4bit[1] = (unsigned char) count;
out4bit[2] = 0;
out4bit[3] = avg.sb[0];
send = four_to_1(out4bit);
*bfptr = c_hexa(send);
out4bit[2] = 1;
out4bit[3] = avg.sb[1];
send = four_to_1(out4bit);
*(bfptr + 37) = c_hexa(send);

/* stuff remaining status bits into +1 (even bins) and +35 (odd bins) bfptr*/
out4bit[0] = avg.sb[2];
out4bit[1] = avg.sb[4];
out4bit[2] = avg.sb[6];
out4bit[3] = avg.sb[8];
send = four_to_1(out4bit);
*(bfptr + 1) = c_hexa(send);
```

```
out4bit[0] = avg.sb[3];
out4bit[1] = avg.sb[5];
out4bit[2] = avg.sb[7];
out4bit[3] = avg.sb[9];
send = four_to_1(out4bit);
*(bfptr + 38) = c_hexa(send);


/* stuff 4 error bits from global error variable into output buffer */

*(bfptr + 2) = c_hexa(avg.error);
*(bfptr + 39) = c_hexa(avg.error);



  /* code temperature  float value found in avg.ldr[7] */
  /* add 5 and divide by .098 temp */
/* -5 deg () to 20 deg () over one byte of data (.098 per increment of 256) */


   t1 = (avg.ldr[7] + 5.0);
   if (t1 < 0)
      t1 = 0.0;
   if (t1>=24.99)
      t1 = 24.99;
   t2 = (unsigned char) (round (t1/0.098));

   /* stuff temp */
   b_to_n(t2,&lsn,&msn);
   *(bfptr + 3) = c_hexa(msn);
   *(bfptr + 40) = c_hexa(msn);
   *(bfptr + 4) = c_hexa(lsn);
   *(bfptr + 41) = c_hexa(lsn);

   /* stuff number of records processed */
   if ( (0 <= nrp) && (nrp <= NRECA) )
   {
     *(bfptr + 5) = c_hexa(nrp);
     *(bfptr + 42) = c_hexa(nrp);
   }
   else
   {
    *(bfptr + 5) = c_hexa(15);
    *(bfptr + 42) = c_hexa(15);
   }

   /* stuff sdx, sdy and sdh */
   sdrp = ((avg.ldr[11] + avg.ldr[12])/2)*10;/* standard dev. roll & pitch */
   if (sdrp > 62.0)
      sdrp = 62.0;
   if (sdrp < 0)
      sdrp = 63.0;
   send = (unsigned char) round(sdrp);
   lsn = 0x0f & send;
   *(bfptr + 6) = c_hexa(lsn);     /* lsn of sdrp stored in even output */
   *(bfptr + 43) = c_hexa(lsn);    /* lsn of sdrp stored in odd output */
   send = (send >> 2) & 0x0c;
   if (avg.ldr[13] > 62.0)
      avg.ldr[13] = 62.0;
    if (avg.ldr[13] < 0)
       avg.ldr[13] = 63.0;
    sdh = (unsigned char) round(avg.ldr[13]);
    lsn = 0x0f & sdh;
    send = ((sdh >> 4) & 0x03) | send;
    *(bfptr + 7) = c_hexa(send);   /* 2msbits of sdrp and sdh respectively */
    *(bfptr + 44) = c_hexa(send);
    *(bfptr + 8) = c_hexa(lsn);    /* lsn of sdh */
    *(bfptr + 45) = c_hexa(lsn);


  /* initialize places in array where east, north and vertical velocities will
       be stored in the even portion of the array */
    k[0] = 8;
    k[1] = 18;
    k[2] = 28;

    for (i=0; i<3; i++)
    {
        index = k[i];
        for (j=0; j<10; j+=2)
        {
            if ((i==0) || (i==1))
                vel = f_to_c(avg.jan[i][j]);
                                        /* east and north convert to char */
            else vel = f_to_n(avg.jan[i][j]);
                                        /* vert convert to nibble (+/- 8) */
            index = index + 1;
            b_to_n(vel,&lsn,&msn);
            if ((i==0) || (i==1))
            {
                *(bfptr + index) = c_hexa(msn);
                index = index + 1;
            }
            *(bfptr + index) = c_hexa(lsn);
                                        /* verical vel in lsn of vel byte */
        }
    }


  /* process and stuff east, north and vert velocities for averge odd bins */

  /* initialize places in array where east, north and vertical velocities will
       be stored in the odd portion of the array */

    k[0] = 45;
    k[1] = 55;
    k[2] = 65;

    for (i=0; i<3; i++)
    {
        index = k[i];
        for (j=1; j<10; j+=2)
        {
            if ((i==0) || (i==1))
                vel = f_to_c(avg.jan[i][j]);
                                        /* east and north convert to char */
            else vel = f_to_n(avg.jan[i][j]);
                                        /* vert convert to nibble (+/- 8) */
            index = index +1;
```

```
                b_to_n(vel,&lsn,&msn);
                if ((i==0) || (i==1))
                {
                    *(bfptr + index) = c_hexa(msn);
                    index = index + 1;
                }
                *(bfptr + index) = c_hexa(lsn);
                                        /* verical vel in lsn of vel byte */
            }
        }

    /* print the output values pointed to by bfptr */
  /* for (i=0;i<74;i+=5)
     {
        printf("%d = %c\t%d = %c\t%d = %c\t%d = %c\t%d = %c\n",i,*(bfptr + i),
        i+1,*(bfptr+i+1),i+2,*(bfptr+i+2),i+3,*(bfptr+i+3),i+4,*(bfptr+i+4));
     }
     printf("\n\n");
  */


  }



  /* SUBROUTINES FOR REPACK */


    /* b_to_n splits byte( byt ) into l (least significant nibble)
     * and m (most significant nibble) each packed into the least significant
     * nibble of the two character bytes they are found in
     */

     void b_to_n(signed char byt,unsigned char *l,unsigned char *m)


     {
          *l = byt & 0x0f;
          *m = (byt >> 4) & 0x0f;

     }



    /* f_to_c converts float variable passed to function into a signed char
     * If the float is < -127 it returns _127, if > 127 it returns 127.
     * The fractional portion is lost when cast into a signed char.
     */

     signed char f_to_c(float f1)
     {
         int round (float f);
         if (f1<-127)
            return -127;
         if (f1>127)
            return 127;
         return (signed char) (round(f1));
     }
```

39

```
/* f_to_n converts float variable passed to function into an unsigned char
 * with the four bit value packed in the lsn (least significant nibble) of
 * the byte.
 * 7 is added to the value of the float.
 * If the float is < -7 then -7 is returned, if > 8 then 8 is returned.
 * The float is the rounded; fractions between .45 and .55 are rounded to
 * the nearest even integer.
 */

 unsigned char f_to_n (float f0)
 {
    int round (float f);
    f0 = round(f0);
    if (f0 < -7)
        f0 = -7;
    if (f0 > 8)
        f0 = 8;
    return (unsigned char) (f0 + 7);
 }

/* round function converts float to int and rounds the value to the next
 * larger absolute value integer if the fractional component is > 0.50,
 * smaller absolute value integer if the fractional component is < 0.50
 * if the fraction = .50 then it is rounded to the closest even integer
 */


int round (float f)
{
   float t;
   t = f - (int) f;                    /* t = fractional component of f */
   if (fabs(t) < 0.50)
      return (int) f;



   if (t > .5)
      return (int) (f+1);
   if (t ==  0.50)
   {
      if (((int) f % 2) == 0)       /* if int of f is even */
         return (int) f;
      else return (int) (f+1);     /* when int of f is odd add 1 */
   }


   if (t < -.50)
      return (int) (f-1);
   if (t==-0.50)
   {
      if (((int) f % 2) == 0)      /* if int of f is even number */
         return (int) f;
      else return (int) (f-1);     /* when int of f is odd */
   }
}

/* c_hexa changes the signed and unsigned char in output array into hex asci
 * values in the least significant nibble of the unsigned char it returns
 * The steps in the routine are:
 * 1:checks to see that the most sig nibble is zero, returns an error flag
 *   if it is not
 * 2:the least sig nibble is converted to an ascii char by adding 30(hex)
 * if it is under 10 and adding 37 (hex) if it is 10 to 15.
 */

unsigned char c_hexa(unsigned char ch)
{

  ch = ch & 0x0f;

  if (ch<10) return (unsigned char) (ch + 0x30);
  if (ch>9) return (unsigned char) (ch + 0x37);
}

/* four_to_1 takes four char variables and puts the least significant bit of
 * each one into the least significant nibble of the output char variable
 * The nibble is stuffed from the right so the first of the characters
 * pointed to will contribute the most significant bit in the nibble.
 */
 unsigned char four_to_1 (unsigned char *four)
 {
    unsigned char b,i;
    b = 0;
       for (i=0;i<4;i++)
       {
          *four = (*four & 0x01);
          b = (b | *four) <<1;
          four++;
       }
    b = b>>1;
    return b;
 }
```

```
/*  ppow.c */
/*  by Robin Singer January 1991  */
/*  used ppow to avoid conflict with qc library function pow */
/*  added to adcp code because Franklin C does not have pow function */

/* ppow raises the base to the nth power */
/* it is assumed that n>=0 */

/* parameters declared as a double to be consistent with MSC but n really */
/* needs to be an integer for this function to work */


double ppow(double base,double n)
{
   double p;
   int i;

   p=1.0;
   for(i=1; i<=(int)n; i++)
       p=p*base;
   return p;
}
```

```
/* takes one byte and returns the most and least significant 4 bits (1 nibble)
   packed into an unsigned byte with leftmost digits zero filled              */

splitb(byt,lsn,msn)

unsigned char byt,*lsn,*msn;

{
        *lsn = byt & 017;
        *msn = (byt & ~017) >> 4;
        return;
}
```

```
/* takes an unsigned short integer, determines if it is greater than 2047
   and generates a signed integer by wrapping the 12 bit value          */

short signb(ival)

unsigned short ival;

{
        short i;

        if ( ival > 2047)
                i = ival - 4096;
        else
                i = ival;
        return(i);
}
```

```
/* takes two bytes and packs them into an unsigned 16 bit integer which is
   returned */

   unsigned int comb(msb,lsb)

   unsigned char lsb,msb;

   {
        return (((unsigned int)msb << 8)| lsb);
   }
```

```
/* The function combn combines 1 byte and 1 nibble into an unsigned
   16 bit integer
   If "which" is zero, the nibble lives in the high order bits
   else nibble lives in the lowest four bits

*/

/* modified by rcs for use with Franklin C v 3.07 */

unsigned short combn(int which, unsigned char by,unsigned char nibble)


{

        if (!which)
                return(((unsigned int)nibble << 8) | by);
        else
                return(((unsigned int)by << 4) | nibble);
}
```

```
/* this routine returns the julian day associated with a month, day and year */

int julian(m,d,y)

int m;
int d;
int y;

{
        int j;

        switch(m)       {

        case 1:
                j = d;
                break;
        case 2:
                j = d + 31;
                break;
        case 3:
                j = d + 59;
                break;
        case 4:
                j = d + 90;
                break;
        case 5:
                j = d + 120;
                break;
        case 6:
                j = d + 151;
                break;
        case 7:
                j = d + 181;
                break;
        case 8:
                j = d + 212;
                break;
        case 9:
                j = d + 243;
                break;
        case 10:
                j = d + 273;
                break;
        case 11:
                j = d + 304;
                break;
        case 12:
                j = d + 334;
                break;
        default:
                j = -1;
                break;
        }

        if ( y % 4 == 0 && m > 2)
                j = j + 1;
        return(j);
}
```

```
/*      lintp( x0, y0, n0, x, y, n, nf, nl, ierr )
 *
 *      Computes interpolated values of ordinate y from original
 *      data arrays (x0,y0) given new abscissae values x.
 *        x0, y0 = original data arrays of length n0
 *        x      = array of new abscissae values, length n,
 *                 in increasing order
 *        y      = interpolated ordinate values
 *
 *      If all points in x are contained in the interval (1,n0)
 *      then nf=0, nl=n-1, and ierr=0.  If some points in x are
 *      outside the interval (1,n0), then nf, nl are first and
 *      last points within the interval and ierr is the number
 *      of points outside the interval.  Values of y outside the
 *      interval are set to zero.
 */

int lintp( x0, y0, n0, x, y, n, nf, nl, ierr )
int *nf, *nl, *ierr;
unsigned char n0,n;
float x0[], y0[], x[], y[];

{
        int i, j;              /* counters */
        int jf, jl;            /* start, end pts for interp */
        float deltax, slope;   /* difference and slope */

        /* set defaults */
        *ierr = 0;
        *nf = 0;
        *nl = n-1;

        /* check for initial x values less than smallest x0 */
        for( i = 0; i < n; i++ )  {
          if( x[i] < x0[0] )  {
            *ierr = *ierr + 1;
            y[i] = 0.0;
            *nf = i + 1;
            }
          if( x[i] >= x0[0] )  {
            break;
            }
          }

        /* check for final x values greater than largest x0 */
        for( i = (n-1); i > -1; i-- )  {
          if( x[i] > x0[n0-1] )  {
            *ierr = *ierr + 1;
            y[i] = 0.0;
            *nl = i - 1;
            }
          if( x[i] <= x0[n0-1] )  {
            break;
            }
          }

        /* find the index for values of x0 nearest to but less
        *  than x[nf] and nearest to but greater than x[nl] */
        for( j = 1; j < n0; j++ )  {
          if( x0[j] > x[*nf] )  {
            jf = j - 1;
            break;
            }
          }
        for( j = (n0-2); j > -1; j-- )  {
          if( x0[j] < x[*nl] )  {
            jl = j + 1;
            break;
            }
          }

        /* find nearest neighbors to x in interval (jf,jl)
        *  and interpolate for y */
        i = *nf;
        for( j = jf; j < jl; j++ )  {
          while( x[i] <= x0[j+1] && i <= *nl )  {
            deltax = x[i] - x0[j];
            slope = ( y0[j+1] - y0[j] ) / ( x0[j+1] - x0[j] );
            y[i] = y0[j] + slope * deltax;
            i = i + 1;
            }
          if( i > *nl ) break;
          }

        return (0);
}
```

```
/* Overnite.c */
/* May 15, 1991 */
/* by Robin C. Singer */

/* Read in binary data from a user specified data file */
/* and send it out via COM1 at 1200,N,8,1               */
/* Using Greenleaf functions because QC calls seem to  */
/* do something odd with the binary character Hex 1A.  */

/* Modification of asimi2 (asi2) to make it repeatedly */
/* send a user specified number of records at a user   */
/* specified interval.                                 */

/* This program simulates an ADCP and is for use testing */
/* the 87C51 based ADCP controller.                      */

     #define FALSE 0
     #define TRUE  1

     #include <stdio.h>
     #include <graph.h>
     #include <conio.h>
     #include <time.h>
     #include <stdlib.h>
     #include <bios.h>
     #include "asiports.h"
     #include "gf.h"

     int  getch(void);

     int  ensembs, intchar, n, m, status,dly, st, nummin;
     int  now,past;
     char fnarr[25];
     unsigned char binchar;

     FILE  *fp,*fpc;

     void inittime(void);
     void waitmin(void);

main()
{
     _clearscreen(_GCLEARSCREEN);
     status = asiopen(COM1,ASOUT|BINARY|NORMALRX,100,100,1200,P_NONE,1,8,1,1);
     if(status<ASSUCCESS)
     {
       printf("port not open. status = %d\n",status);
       exit(1);
     }
     printf("\n\n\nEnter name of binary file to use for adcp simulation  ");
     scanf("%s",fnarr);
     printf("\n\nHow many 719 byte ensembles are in %s ? ",fnarr);
     scanf("%d",&ensembs);
     printf("\n\nHow many minutes between ensembles? (no less than 4) ",nummin);
     scanf("%d",&nummin);
     printf("\n %d ensembles in %s,",ensembs,fnarr);
     printf("\n\n Waiting for the 00 second");
     printf("\n Then sending an ensemble every %d minutes", nummin);
     inittime();
     while(1)
        {
          if(fp=fopen(fnarr,"rb"))
          {
             for(n=1;n<=ensembs;n++)
             {
               for(m=1;m<=719;m++)
               {
                  intchar = fgetc(fp);  /* fgetc returns an int */
/*                binchar = intchar;*/    /* put it into unsigned char */
                  st=-1;
                  while(st)
                      st=(asiputc(COM1,intchar));
               }
               printf("\nEnsemble number %d sent.",n);
               waitmin();
             }
          fclose(fp);
          }
        }
}

/* Wait for second to be 00 and initialize 'now' */

void inittime(void)
{
  int    sec;
  char   second[2],minute[3];
  char   *timeptr;
  struct tm *curtime;
  time_t bintime;

  time(&bintime);   /* time in seconds since midnite 1/1/70 GMT */
  curtime=localtime(&bintime);      /* convert to local time */
  timeptr = 11+(asctime(curtime)); /* assign ptr to pt at hour */
  strncpy(second,timeptr+6,2);
  while((sec=atoi(second))!=0)
  {
      time(&bintime);   /* time in seconds since midnite 1/1/70 GMT */
      curtime=localtime(&bintime);      /* convert to local time */
      timeptr = 11+(asctime(curtime)); /* assign ptr to pt at hour */
      strncpy(minute,timeptr+3,2);
      strncpy(second,timeptr+6,2);
  }
  now=atoi(minute);
}


void waitmin(void)
{
  int    yet;
  char   minute[3];
  char   *timeptr;
  struct tm *curtime;
  time_t bintime;

  past = now;
  yet = FALSE;
  while(!yet)
  {
    time(&bintime);   /* time in seconds since midnite 1/1/70 GMT */
```

```
     curtime=localtime(&bintime);     /* convert to local time */
     timeptr = 11+(asctime(curtime)); /* assign ptr to pt at hour */
     strncpy(minute,timeptr+3,2);
     now=atoi(minute);
     if(((now<nummin)&&((now+60-past)==nummin))||((now-past)==nummin))
         yet = TRUE;   /* nummin minutes has passed */
   }

}
```

```
/* TT.C */
/* May 17, 1991 */
/* by Robin C. Singer */

/* Tattletale Emulation Program */
/* Addressing the DPM by alternating between the two addresses */
/* with an offload command, at a user selectable interval.     */
/* Receiving and displaying the data send back by the DPM.     */

     #define FALSE 0
     #define TRUE  1


     #include <stdio.h>
     #include <graph.h>
     #include <conio.h>
     #include <time.h>
     #include <stdlib.h>
     #include <bios.h>
     #include "asiports.h"
     #include "gf.h"

     int  getch(void);

     int  ensembs, intchar, n, m, status,dly, st, nummin, altflag;
     int  now,past,exit_,chhit,flag60;
     char fnarr[25];
     unsigned char binchar;

     FILE  *fp,*fpp;

     void inittime(void);
     void sendcmd(int flag);
     void waitmin(void);
     void getdata(void);

main()
{
     _clearscreen(_GCLEARSCREEN);
     status = asiopen(COM1,ASINOUT|BINARY|NORMALRX,100,100,9600,P_NONE,1,8,1,1);
     if(status<ASSUCCESS)
     {
       printf("port not open. status = %d\n",status);
       exit(1);
     }
     printf("\n\n\nEnter name of output .dat file  ");
     scanf("%s",fnarr);
     if((fpp=fopen(fnarr,"w"))==NULL)  /* create it */
       printf("error opening %s\n",fnarr);
     printf("\n\nHow many minutes between addresses? ",nummin);
     scanf("%d",&nummin);
     printf("\n\n Waiting for the 00 second");
     printf("\n Then sending an address and offload command every %d minutes", nummin
);
     printf("\n Type a 'Q' to end.\n");
     exit_=FALSE;
     inittime();
     while(!exit_)
     {
        altflag=!altflag;
          sendcmd(altflag);
          getdata();
          waitmin();
       }
       fclose(fpp);


}

/* Wait for second to be 00 and initialize 'now' */

void inittime(void)
{
  int    sec;
  char   second[2],minute[3];
  char   *timeptr;
  struct tm *curtime;
  time_t bintime;

  time(&bintime);   /* time in seconds since midnite 1/1/70 GMT */
  curtime=localtime(&bintime);     /* convert to local time */
  timeptr = 11+(asctime(curtime)); /* assign ptr to pt at hour */
  strncpy(second,timeptr+6,2);
  while((sec=atoi(second))!=0)
  {
      time(&bintime);   /* time in seconds since midnite 1/1/70 GMT */
      curtime=localtime(&bintime);     /* convert to local time */
      timeptr = 11+(asctime(curtime)); /* assign ptr to pt at hour */
      strncpy(minute,timeptr+3,2);
      strncpy(second,timeptr+6,2);
      chhit = 0;
      if(_bios_keybrd(_KEYBRD_READY))
      {
        chhit=_bios_keybrd(_KEYBRD_READ)&0xFF;
        if(((char)chhit=='q')||((char)chhit=='Q'))
        {
          exit_ = TRUE;
          second[0]='0';
          second[1]='0';
        }
      }

  }
  now=atoi(minute);
}


void waitmin(void)
{
  int    yet;
  char   minute[3];
  char   *timeptr;
  struct tm *curtime;
  time_t bintime;

  past = now;
  if(nummin==60)
     flag60 = FALSE;
  else
     flag60 = TRUE;
```

```
 yet = FALSE;
 while(!yet)
 {
   time(&bintime);   /* time in seconds since midnite 1/1/70 GMT */
   curtime=localtime(&bintime);     /* convert to local time */
   timeptr = 11+(asctime(curtime)); /* assign ptr to pt at hour */
   strncpy(minute,timeptr+3,2);
   now=atoi(minute);
   if(((now<nummin)&&((now+60-past)==nummin)&&(flag60))||((now-past)==nummin))
       yet = TRUE;   /* nummin minutes has passed */
   if(now!=past)
       flag60 = TRUE;
   chhit = 0;
   if(_bios_keybrd(_KEYBRD_READY))
       chhit=_bios_keybrd(_KEYBRD_READ)&0xFF;
   if(((char)chhit=='q')||((char)chhit=='Q'))
   {
         exit_ = TRUE;
         yet = TRUE;
   }


 }

}

void sendcmd(int flag)
{
    st=-1;
    while(st)
        st=(asiputc(COM1,'#'));
    st=-1;
    while(st)
        st=(asiputc(COM1,'4'));
    st=-1;
    if(flag)
      while(st)
          st=(asiputc(COM1,'0'));
    else
      while(st)
          st=(asiputc(COM1,'1'));
    st=-1;
    while(st)
        st=(asiputc(COM1,'R'));

}

void getdata(void)
{
    int c,n,cont;

        printf("\n");
        for(n=1;n<=38;n++)
        {
           cont=TRUE;
           while(cont)
           {
              c=asigetc(COM1);
```

```
              if(c>=ASSUCCESS)
              {
                 putc(c,stdout);
                 putc(c,fpp);
                 cont=FALSE;
              }
           }
        }
        fprintf(fpp,"\n");
}
```

D. Technical information

The layout of the principal DPM board components, including the specially made DPM component carriers, is shown in Figure 11. The DPM board schematic is shown in Figure 12. DPM mechanical and electrical specifications are provided in Table 1. Connector specifications for the DPM and cable specifications for the DPM to ADCM interconnection are given in Table 2. A parts list is provided in Table 3.

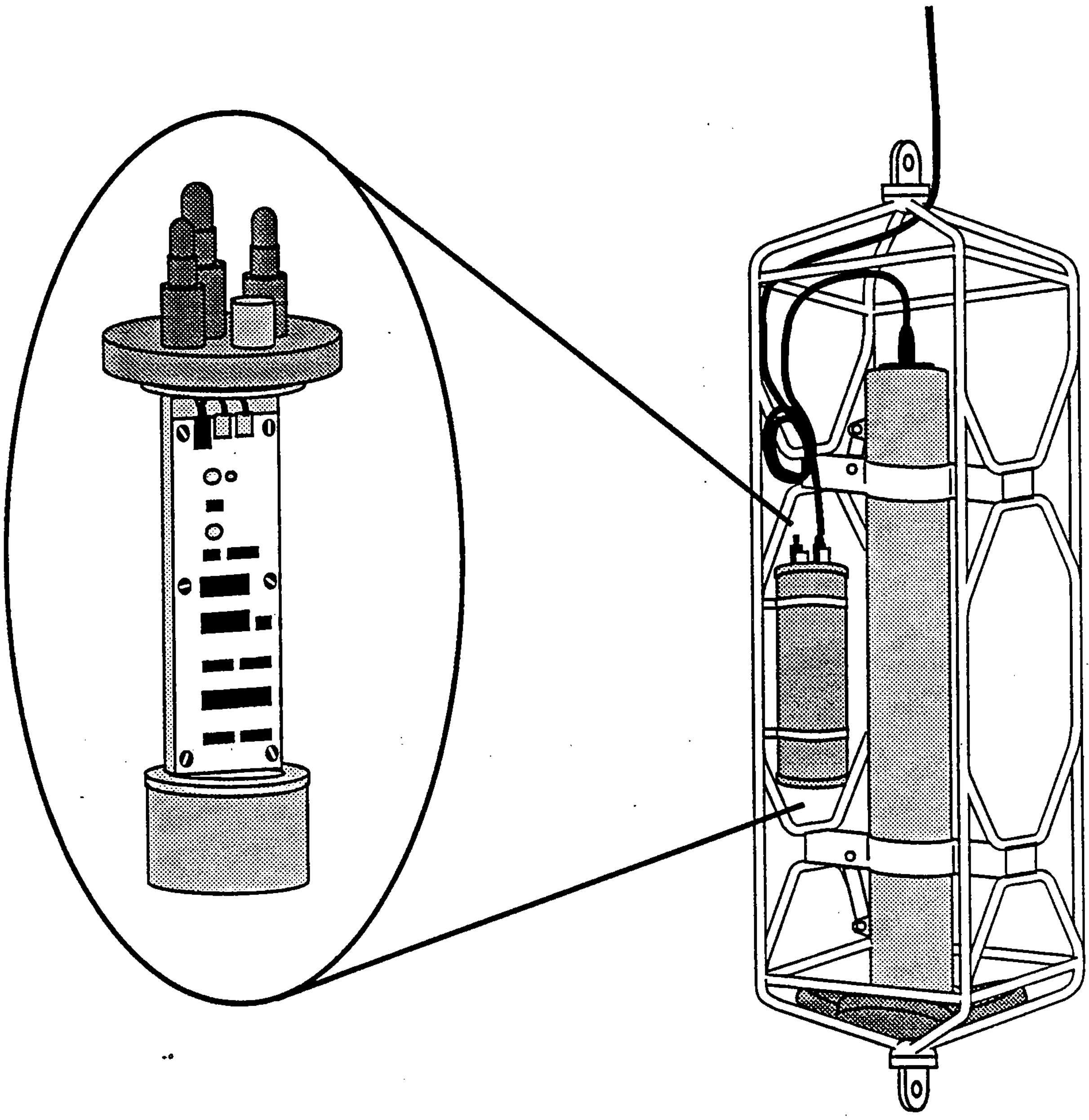


Figure 1: The Data Processing Module (DPM) is a self-powered unit in its own pressure case designed to be deployed along with an RD Instruments Acoustic Doppler Current Meter (ADCM). The figure shows a typical deployment configuration with the DPM clamped onto the ADCM load cage. Inside the DPM pressure case is a single-board electronics package and two battery packs (inset). The DPM serves as an interface between the ADCM and a satellite telemetry controller.

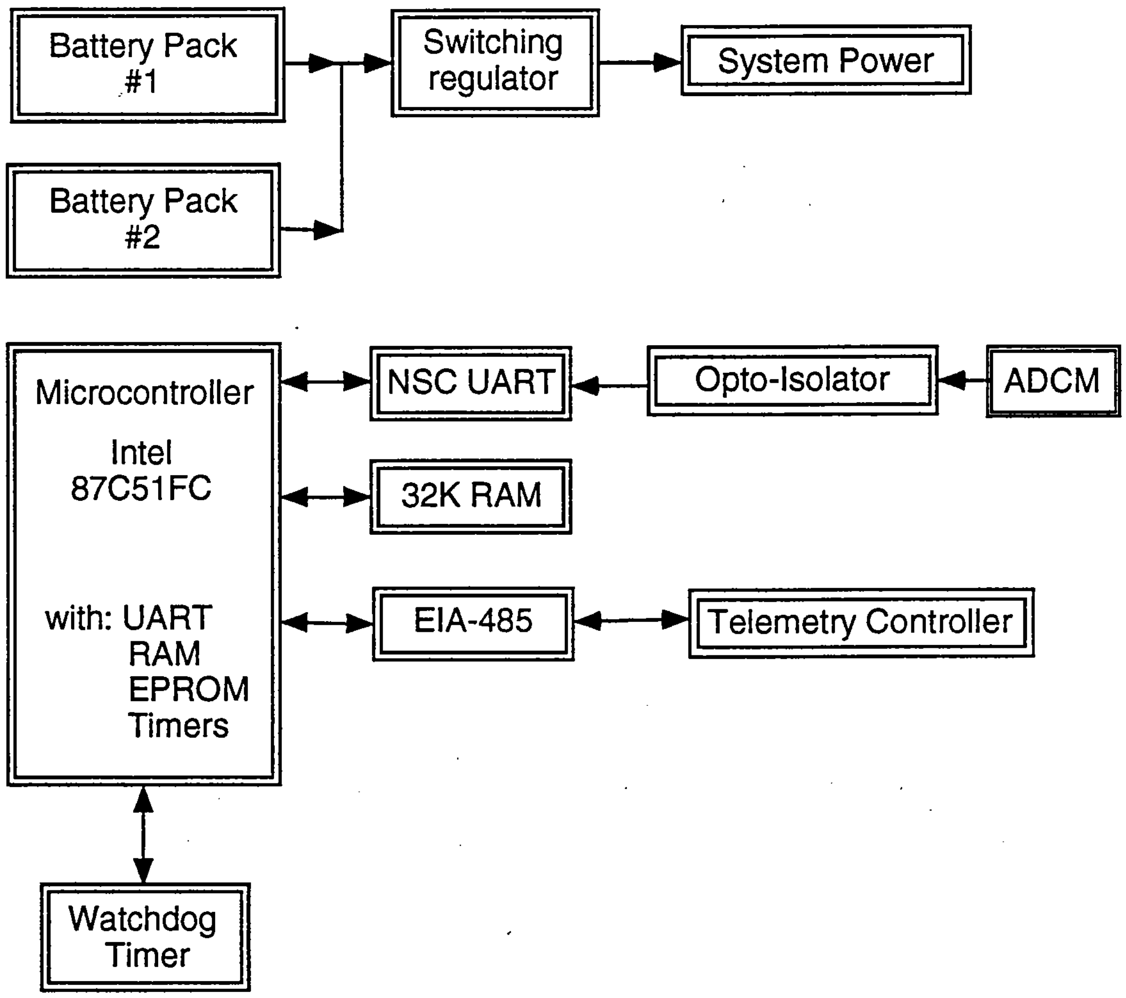


Figure 2: A block diagram of the principal DPM hardware components is shown. The power system consists of two battery packs and a switching regulator. The heart of the electronics is an Intel 87C51FC microcontroller with an onboard UART, 256 bytes of RAM, 32 kbytes of EPROM, and three 16-bit timers. Additional memory is provided by an external 32 kbyte RAM chip. The onboard UART is used for EIA-485 communications to the telemetry controller while an external UART talks to the ADCM through an opto-isolator. A watchdog timer circuit is used to reset the microcontroller in the event of software or communication errors.

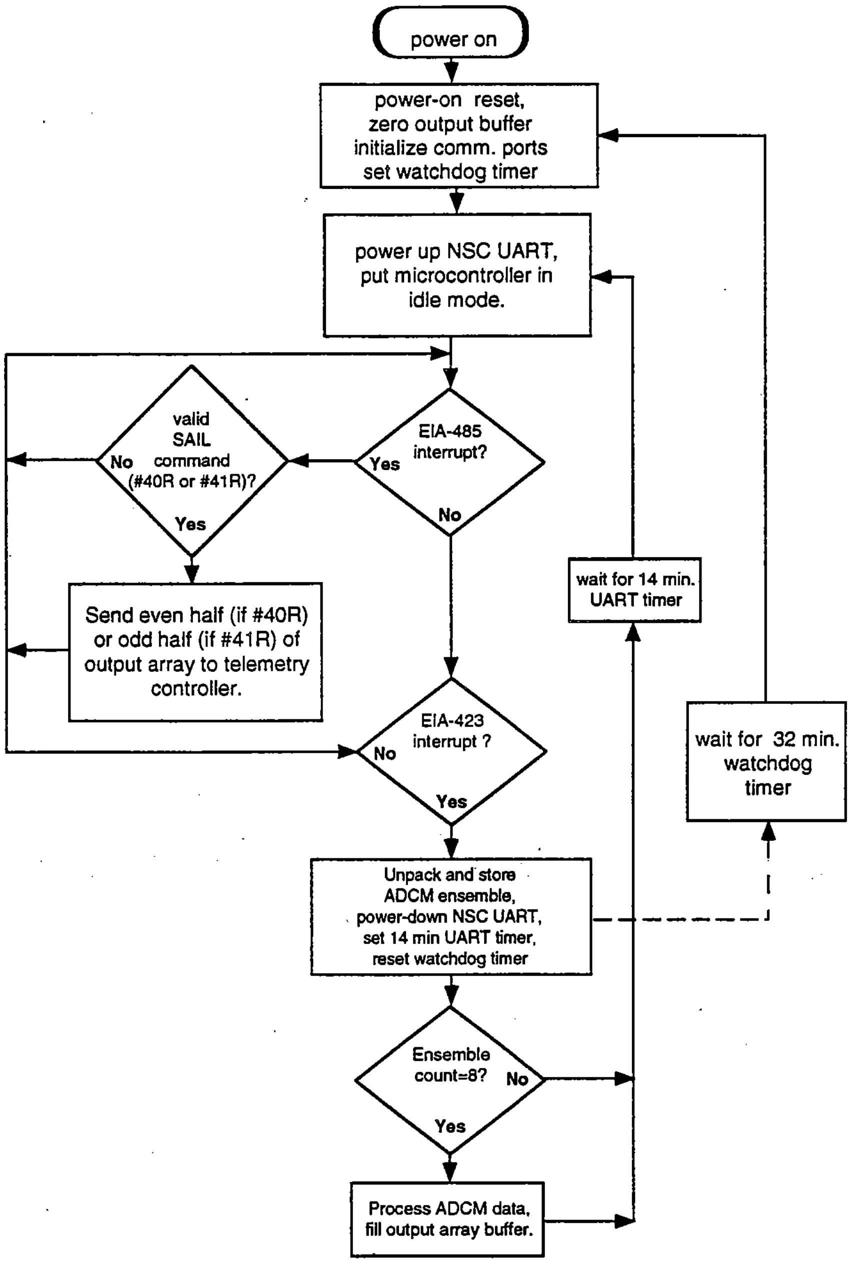

power on
power-on reset, zero output buffer initialize comm. ports set watchdog timer
power up NSC UART, put microcontroller in idle mode.
valid SAIL command (#40R or #41R)?
No
Yes
EIA-485 interrupt?
Yes
No
Send even half (if #40R) or odd half (if #41R) of output array to telemetry controller.
wait for 14 min. UART timer
EIA-423 interrupt ?
No
Yes
wait for 32 min. watchdog timer
Unpack and store ADCM ensemble, power-down NSC UART, set 14 min UART timer, reset watchdog timer
Ensemble count=8?
No
Yes
Process ADCM data, fill output array buffer.

Figure 3: The main control loops of the DPM processing program and the response to communication interrupts are shown in a flow chart. After initialization, the DPM waits for either an EIA-485 interrupt from the telemetry controller or an EIA-423 interrupt from the ADCM. A SAIL data offload command received on the EIA-485 channel initiates the data offload sequence. A valid data stream received through the EIA-423 channel initiates the processing sequence. DPM communication and control are described in more detail in the text.

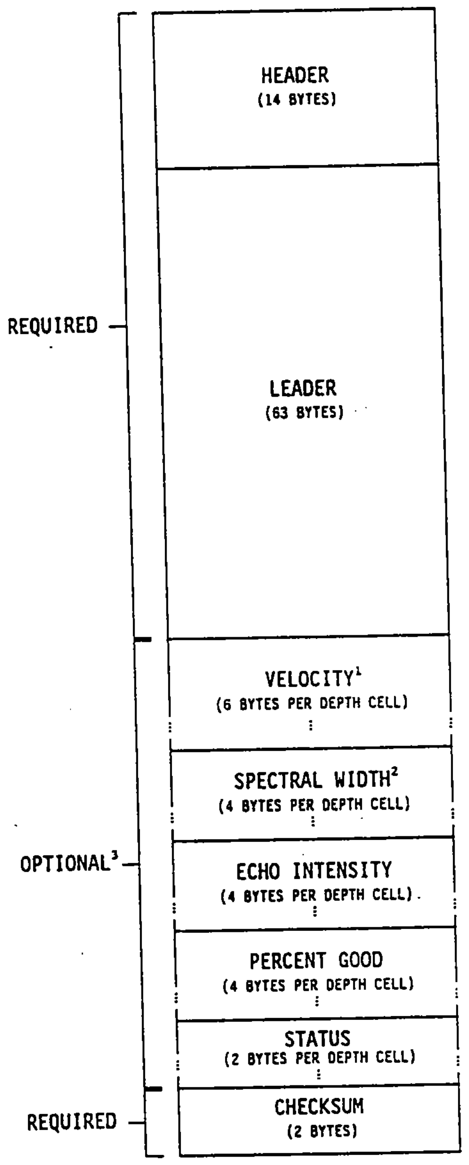


Figure 4: A schematic diagram shows the packed binary data stream transmitted through the ADCM serial I/O connector for each ensemble. The data stream consists of a header, a leader, up to four data arrays, and a checksum. For the implementation of the DPM on the Ice-Ocean Environmental Buoy (IOEB), the data stream is 719 bytes long and the data arrays selected are velocity, echo intensity, percent good, and status. The DPM decodes the variables from each ensemble and stores them in RAM. After eight ensembles have been accumulated, the processing sequence is initiated.

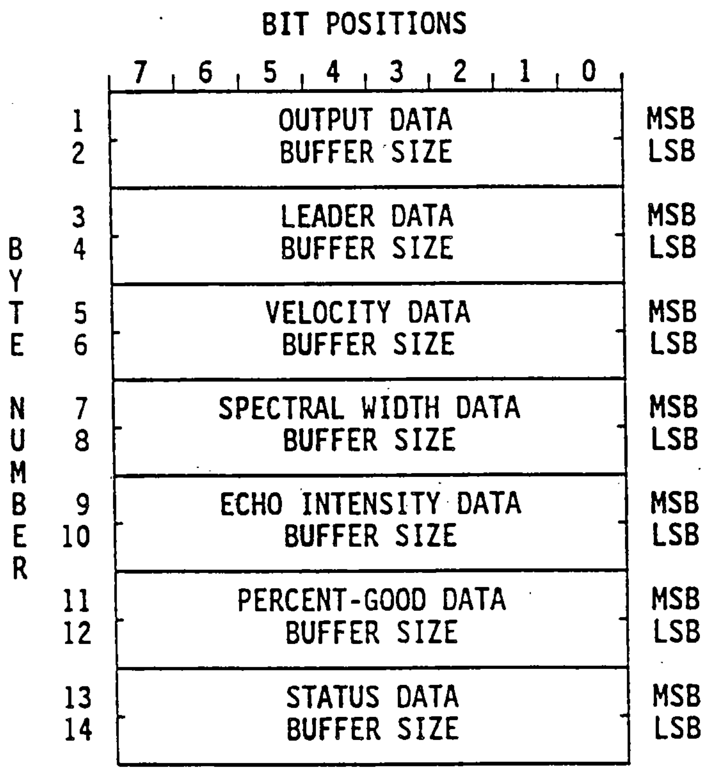

BIT POSITIONS
7 6 5 4 3 2 1 0
BYTE NUMBER
1 2 OUTPUT DATA BUFFER SIZE MSB LSB
3 4 LEADER DATA BUFFER SIZE MSB LSB
5 6 VELOCITY DATA BUFFER SIZE MSB LSB
7 8 SPECTRAL WIDTH DATA BUFFER SIZE MSB LSB
9 10 ECHO INTENSITY DATA BUFFER SIZE MSB LSB
11 12 PERCENT-GOOD DATA BUFFER SIZE MSB LSB
13 14 STATUS DATA BUFFER SIZE MSB LSB

Figure 5: The contents of the ADCM header are shown. The data array sizes transmitted in the header are compared to the expected array sizes based on the ADCM configuration. Since the array sizes are fixed after the initial configuration, this comparison serves as a check of the integrity of the incoming data stream.

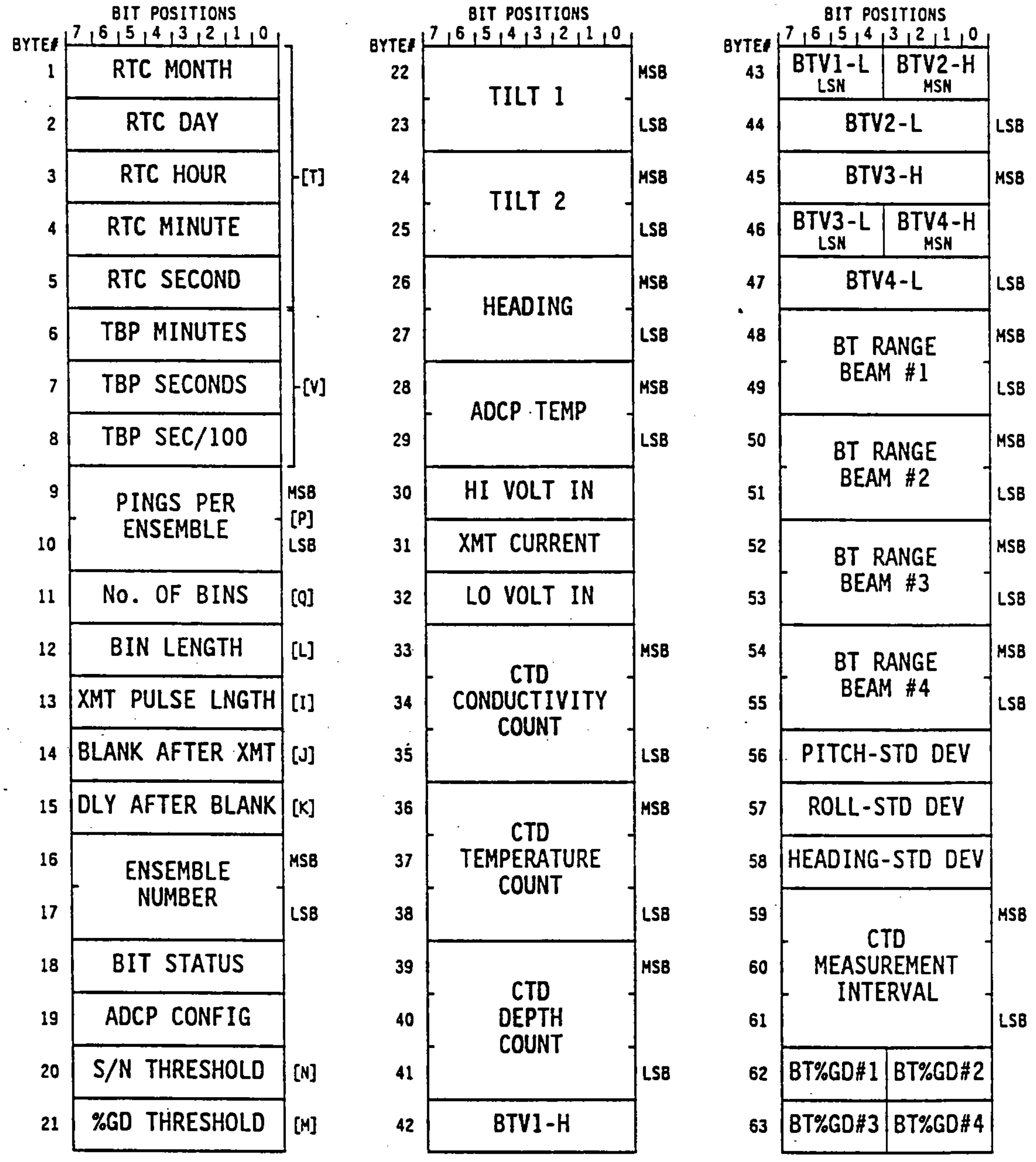


Figure 6: The contents of the ADCM leader are shown. All leader data except that related to CTD sampling and bottom tracking (neither of which are implemented) are decoded and stored in RAM. Some data (e.g., number of bins, BIT status) are used in error checking. Other data (e.g., heading and tilt) are used during processing.

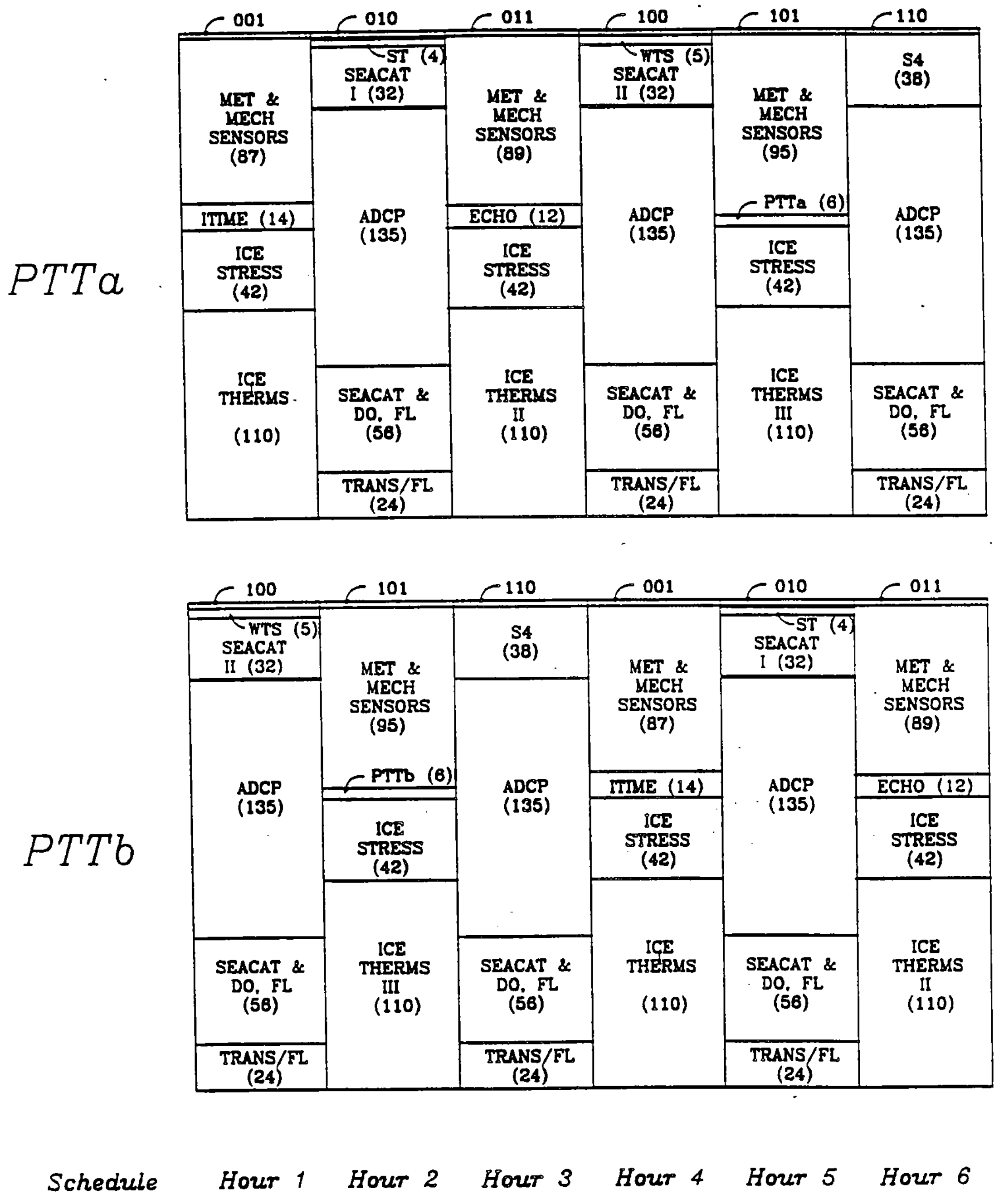


Figure 7: The transmission scheme for the IOEB Argos telemetry system is shown. Two PTTs are used to transmit data from various sensors. The two PTT controllers, each using a different SAIL address, interrogate the DPM at two hour intervals to request ADCM data. The DPM sends the even-bin data in response to one of the SAIL addresses, and the odd-bin data in response to the other. Since the two-hour PTT transmission intervals are staggered by one hour, the DPM is interrogated twice over a two hour interval (once by each PTT) and the full output array is transmitted in two halves.

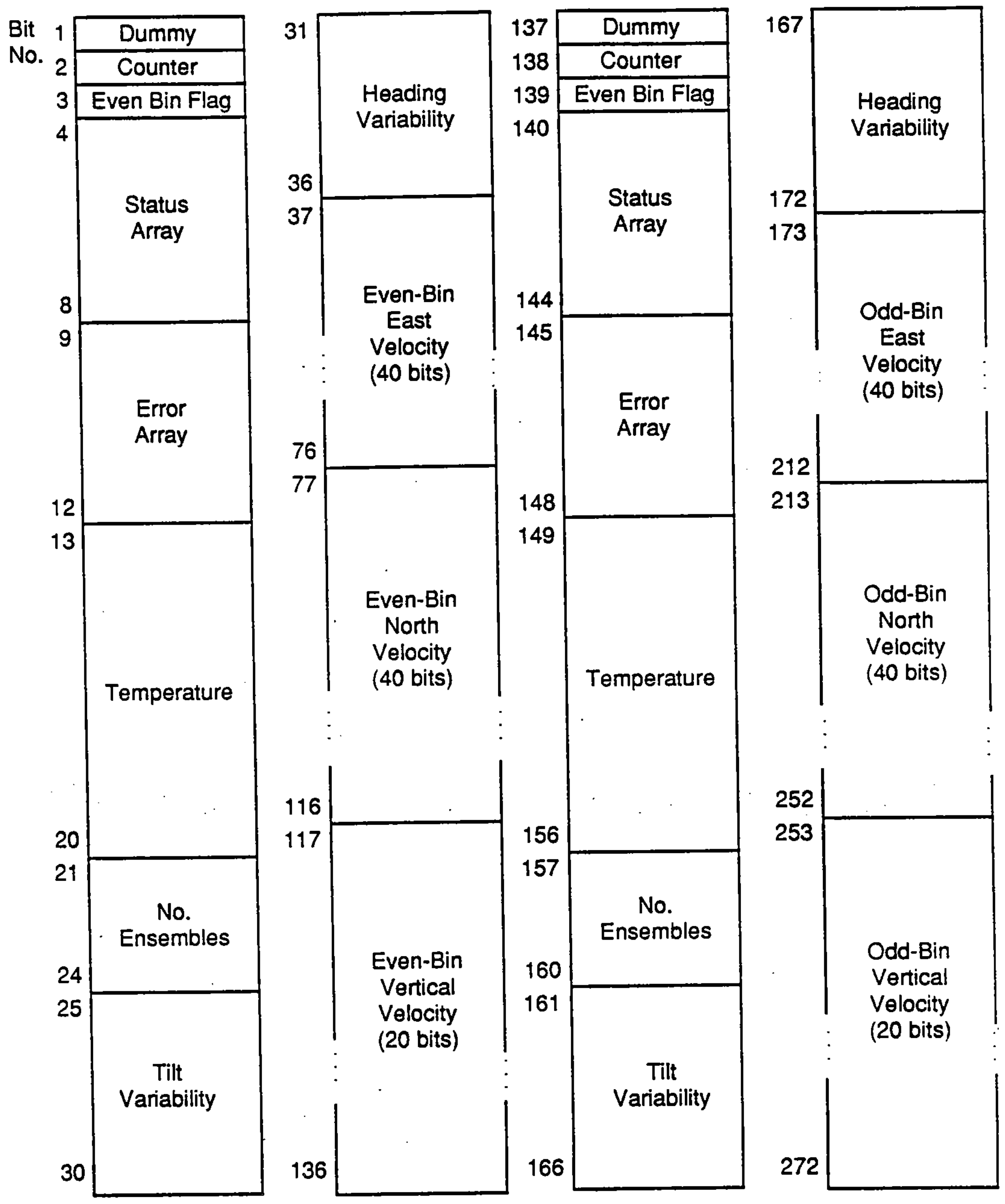


Figure 8: The contents of the DPM output array are shown. The output array is sent in two 136 bit halves in response to interrogation by two different PTT controllers (see Fig. 7). The dummy bit is stripped off by the telemetry controller to give a 135 bit sequence for transmission. The values of the error array, temperature, number of ensembles, tilt variability and heading variability are the same for both halves of the array.

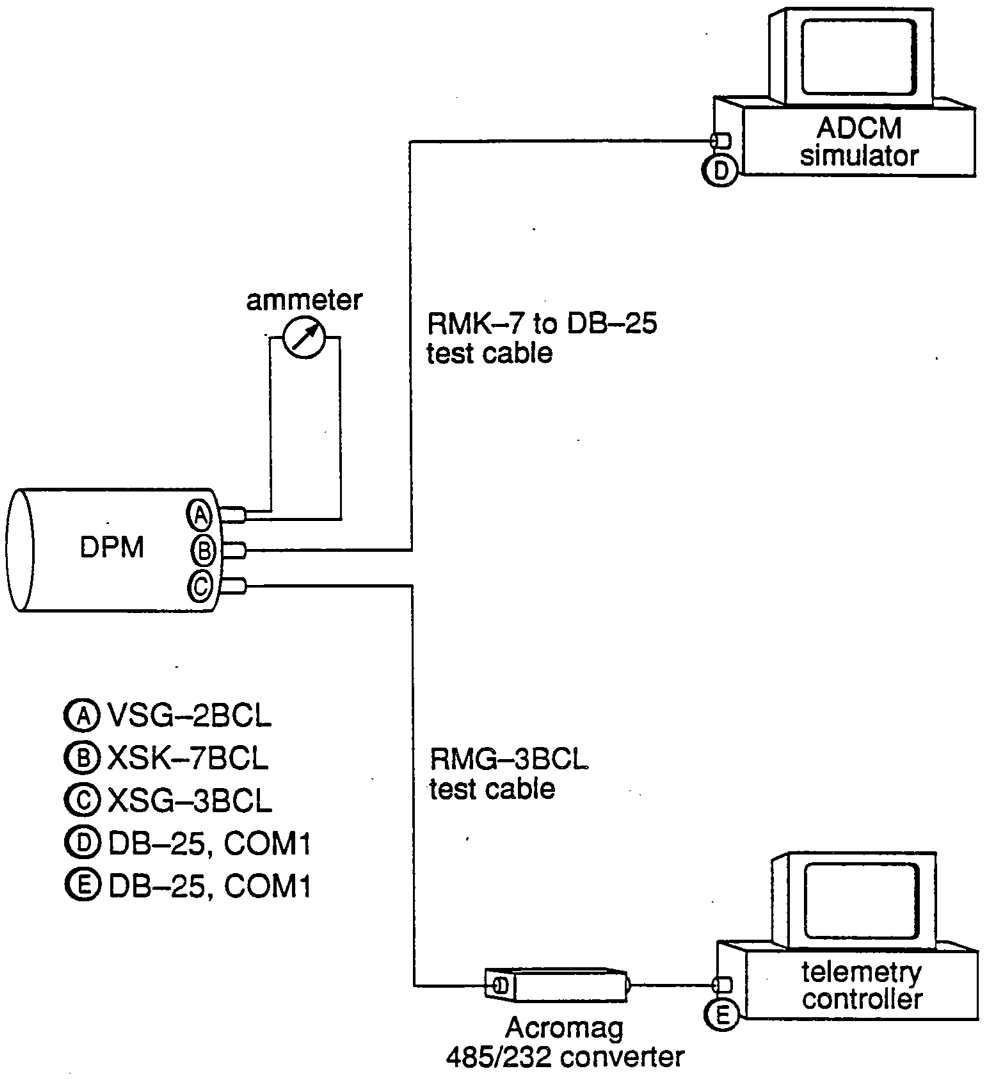
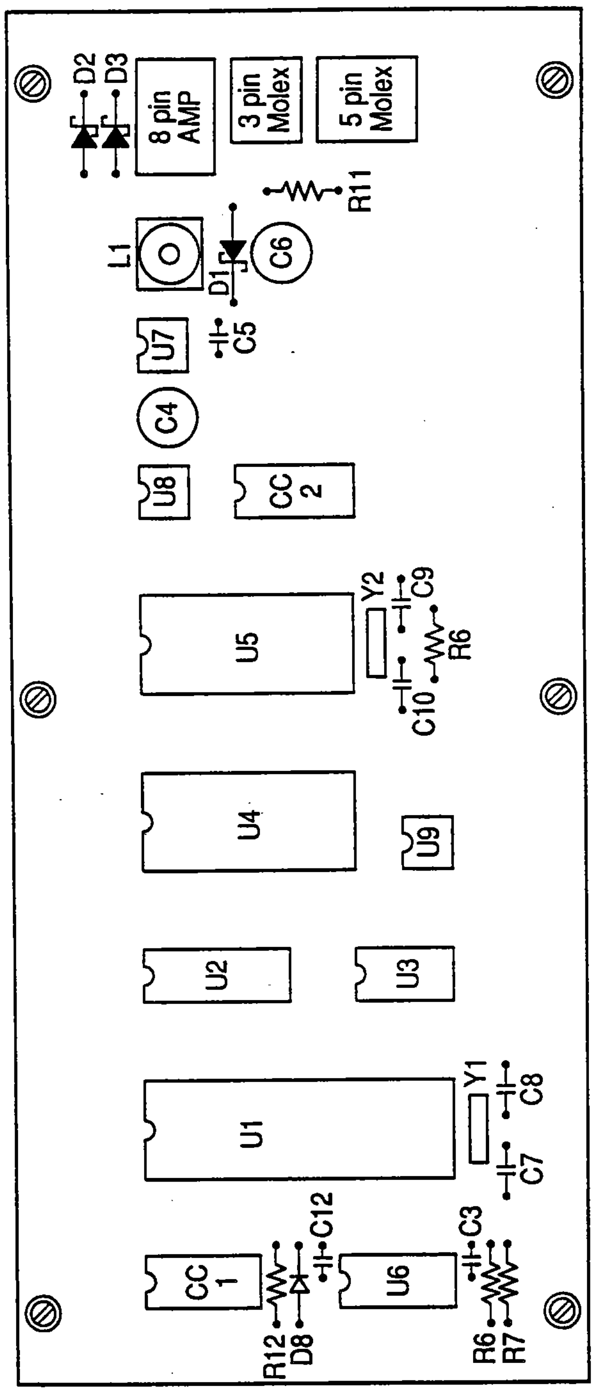


Figure 9: A schematic of the DPM test configuration, including two IBM compatible PCs, an ammeter, and various test cables, is shown. The ammeter replaces the DPM shorting plug and is used to check current draw by the UART and microcontroller. The PCs simulate the ADCM and telemetry controller. The ADCM simulator sends a sequence of data ensembles designed to test a variety of DPM error checking features to the DPM. The telemetry simulator interrogates the DPM and records the output. The output from a test run can be compared to the desired results to confirm proper operation.

```
40R006217101FCFF020000FCFF00FF0177776
41R206217101FE00020000FFFFFFFFFF77776
40R402218101FBFE0100FFFCFEFEFEFD77777
41R602218101FD010000FFFCFFFCFEFD77776
40R006216100FCFEFFFEFCFBFCFDFDFE87767
41R206216100FD00FFFFFDFCFCFEFCFE77776
40R406217101FBFDFFFCFDFBFD00FDFF87777
41R606217101FDFDFFFCFCFCFEFDFDFE77776
40R004330000000000000000000000077777
41R204330000000000000000000000077777
40R406217101FE010100FF0507070604877777
41R6062171010002010OFD060607060577766
40R002218101010406050406080907067777 6
41R202218101030506030308090807097776 7
40R40221810102070A080606090908057777 7
41R60221810105090A080708090808067777 7
40R00221810102090B090706090908067778 7
41R2022181010 70B0A0807090908080877777
40R40221820203 0B0909080608090806877777
41R602218202070A0A0A0908080907057777 7
40R002218101050A0A0A0904060606047777 7
41R202218101080A0A0A0906060504047777 7
40R402218101060708 0A090407050504777 77
41R602218101070809090807060605047777 6
40R00221820106070 90B0A030507050577777
41R2022182010708 0A0B0B040506040377777
40R402218101050607090A010405030377776
41R602218101070609 0A090304040402777 76
40R002218100020406070701030405047777 7
41R2022181000304060708030404040477667
40R402218100020304070803030405047777 7
41R602218100030407060703030404037777 6
40R002218101030002050604030303047777 7
41R202218101FF0104060502040404028777 6
40R402218101 00FEFF0302060304050287777 7
41R602218101FCFE01030204040404067777 6
```

Figure 10: The expected output arrays from a DPM test run using the configuration shown in Fig. 9 and the data file DPMCCS6.BIN as input to the OVERNITE.C program are shown. Each line represents the response of the DPM to an interrogation from the PC simulating the telemetry controller. Over a 36 hr interval the 144 ensembles in DPMCCS6.BIN are processed into 18 output arrays (there are 36 lines since the array is output one half at a time).



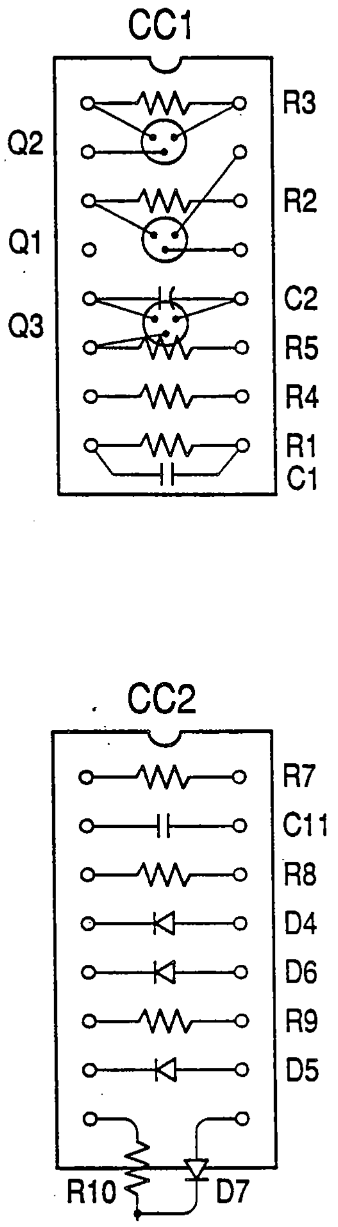


Figure 11: The layout of the DPM processor board is shown. Component identification can be made by referring to Figure 12 and Table 3. The layouts of the component carriers CC1 and CC2 are shown in detail.

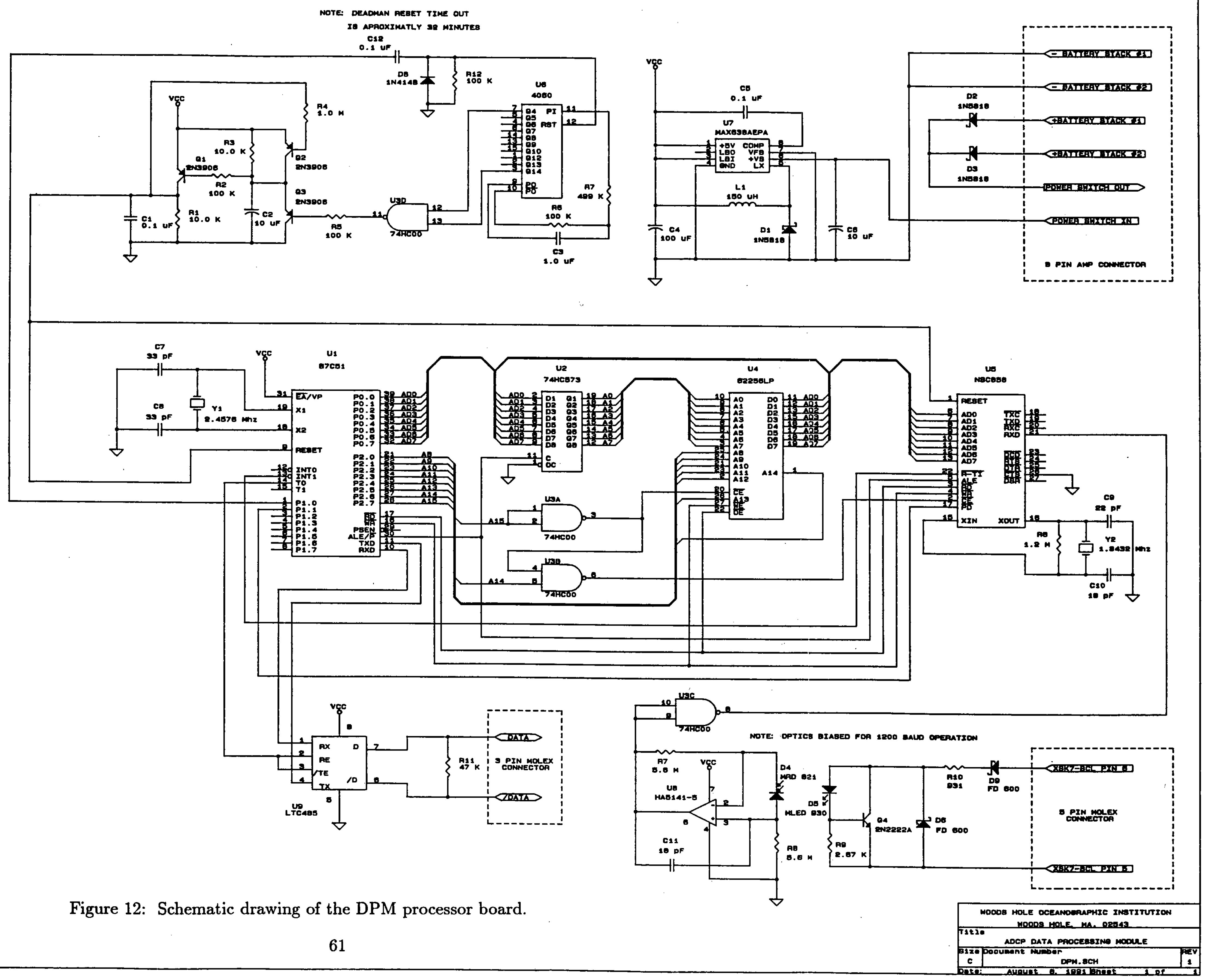


Figure 12: Schematic drawing of the DPM processor board.

Table 1: DPM specifications

12 July 1991

Mechanical:	
Housing Material	- 6061–T6 Aluminum Alloy
	- Hardcoated, Anode protected
Weight in air	- 13 kg
Weight in water	- 6.6 kg
*Length	- 50.5 cm
Diameter (end caps)	- 14.6 cm
(housing)	- 14 cm
Electrical penetrators	- 3
(VSG-2BCL)	- (1 each)
(XSG-3BCL)	- (1 each)
(XSK-7BCL)	- (1 each)
Pressure Rating	- 5000 db
Electrical:	
Avg. power consumption	- 15 mW
Battery capacity	- 28 Ah @ 10.5 VDC (Alkaline)
Controller	- Intel 87C51FC
EPROM (Internal)	- 32 k
RAM (Internal)	- 256 Bytes
(External)	- 32 k
COM. Ports	- 2
(EIA - 485)	- (1 each)
** (EIA - 423)	- (1 each)
Features:	- Watchdog Reset
	- Isolated EIA 423 Port
	- Addressable
	- Low power consumption
	- Environmentally tested from 50 to −30 deg. C

* Length with connectors mated, includes anodes
** Optically isolated, configured for Simplex operation

Table 2: DPM connector & cable specifications

Manufacturer	:	Brantner & Associates Inc. 1240 Vernon Way El Cajon, CA 92020-1874

DPM Connectors

Bulkhead Connectors	:	XSK-7BCL, 1 each (for EIA-423 port)
	:	XSL-3BCL, 1 each (for EIA-485 port)
	:	VSG-2BCL, 1 each (for power switch)
Dummy connector (for shipping)	:	RMK-7-FSD w/locking sleeve K-FSL-P
	:	RMG-3-FSD w/locking sleeve G-FSL-P
	:	VMG-2-FSD w/locking sleeve G-FSL-P
Shorting connector	:	Specified as VMG-2-FSD with Pins # 1 and 2 Electrically connected, used with locking sleeve P/N G-FSL-P

ADCP - DPM Interconnecting Cable Assembly

Cable Terminations	:	XSL-20CCP
	:	RMK-7FS (with locking sleeve p/n K-FLS-P)
Cable Length	:	2 meters
Cable material	:	18/7-SO (7 conductor, #18 AWG copper wire, rubber insulated, with neoprene outer jacket)
Pressure Rating	:	20,000 psi (mated)

Cable Wiring :

XSL-20CCP Pin#	XSK-7FS Pin#
2	7
4	6
5	5
13	4
14	3
15	2
16	1

Table 3: DPM parts list

ADCP DATA PROCESSING MODULE Revised: 12 July 1991
Bill of Materials

Item	Quantity	Reference	Part
1	3	C1,C5,C12	0.1 uF
2	2	C2,C6	10 uF
3	1	C3	1.0 uF
4	1	C4	100 uF
5	2	C7,C8	33 pF
6	1	C9	22 pF
7	2	C10,C11	18 pF
8	3	D1,D2,D3	1N5818
9	1	D4	MRD 821
10	1	D5	MLED 930
11	2	D6,D9	FD600
12	1	D8	1N4148
* 13	1	L1	150 uH
14	3	Q1,Q2,Q3	2N3906
15	1	Q4	2N2222A
16	2	R1,R3	10.0 K
17	4	R2,R5,R6,R12	100 K
18	1	R4	1.0 M
19	1	R6	1.2 M
20	1	R7	499 K
21	2	R7,R8	5.6 M
22	1	R9	2.67 K
23	1	R10	931
24	1	R11	47 K
25	1	U1	87C51FC
26	1	U2	74HC573
27	1	U3	74HC00
28	1	U4	HM62256LP-15
29	1	U5	NSC858N-4I
30	1	U6	74HC4060
31	1	U7	MAX638AEPA
32	1	U8	HA5141-5
33	1	U9	LTC485IJ8
34	1	Y1	2.4576 Mhz
35	1	Y2	1.8432 Mhz

* L1 was constructed by using 39 turns of #30 AWG enamel wire and a Magnetics Inc. P/N 1107CA100-3B7 ferrite core.

DOCUMENT LIBRARY

March 11, 1991

Distribution List for Technical Report Exchange

Attn: Stella Sanchez-Wade
Documents Section
Scripps Institution of Oceanography
Library, Mail Code C-075C
La Jolla, CA 92093

Hancock Library of Biology &
Oceanography
Alan Hancock Laboratory
University of Southern California
University Park
Los Angeles, CA 90089-0371

Gifts & Exchanges
Library
Bedford Institute of Oceanography
P.O. Box 1006
Dartmouth, NS, B2Y 4A2, CANADA

Office of the International
Ice Patrol
c/o Coast Guard R & D Center
Avery Point
Groton, CT 06340

NOAA/EDIS Miami Library Center
4301 Rickenbacker Causeway
Miami, FL 33149

Library
Skidaway Institute of Oceanography
P.O. Box 13687
Savannah, GA 31416

Institute of Geophysics
University of Hawaii
Library Room 252
2525 Correa Road
Honolulu, HI 96822

Marine Resources Information Center
Building E38-320
MIT
Cambridge, MA 02139

Library
Lamont-Doherty Geological
Observatory
Columbia University
Palisades, NY 10964

Library
Serials Department
Oregon State University
Corvallis, OR 97331

Pell Marine Science Library
University of Rhode Island
Narragansett Bay Campus
Narragansett, RI 02882

Working Collection
Texas A&M University
Dept. of Oceanography
College Station, TX 77843

Library
Virginia Institute of Marine Science
Gloucester Point, VA 23062

Fisheries-Oceanography Library
151 Oceanography Teaching Bldg.
University of Washington
Seattle, WA 98195

Library
R.S.M.A.S.
University of Miami
4600 Rickenbacker Causeway
Miami, FL 33149

Maury Oceanographic Library
Naval Oceanographic Office
Stennis Space Center
NSTL, MS 39522-5001

Marine Sciences Collection
Mayaguez Campus Library
University of Puerto Rico
Mayaguez, Puerto Rico 00708

Library
Institute of Oceanographic Sciences
Deacon Laboratory
Wormley, Godalming
Surrey GU8 5UB
UNITED KINGDOM

The Librarian
CSIRO Marine Laboratories
G.P.O. Box 1538
Hobart, Tasmania
AUSTRALIA 7001

Library
Proudman Oceanographic Laboratory
Bidston Observatory
Birkenhead
Merseyside L43 7 RA
UNITED KINGDOM

50272-101

REPORT DOCUMENTATION PAGE	1. REPORT NO. WHOI-92-05	2.	3. Recipient's Accession No.
4. Title and Subtitle A Data Processing Module for Acoustic Doppler Current Meters			5. Report Date January 1992
			6.
7. Author(s) Albert J. Plueddemann, Andrea L. Oien, Robin C. Singer, Stephen P. Smith			8. Performing Organization Rept. No. WHOI-92-05
9. Performing Organization Name and Address Woods Hole Oceanographic Institution Woods Hole, Massachusetts 02543			10. Project/Task/Work Unit No.
			11. Contract(C) or Grant(G) No. (C) N00014-89-J-1288 (G)
12. Sponsoring Organization Name and Address Office of Naval Research			13. Type of Report & Period Covered Technical Report
			14.

15. Supplementary Notes

This report should be cited as: Woods Hole Oceanog. Inst. Tech. Rept., WHOI-92-05.

16. Abstract (Limit: 200 words)

This report describes the development of a Data Processing Module (DPM) designed for use with an RD Instruments Acoustic Doppler Current Meter (ADCM). The DPM is a self-powered unit in its own pressure case and its use requires no modification to the current meter. The motivation for this work was the desire for real-time monitoring and data transmission from an ADCM deployed at a remote site. The DPM serves as an interface between the ADCM and a satellite telemetry package consisting of a controller, an Argos Platform Transmit Terminal, and an antenna. The DPM accepts the data stream from the ADCM, processes the data, and sends out the processed data upon request from the telemetry controller. The output of the ADCM is processed by eliminating unnecessary data, combining quality control information into a small number of summary parameters, and averaging the remaining data in depth and time. For the implementation described here, eight data records of 719 bytes each, output from the ADCM at 15 minute intervals, were processed and averaged over 2 hr intervals to produce a 34 byte output array.

17. Document Analysis a. Descriptors

satellite telemetry
Acoustic Doppler Current Profiler
Argos

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

18. Availability Statement Approved for public release; distribution unlimited.	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 71
	20. Security Class (This Page)	22. Price

(See ANSI-Z39.18) *See Instructions on Reverse* OPTIONAL FORM 272 (4-77) (Formerly NTIS-35) Department of Commerce