Louisiana Tech University
Louisiana Tech Digital Commons

Doctoral Dissertations Graduate School

Winter 1999

Sonar three-dimensional image formation for underwater vehicular collision avoidance

Gary Robert Boucher
Louisiana Tech University

Follow this and additional works at: https://digitalcommons.latech.edu/dissertations

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Boucher, Gary Robert, "" (1999). *Dissertation*. 748.
https://digitalcommons.latech.edu/dissertations/748

This Dissertation is brought to you for free and open access by the Graduate School at Louisiana Tech Digital Commons. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of Louisiana Tech Digital Commons. For more information, please contact digitalcommons@latech.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

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

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

UMI

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

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NOTE TO USERS

The original manuscript received by UMI contains pages with indistinct and/or slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SONAR 3-D IMAGE FORMATION FOR UNDERWATER VEHICULAR COLLISION AVOIDANCE

by

Gary Robert Boucher, B.S., M.S., M.S.

A Dissertation Presented In Partial Fulfillment
of the Requirements for the Degree
Doctor of Engineering

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

March 1999

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9918273

Copyright 1999 by
Boucher, Gary Robert

All rights reserved.

UMI Microform 9918273
Copyright 1999, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

2/18/99
Date

We hereby recommend that the thesis prepared under our supervision by Gary Robert Boucher entitled Sonar 3-D Image Formation For Underwater Vehicular Collision Avoidance be accepted in partial fulfillment of the requirements for the Degree of Doctor of Engineering

[illegible] (David Cowling)
Supervisor of Thesis Research

[illegible]
Head of Department

Electrical Engineering
Department

Recommendation concurred in:

Louis Roemer
Dr. Louis Roemer

[illegible]
Dr. Melvin Corley

James D. Lowther
Dr. James Lowther

Advisory Committee

Approved:
Richard J. Greechie
Director of Graduate Studies

Leslie K. Guice
Dean of the College

Approved:
Terry M. McConathy
Director of the Graduate School

GS Form 13
2/97

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

The last ten years have shown a marked increase in research into Autonomous Underwater Vehicles. A key component of this research is the ability to "look" ahead of the vehicle's projected path and translate active sonar returns into three-dimensional (3-D) computer data structures used to navigate and avoid obstacles. This need to avoid obstacles is also common to other underwater vehicles including submersibles.

Of special interest to this research is the "Occupancy Grid Framework." This technique divides the forward looking sonar field into cells. These cells can be maintained either in spherical or cartesian coordinate systems.

This research demonstrates a method of maintaining an array-type data structure based on the cartesian coordinates of returned sonar echoes. A volume of array elements are colored to reflect the probability of potential obstacles. Also, between sweeps of a scanning sonar transducer array, the locations of these volumes of probability are both rotated and translated inside the data structure as the vehicle turns and/or moves forward.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This research is different from prior research in two respects. First, rotation and translation of target probability spheres, located in a 3-D array, are accomplished by rotating and translating the sphere centers rather than the actual voxels comprising the spheres. In this way, probability spheres are continuously being removed, and regenerated at new locations inside the data structure. This relocation can be done in real-time as the vehicle moves.

Secondly, this research shows a method of pre-processing real-time data for increased speed using a series of two microcontrollers located between the sonar transceiver and the host computer, where the data is processed.

Programs used in this system consist of both assembly language programs for the microcontrollers, and a C language program for the host computer. These programs demonstrate a software approach which can be used as a basis for future research.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

I would like to think everyone who helped me on this dissertation. Also, much appreciation is extended to those individuals that aided me in the field testing phase of this sonar concept. These people include my wife Marie, Carl Hennigan, Philip Slay, and Decker Moore. Without their help it would not have been possible to obtain results from these trials.

I would also like to extend my appreciation to Steve Culp for the use of his lathe and milling machine. Without this type of equipment, it would not have been possible to construct the submersible components of this system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

INTRODUCTION

Area of Research

Concept Background

The last ten years have shown a marked increase in research efforts to develop Autonomous Underwater Vehicles (AUVs). A key component of this research is the ability for the vehicle to "look" ahead and translate incoming active sonar information into three-dimensional (3-D) image data to be stored in array-type data structures that can be processed by a computer. Of special interest is the recent research presenting the "Occupancy Grid Framework" [5]. This technique divides the forward-looking sonar field into cells.

The occupancy grid cell is a volume element (voxel) that is generated by sectioning space around the vehicle into spherical coordinates. Each of these cells may be either empty, or occupied by a return echo denoting an object at a certain distance and angle, both in horizontal and vertical angular coordinates [5].

The occupancy grid approach has been used to build bathymetric maps of sea floor structures, and has served as a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

general tool for AUV collision avoidance [2]. The use of the occupancy grid approach to sonar data gathering and processing for manned submersible operation did not appear in the literature.

By maintaining the most current occupancy grids possible the operation of the vehicle becomes safer. The risk of collision and, in the case of the manned submersible, the possibility of a fatality, is reduced.

Often water conditions and operating depth prevent a submersible pilot from being able to rely solely on visual perception for navigation to avoid collisions. Submersible operations in lakes sometimes require the pilot to descend through a low visibility layer existing from the surface down to the first thermocline, where water clarity may permit better visual navigation. In ocean operations submersibles often penetrate to a depth where sufficient light is not available. Also, some submersible operations occur at night. In both the fields of AUV and manned submersible operation, there is a clear need for better systems of sonar obstacle avoidance.

This dissertation examines a method of scanning a space above and below, and also left and right of the projected heading of the vehicle, using low-resolution low-cost sonar techniques to construct 3-D data structures within the memory of a host computer. Much of the research into the area of sonar 3-D image formation was done by P. G. Auran along with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a number of associates at various universities in Norway and Finland [2,3,4,5].

There are several approaches to constructing memory data structures for this type of application. Volume buffers, Octrees, and Binary Space-Partitioned Trees can be used [30]. This research uses the volume buffer method, which is simply a 3-D array of memory elements set to non-zero values to represent occupancy.

This research demonstrates a data structure which is actually an array of 3-D sonar echo information. This information can be utilized by future researchers through various methods to generate either true 3-D or quasi 3-D visual displays, allowing the viewer to "see" objects in the path of the submersible. Several innovative approaches to 3-D viewing are suggested by Tsao and Chen [31]. The major benefit to the AUV would be to have a 3-D cartesian coordinate array of stored data to construct a computer generated obstacle-free path.

Research in this area has yielded several techniques whereby occupancy grids are constructed and then translated into 3-D image arrays for building maps of sea floor features and also identifying obstructions ahead of AUVs. In the past, much of this information has been presented in maps drawn by large pen plotters showing navigation and bathymetry contours [4]. Various algorithmic techniques have also been developed for connecting grid information in a logical manner so as to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

fill in missing echo information and other grid cell connectivity operations [3].

One feature not reported in the literature is the ability to predict the current location of obstructions previously scanned as the submersible or AUV is translating forward and/or turning using 3-D array structures. Suppose that an underwater structure was located 16 degrees to the left of course level with the vehicle. This feature depicted either in the memory of an AUV or on a display of a manned submersible would, until the next scan refreshed the system, remain at the original 16 degree position as the craft turned to the left and speeded forward. Once refreshed the obstacle could suddenly appear in the direct path of, and much closer to the vehicle. Most conventional scanning sonar systems operate at less than 2 meters/second forward velocity [8].

Since the speed of sound in fresh water is 4,876 ft/sec (1,486 m/s) and the speed of sound in sea water is only slightly greater at 4,984 ft/sec (1,519) [13], a time limit is placed on how long it will take to complete a ping cycle. This is also affected by the maximum range of the sonar system. When more resolution is required and longer ranges needed, the scan time can be much longer [10]. It is easy to see how obstacles "painted" ahead of the vehicle can make a substantial change in position as a craft moves forward or turns.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A method of estimating changes in target position using linear algebra techniques could allow this object to be moved in a 3-D array of data as the vehicle moves. This is a logical extension of the research presented by Auran. Speed and rate of turn can be measured easily using standard techniques.

This research utilizes a Vector Model 2XG remote sensing compass module utilizing magneto-inductive magnetometer technology [22]. This compass module is interfaced to a Motorola MC68HC811E9 microcontroller to continuously report magnetic heading information to the host computer. In this manner any turning will be instantly noted by the computer.

A solution to the speed sensing problem was suggested with the use of a specially designed water velocity indicator. This indicator generates digital pulses as a magnetized wheel is rotated by a set of small turbine blades as water passes the sensor. The pulses are sampled with the same microcontroller used for heading information acquisition and both are presented to the host computer for processing. Thus heading and speed information can be used to facilitate changes in the host's data structure.

Outline of Research Method

Sonar Data Acquisition

A low resolution scanning sonar system is used to take measurements of the volumetric space ahead of the sonar

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system's path. Sonar "pings" are generated at 192KHz. The pings measure the range for each cell in a 6 by 6 array. Each array element will represent an 8-degree wide sonar cone (3dB down points). Six samples in both vertical and horizontal directions are taken to yield a total of 36 different angular distance measurements.

The sonar system developed has three transducers, each with an 8-degree cone projection pattern. These three sensors have been stacked one over the other looking 16 degrees apart in the vertical plane and placed so as to be rotatable in a left-right manner. The array can be rotated left or right of the projected path of the submersible or AUV by 20 degrees, thus giving a total of 48 degrees of horizontal viewing angle. This is a relatively narrow scan angle for looking ahead of a submersible or AUV. However, for the intended end-use application, this angle should prove adequate due to the slow turning rate of the specific submersible planned for the installation.

The major advantage to using transducers operating at 192KHz to transmit the sonar pings is their ability to transfer large amounts of energy in narrow beams with small physical size [6]. This is important in developing a multi-beam sonar system especially if scanning is used.

In scanning from left to right, the array stops rotation every 8 degrees and generate three pings, one ping for each of the three sensors. Returning information for each separate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sensor is transferred to an Analog Amplifier-Filter Module (AAFM) in the sonar pod. This submerged sonar pod contains the front-end stages of the electronic processing chain. Here the 192KHz analog signals representing returning echoes are amplified and highly filtered before being detected and sent to an analog to digital converter on an embedded microcontroller located inside the sonar pod.

The Sonar Pod Microcontroller Module (SPMM) has built-in 8-bit resolution A/D conversion capability where the returning analog signals are sensed and converted into a series of data elements, each representing a signal level of one byte [19]. Each ping will generate 300 such samples, with each sample representing a total time of flight for the sonar signal equal to one foot of range.

Each time a ping is generated a formatted packet of data is transmitted from the SPMM in the sonar pod to a pipeline of pre-processing stages for preliminary analysis. Once three sets of data are recorded (one for each transducer) representing the current left-right scan angle, the sonar transducers move to their next angular position and another set of data is taken. Once a scan from left to right is completed, the sonar assembly is rotated vertically around a horizontal axis some 8 degrees, returned to the leftmost position, and another sweep is begun.

The echo from each sonar ping will be broken down into 300 separate voxels, each representing a vertical and horizontal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

angular position along with the amplitude of each voxel sample.

Sonar Data Pre-processing

The pre-processing chain uses two Motorola MC68HC711E9 microcontrollers for processing the voxel information as it leaves the SPMM in the sonar pod and proceeds to the host computer. The first microcontroller, "processor A," thresholds each voxel data element with a pre-loaded template of memory information found in auxiliary RAM memory connected to this microcontroller. Information returned by the sonar will be compared against this threshold to see if there is a leading edge of an echo occurring. A simple analog thresholding technique was used by Auran and Malvig to identify returning echoes [2]. If an over-the-threshold echo is detected, processor A identifies it as an "event." Event information is in-turn sent to a second microcontroller in the pre-processor chain "processor B."

Processor B has at its disposal a look-up table in EPROM memory where the address of the table is a binary number composed of X angle, Y angle, and range information. The returned data from this EPROM look-up table gives pre-calculated numerical values for the echo edge in cartesian coordinates (X,Y,Z). For amplitudes of returning echoes that are continuously above the threshold level, the software records a limited series of events spaced at a predetermined distance. This look-up table approach has the major advantage

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of being fast and thus relieves much of the computational burden on the host computer.

Speed and Turning Rate Measurement

It is essential to furnish not only angle and range information to the host computer but also information on vehicle speed and changes in vehicle direction. Without this information no estimating of target position between sweeps can be made.

The need for velocity measurement was solved by the use of a 3-inch diameter 7-blade turbine attached to a rotating magnet with 6 poles. As the water flows over the turbine outside of the submersible or AUV, the turbine blades rotate. The rotation is approximately proportional to the velocity, and thus the number of digital transitions per time period measured by a digital magnetic proximity sensor increases as velocity increases. The magnetic wheel and proximity sensor were removed from an LEI model ST-TBK speed and temperature instrument package sold as a boating accessory. The temperature function was not used.

One unique feature of this turbine speed sensor is the fact that since it must turn freely with almost no friction, ball bearings are required. The problem with standard steel ball bearings is that they must be sealed against the elements. Often they are packed with grease to permanently lubricate them. This causes friction that can have a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pronounced effect on the sensor. This design uses plastic non-lubricated bearings that are open to the water. The actual ball bearings themselves are constructed of glass.

The need for changes in heading to demonstrate left-right turning is accomplished with a digital compass module located outside of the sonar pod frame in the water at a position that is minimally affected by the hard iron magnetic properties of the sonar system and attached vehicle. The connection bracket was fabricated from aluminum and stainless steel to have no effect on the magnetic fields in the area of the compass.

Both the magnetic compass and speed sensor utilize the same microcontroller located inside the compass module. This controller is the same general type as the pre-processors, a Motorola MC68HC811E9. Signals from this microcontroller enter the host computer through a second serial port, separate from the pre-processor information described earlier.

Host Computer Function

Generated events and sensor information flow to the host computer, where they are received as coordinates for a cartesian based occupancy grid along with rates of translation and turning. Later, when the system is actually employed for collision avoidance on a submersible, a Pentium-based computer capable of fast numeric processing should be used along with perhaps the LINUX (UNIX) operating system.

For this research the effectiveness of the entire approach is studied on a non-real-time basis in a laboratory setting.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For this reason processor B and the sensors need only deliver their data to a recording host, a much slower Dell 386-SX computer, where each data element is time-stamped to indicate when it arrived and was recorded with all necessary information. This concept of recording data to be processed later is described by Spitzak, Caress, and Miller [28].

Once several runs of data are recorded, the data can be taken back to a laboratory and processed through a faster computer using host software that is almost identical to the end-use version. In this way the performance of the overall system can be evaluated.

This approach has several major advantages. First; it allows the researcher to stop the flow of data into the host at any time and take "cross sections" of the memory data structure for evaluation of system performance. Another advantage to non-real-time evaluation is the fact that the amount of computer equipment necessary at the testing site is greatly reduced. This can be especially true if the data is going to be used for testing graphics displays in the future. This approach also allows evaluation on Windows 95/98-based operating systems, which in practice may prove difficult to make portable and reliable for use in an actual AUV or submersible.

It is believed by the author that the best approach to a permanent on-board operating system for submarine or AUV applications is the LINUX (UNIX)-based operating system. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operating system was shown far superior to Microsoft Windows NT in recent tests [25]. A recent Internet article in the July 1998 publication "Cover Story" points to the strong preference of engineers for LINUX over Windows NT [26]. This approach was not implemented in this research, but could be used in future efforts with little or no modification to the source programs.

Although as mentioned earlier, the host computer employed in this research is less "embedded" than the final version to be installed in the end application, the programs and their functions would be virtually identical. The interim "recording host" concept is useful in debugging the host programs since the gathering of data and processing of data can be done at two different times.

As mentioned earlier, the host computer is fed events in their (X,Y,Z) coordinates along with associated range information. The longitudinal axis of the vehicle will be pointed in the direction of the positive X-axis. Left and right will be represented by the Y-axis, and the vertical direction will be represented by the Z-axis. A list of events will be maintained in the host computer in this manner. Each event will also be time-stamped to indicate when it was received. Since events represent obstacles, new events will enter an event table in memory even though they may represent the same echo that occurred on a previous scan, generating a previous event.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As the vehicle proceeds forward and/or turns, the older entries in the event table need to be updated. Since the events are recorded in the separate event table, their coordinate points can be both rotated and translated with standard linear algebra techniques [14].

Rotation would be required when the vehicle turns left or right or pitches up or down. Translation would be required when the vehicle moves forward or backwards through the water (along the X-axis). This research will limit these possibilities to only forward motion and turns left or right at zero pitch angles. These would be the predominant movements observed by either an AUV or submersible. It would be a relatively easy task to extend pitch capability to the system if appropriate sensing is provided.

The host computer maintains a 3-D array of memory locations each represented by an integer value set to zero at initialization. Events represented by table coordinates update this 3-D spatial representation inside the host computer. As new events are generated, they are mapped into the array as a cluster of elements with the center of the cluster at the tabled coordinates of that event. As events move through translation and rotation, the array will have to reflect the changes.

If an array element has no probability of representing an obstacle, the value of the element is zero at that location. When an event is processed from the event table, a cluster of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

array elements will be "colored" by adding a fixed number to each element. Viewing the array as 3-D information, the cluster of now non-zero elements is centered at the X, Y, and Z location of the event and is composed of a quasi-spherical geometry.

The size of each sphere in radius will be related to the range from which the echo returned. For this reason, processor A also furnishes additional range information to the host. Close echoes are represented by small spherical volumes, and distant echoes have much larger volumes and correspondingly more uncertainty. Machine vision techniques offer one example of single-element numerical tagging using a procedure called "blob coloring". Often each element in a 2-D display is represented not as a bit but as a full byte of data capable of holding a small integer number.

As events are generated, and also after events are translated or rotated, a new sphere cluster is generated with all elements tagged with necessary information. Spheres can be removed by simply subtracting from each array element in the sphere a total of the original amount added to it for creation. Most of the time events will generate many overlapping elements in this 3-D array structure. Overlapping elements will have higher values than non-overlapping ones. Both will be identified as a target based solely on the non-zero status of the element.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The advantage of this approach is that only the event centers have to be translated and rotated and not array elements. The method used to generate the cluster spheres is quite simple and can be done quickly with look-up table techniques. No algorithm was used for formation or removal of sphere clusters. Samet suggests various methods for locating adjacent nodes in an array by use of tree techniques [24].

Old events will continue to be represented for a period of several seconds and then be removed based on age. Removal will erase both the event from the event table and also its effect on the 3-D grid elements attributed to the event. When events are removed from the event table, the value for each sphere element added in its last generation or regeneration will be subtracted. Thus, if an element is overlapping, subtraction will reduce the number present but it will not return it to zero, the non-occupied state.

The reason for maintaining events past the subsequent scan is that because of the uncertainty of sonar returns, valid data may sometimes be missing [4]. It would be safer to assume that the target is there and not being seen than erasing it prematurely on the next subsequent scan.

Many of the techniques used to link generated echoes into groups of cells can be employed on the final computer grid structure. Missing cells can be dealt with in ways similar to methods described by Auran et. al. [2,3,4,5]. A marked increase in computer speed for the typical PC has been seen

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

over the last two or three years since most of these articles have been published. Computer speed should not be a problem for implementing the described system in its end-use application, especially since the generation of event information is done in a pipeline manner using pre-processing.

The product of this proposed research is a method for continuously updating an array type data structure of 3-D information containing both estimated target position and currently scanned target position. It will be up to the reader to utilize this array in a way suited to the type of underwater operation planned.

Data Gathering and Analysis

After the sonar system was constructed, data was recorded with a computer. This recorded test data was the standard output of processor B as would normally be transferred to the host. A recording host computer temporarily takes the place of the host as an interim step in the sonar evaluation. Test data can be gathered once and replayed many times to process events as they affect the memory array elements.

Real-time display of data on a visual screen requires a capture mechanism to depict the image as it appeared when displayed. Using this method, data can be maintained in the data array and frozen in time by interrupting the process and taking "slices" of the array elements to show the estimated position of rotated and translated objects. The array can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

then be sliced later to show newly updated data for comparison. This can be a useful analytical tool.

Observing the speed of data manipulation with no "stop and look" data evaluations will assure the researcher that once installed on the actual submersible or AUV, the system will have sufficient capacity to handle the computational load and thus be practical for the application.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

HARDWARE SYSTEM DESIGN

Hardware System Overview

General Considerations

The sonar equipment of this study was constructed with the intent to use it later as a functional sonar system including visual display for the one-person submersible "The Vindicator." Construction of this submersible was finished in the spring of 1997 by the author. With this intended use in mind these sonar modules can be used both in controlled laboratory experiments and later in an end-use application for further field testing of research concepts.

This sonar design operates at 192KHz, which is a relatively high frequency for most sonar signals. The higher the frequency, the greater the detail obtained. However, the down side for operation at this frequency is loss of range. The high-frequency loss phenomena has been known by the British since the 1920's [12].

Several related hardware modules were constructed. These modules fall into two categories. First, the outside modules were designed to withstand pressures to 200 feet of salt

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

water. These are basically systems that will be placed outside the hull of the submersible in later application. Much care was taken in making these components pressure-and corrosion-resistant.

The interior systems contain most of the computer and microcontroller processing capability. These systems did not have to be pressure-or water-resistant except for the occasional splash encountered in normal submersible operation. A through-hull interface is intended for later, but is not necessary for this research.

External Systems

The external systems are mounted both inside and onto a wedge shaped welded steel frame. This one-inch angle iron frame was custom fabricated to fit onto the bow of The Vindicator. This frame is used for protection of the sonar modules that must be exposed to the rough environment of submersible operation. The frame is designed to cradle a 6.5-inch diameter pressure vessel referred to as the Sonar Pod. This pod enclosure houses most of the non-computing electronic systems along with the Sonar Pod Microcontroller Module (SPMM).

At the end of the support frame, a pivotable and rotatable sonar array of 3 transducers is mounted to send and receive sonar pings to establish the occupancy grid. Similar sonar approaches are used in robotics to establish "evidence grids"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[32]. This array is pivoted by a pneumatic cylinder controlled by a 4-way electric solenoid air valve.

In actual practice, the use of a pneumatic valve of this type would result in constant off-gassing from a submersibles pneumatic system, or else the release of exhaust air into the submersible itself. Release of air in bursts could be annoying to the submersible pilot. Release of air into the submersible compartment is not tolerated as a safety hazard. Therefore, in practice the pneumatic cylinder will most likely be replaced with another method for sonar head positioning.

Rotation of the sonar array is done with a stepping motor and gear system housed inside of a 4.5-inch outside diameter PVC motor enclosure. The stepping motor rotates 1.8 degrees per step and must be geared to provide smaller angular movements per step while increasing available torque. This is done with the aid of two separate Delrin/brass gear combinations located inside the motor enclosure yielding a 4:1 gear reduction ratio. Outside of the enclosure, the output shaft is linked via two nylon sprockets and stainless steel chain to the actual sonar array, giving another 4:1 reduction ratio. Thus, the gear system and sprocket/chain reduction ratio is 16:1, yielding a 0.1125 degree movement in the sonar head for each step of the motor.

The sonar pod consists of six sub-systems. The pod's operation is controlled locally with the Motorola MC68HC711E9 microcontroller SPMM connected to the other modules and also

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to an RS-232C link back to the pre-processors and eventually the host computer.

The SPMM is responsible for controlling the Ping Generator Module (PGM) also located inside the pod. The rotation of the sonar array is controlled with the Scan Control Module (SCM), and the analog return signal is conditioned by the Analog Amplifier-Filter Module (AAFM), all under the control of the SPMM.

The sonar pod must also have a power supply for producing various voltages to the other modules as well as a high power ping amplifier module (PAM) used to generate the 192KHz pinging that drives the sonar transducer array. A multiplexing/demultiplexing scheme is required for vectoring the ping power and return signal to and from the selected transducer. This is done with small SPST socket mounted DIP relays located on the ping amplifier module.

Internal Systems

Hardware had to be developed to collect signals from the sonar pod and then process these signals before passing them on to the end-use application. Although the target signals from the sonar pod could have been directed straight to the host computer for processing, any pre-processing that can be done by hardware prior to feeding the pod information to the host would be beneficial in relieving the computational burden on the host computer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A pipeline-type computing approach was used to do as much pre-processing as possible external to the host computer before supplying the signals to the host. This pre-processing is done in two stages. First, the sonar pod returning echo signals are fed to processor A, a MC68HC711E9 microcontroller, where they are collected in a 32K by 8 RAM memory. These signals are already formatted into ASCII strings by the SPMM before the sonar pod sends them to processor A in the chain. Here they are parsed and loaded into the RAM memory for evaluation.

The RAM memory that holds the returning echo information also holds pre-loaded threshold information that is used in checking to see if any returning echo exceeds a minimal threshold level and is thus considered an "event."

The two pre-processors, processors A and B, are housed together in a single enclosure the "interface enclosure." This enclosure links the sonar pod to the host computer, but also serves several other functions such as power supply, overcurrent protection, and pneumatic valve control.

Host Computer Specifications

Almost any moderately fast computer capable of input data buffering and support of either C or C++ language would meet the requirements for the host computer. Due to the fact that this research is being evaluated on a non-real-time basis the requirements can specify both a recording host and a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processing host. The recording host must have the ability to receive a continuous stream of input data from two communications ports simultaneously, buffer the information, and record the data to hard disk along with a time-stamp for synchronization. The requirements for the processing host computer for this evaluation dictate that the machine support C or C++ platforms, have enough memory to build the data structures, and have sufficient speed to facilitate user-friendly operation.

Sonar Pod Hardware Theory of Operation

Ping Generator Module

The Ping Generator Module (PGM) is responsible for generating the 192KHz ping signal, which is amplified and used to drive the sonar transducers. The objective of the PGM design was to produce a signal close in frequency to the designated operating frequency of the transducers that could be controlled in two respects. First, the modulated ping pulse width in respect to the total number of cycles of the 192KHz signal generated is variable under SPMM control. Second, the circuit must be triggerable from a pulse generated by the SPMM.

The output of the PGM consists of two separate signals. One is a continuous 192KHz square wave signal with 50 percent duty cycle. This signal only generates a ping output from the ping amplifier when enabled by a second signal from the PGM.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This enable signal is of a predefined width to allow only a specific number of cycles of ping to be sent to the transducers.

Figure 1 shows that the source of the 192KHz signal is a 5MHz crystal clock module capable of TTL/CMOS level output. This oscillator feeds a divide by 13 counter consisting of U1 and U12A. A further divide by 2 stage, U2A, provides an output at pin 12 that is a square wave with a frequency of 192,308 Hz. This 192KHz signal will be referred to as the system clock.

Initially the system clock was allowed to furnish its output to the ping amplifier module at all times waiting for an enable signal to allow the ping to exit the transducers. However, it was noticed that in the absence of a returning ping echo, the analog section of the sonar pod was reporting a base line signal of moderate strength. An AND gate, U10C, was placed between the 5MHz oscillator and U1 utilizing an enable signal from the SPMM line C4. After the ping has been generated, this enable goes low to prevent any 192KHz signal from being amplified other than that returning from the sonar target.

A strobe pulse is needed for triggering U5A that begins after the rising edge of the system clock, but finishes before the completion of the system clock cycle. For this purpose a dual-edge triggered monostable multivibrator U3, a 74HCT221, is used. The first stage of U3 is triggered on the rising

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

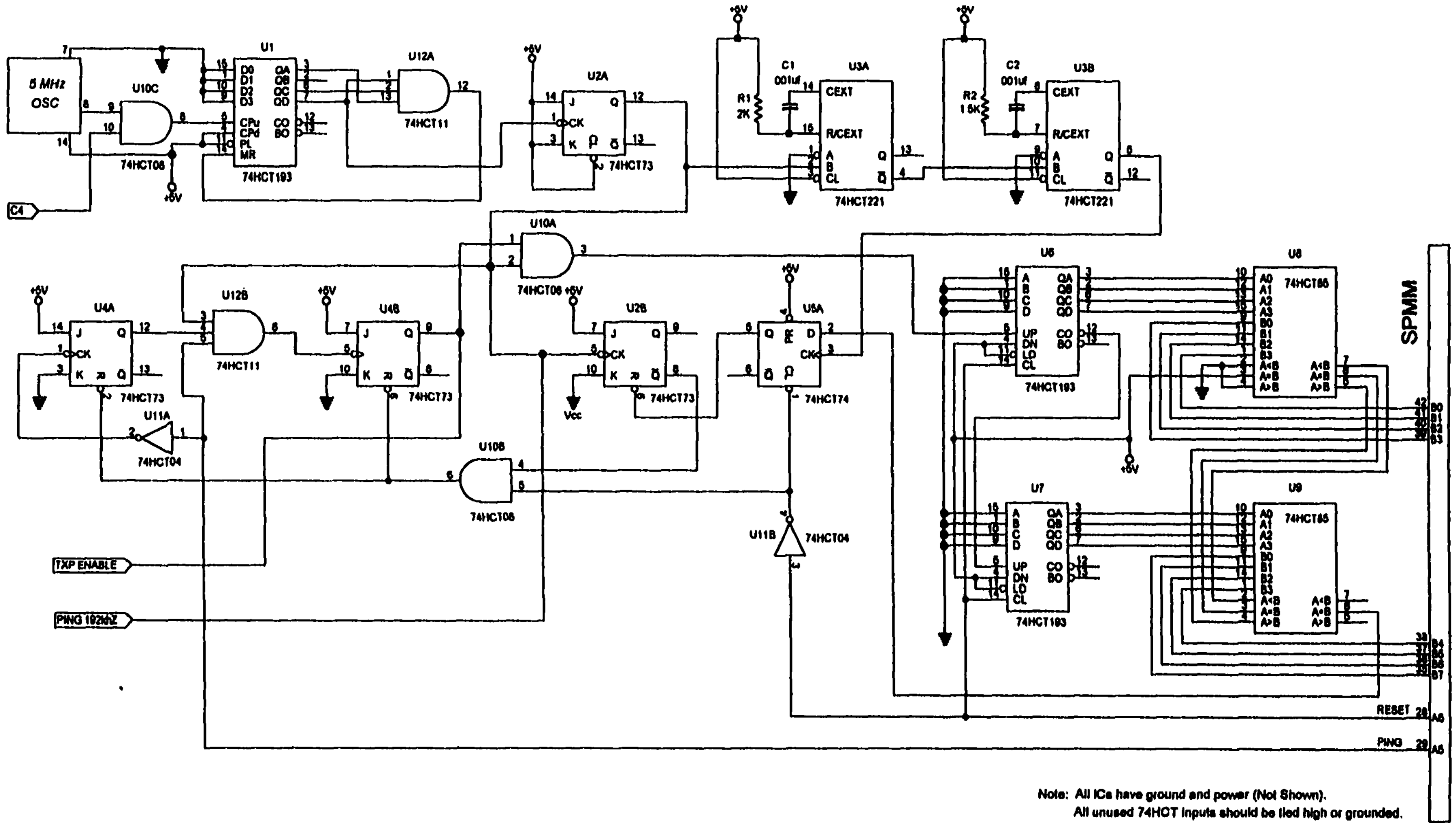


Figure 1

Ping Generator Module (PGM)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

edge of the input waveform from the system clock. When the output of U3A drops low, it triggers U3B, producing a pulse beginning after the system clock's rising edge but finishing before the completion of the systems clock cycle.

In the RESET condition (SPMM pin A6 high), the Q outputs of U4A, U4B, U2B, and U5A are cleared. This disables U12B's output at pin 6. Thus, the system clock signal cannot trigger U4B, making its Q output high. When the PING input is pulled high (SPMM A5), this signal travels through the inverter U11A and triggers U4A, producing a high output at Q. This output from U4A enables U12B's output at pin 6, triggering U4B the next time the system clock signal at U12B pin 3 goes from high to low. This output of U4B (pin 9) is the TXP-ENABLE used to enable the ping output.

When Q of U4B goes high, U10A becomes enabled via pin 1 and the system clock signal is fed to the U6-U7 counter chain, two 4-bit 74HCT193 counters. These counters start counting edges of the 192KHz signal until a count is reached that is equal to the SPMM's port B value. When this occurs the cascaded comparators U8 and U9 generate an A=B signal at pin 6 of U9 which causes the D input of U5A to become high. When the delayed pulse output of U3B goes high, U5A is triggered and the newly established D-equals-one input to U5 is transferred to the Q output of U5A. This high signal takes U2B out of its normal reset state so that on the next falling edge of the system clock signal at pin 5 of U2B, the not-Q

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

output at pin 8 of U2B will go low. This disables U10B at pin 4, causing U4A and U4B to be cleared, ending the ping cycle on the specific number of cycles dictated by the 8 bit SPMM port B output.

RESET (SPMM A6) is used to clear the U6-U7 counter chain and reset other devices to prepare for the next ping cycle. This circuit works well and allows variations in pulse width which inturn allows for variable energy pulses that can be balanced against resolution in sensing.

Ping Amplifier Module

The Ping Amplifier Module (PAM) serves two purposes. It contains the circuitry for ping power amplification prior to sending the ping signal to the transducers and also acts as a relay multiplexer to direct the ping and its return to the amplifiers. Figure 2 illustrates the schematic for the PAM.

The heart of the PAM is a custom-wound toroidial transformer, T1, driven by two TIP-120 power transistors (Darlington pairs), Q1 and Q2, in a push-pull configuration. The transformer has a primary center tap that is attached to the 12-volt source for the sonar pod through a 0.47 ohm 10-watt current-limiting resistor R1. With the collectors of these transistors connected to each end of the primary winding, and the emitters grounded, the transistors turn on, alternately producing a square wave current signal in the primary.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

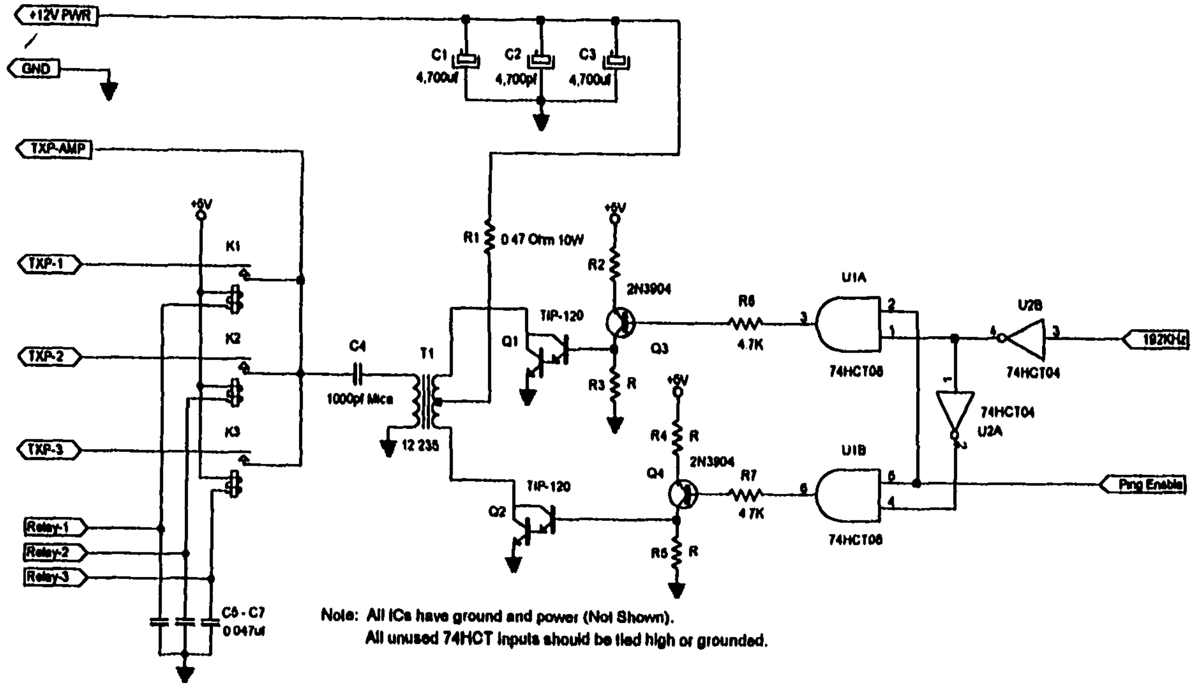


Figure 2

Ping Amplifier Module (PAM)

Due to the inductance of the transformer and the capacitance of the transducer, adjusted by C4, a series 1000pf silver-mica capacitor, the waveform produced is very nearly sinusoidal. No measurement of harmonic distortion was used. However, observation using a 100MHz bandwidth oscilloscope revealed no visible distortion. The 12 turns of the transformer primary and the 235 turns of the number 30 kynar insulated secondary yield pulses on the order of 200 volts peak to peak at the transducer input under load.

The TIP-120 transistors have a minimum h_{FE} gain of 1000 and a power-handling capability of 65 watts [23]. Even with such high current gain, these devices were not driven directly with the signal from the ping generator. Two 2N3904 general purpose NPN transistors, Q3 and Q4, were used in a pair of amplifier circuits to furnish the drive signal. These amplifiers were driven with two 74HCT08 AND gates, U1A and U1B. These gates used an inverter to facilitate having one gate on with the other off at all times as the 192KHz signal is applied from the PGM. The Ping Enable line from the PGM is connected to both AND gates so as to enable the 192KHz signal only when the ping signal is to be transmitted.

It is important to note that merely terminating the 192KHz drive without this enable/disable facility would cause one or the other of the two TIP-120 transistors to be on continuously, providing a high current DC path to ground through one half of the primary winding, and producing a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

nearly shorted circuit. The only current-limiting factor here would be the resistor R1, and the value of this resistor is 0.47 ohms.

Depending on the value of R1, pings can draw as much as 5 amps or more. This places a tremendous demand on the transient-handling capability of the power supply and can cause severe fluctuations capable of disrupting the proper operation of the SPMM. This problem is minimized with the use of three 4,700uF electrolytic capacitors, C1 through C3 placed in parallel across the power supply on the ping amplifier module. Considering the duty cycle of the PAM, there is little current drawn by the 12 volt source, usually less than 400mA on average.

Before a pulse from this amplifier is sent to one of the sonar transducers, one of three relays is closed, K1, K2, or K3. These relays are controlled by sections of a 7405 open collector driver device located in the Scan Control Module (SCM). These relays would not be expected to maintain reliability under repeated switching with the currents and voltages present in the ping amplifier output pulses. However, it should be noted that the relays never close or open while voltage or current is present. Only after the relay closes and has stopped bouncing will the amplifier pulse be passed.

It should also be noted that the TXP-AMP connection is directed to the Analog Amplifier-Filter Module for reception

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

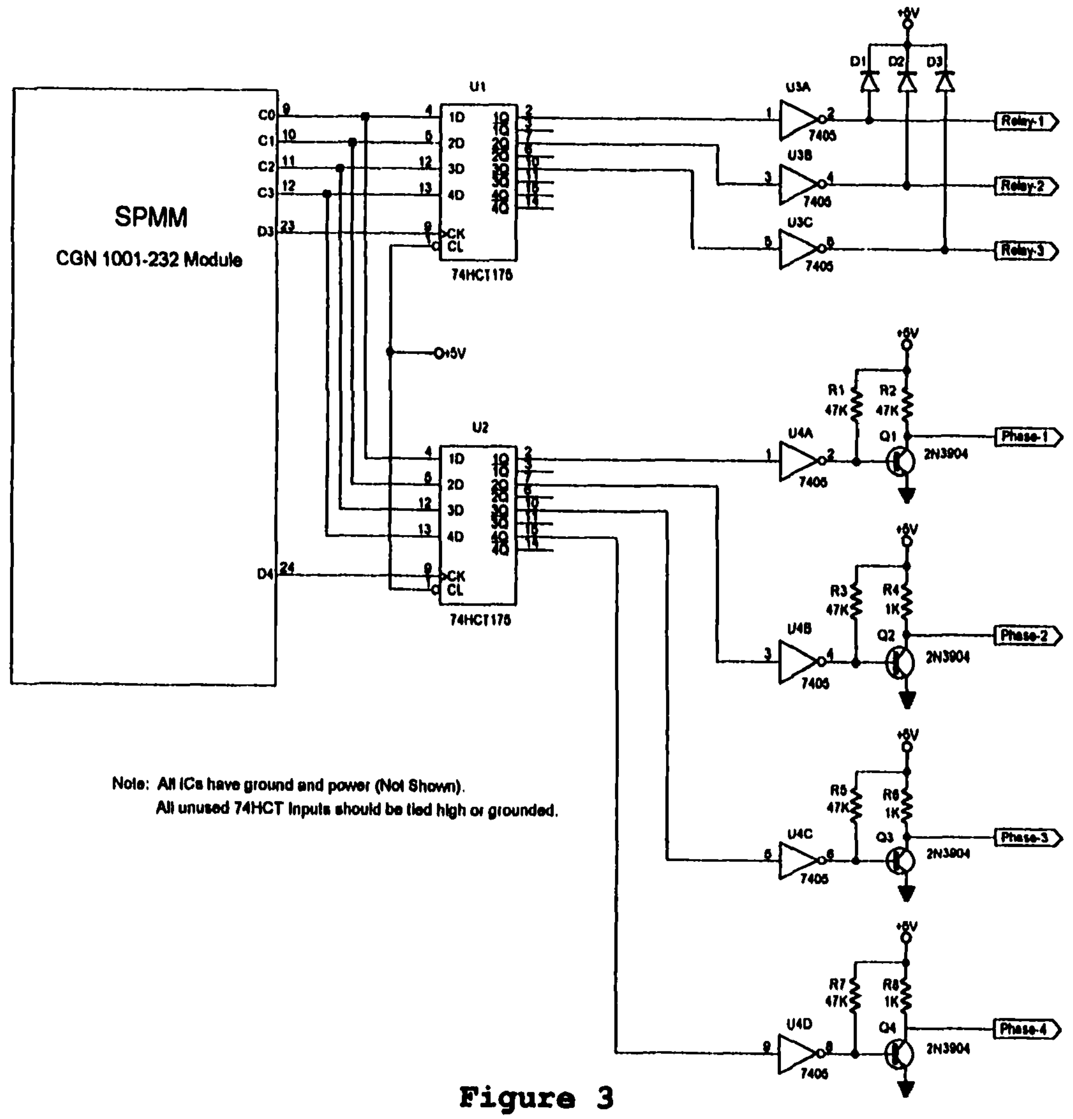
of the sonar echo pulse. The chosen relay is closed and remains closed through the power output of the ping amplifier as well as the recording phase when the echoes return to the transducer. Only after 300 feet of echo return has been recorded does the relay open.

Scan Control Module

This circuit shown in Figure 3 drives both the relays located in the PAM and also the stepping motor used to rotate the sonar head. The stepping motor for sonar array left-right rotation is driven by the SPMM microcontroller. Motor stepping is straightforward. Although not illustrated by figure, there are four motor phase wires (Phase-1 through Phase-4) and two power wires. The phase wires are connected inside the motor enclosure to four TIP-120 Darlington power transistors. These transistors switch the phase wires to ground, furnishing current through the respective motor coils.

The TIP-120 driver transistors are themselves driven by four 2N3904 transistors, each in an amplifier circuit. This drive array is shown in Figure 3. The input to these drivers are 7405 open collector inverters, each with a 47K ohm pull-up resistor. The input to these inverters comes from U2, a 74HCT175 4-bit latch. Another 74HCT175 latch, U1, is used for relay control of the ping amplifier's relay multiplixer. Both of these latches are fed from lines C0 through C3 of the SPMM. Latch control is provided by SPMM pin D3 for U1 and pin D4 for U2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Note: All ICs have ground and power (Not Shown).
All unused 74HCT Inputs should be tied high or grounded.

Figure 3

Scan Control Module (SCM)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Analog Amplifier-Filter Module

The Analog Amplifier Circuit (AAFM), shown in Figure 4, is housed inside the sonar pod and is used to amplify and detect echoed signal returns. All stages up to the detector use the LF356 wide-band operational amplifier. This amplifier is a monolithic JFET internally compensated op-amp with an approximate open loop gain of 24dB at the 192KHz operating frequency of the sonar unit. Among the other features that make this amplifier desirable for this application are extremely high input impedance (10^{12} Ohms) and a high output voltage swing at operating frequency [18].

One feature of the AAFM is its ability to change gain settings through digital control signals from the SPMM. The ability to set the AAFM to a very wide range of gains while not overloading individual amplifier stages was important to the initial design of the AAFM. This ability allows for a very wide range of transducer combinations to be tested without redesign of the basic circuitry of the AAFM.

The signal from the transducer comes through a multiplexer which is part of the PAM discussed earlier. Similar to radar systems, a sonar device using the same transducer for transmission and reception needs to be able to withstand high amplitude pulses and yet be sensitive to returning echoes. The pulses generated for pinging can reach hundreds of volts in amplitude. This would destroy the input to an op-amp on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

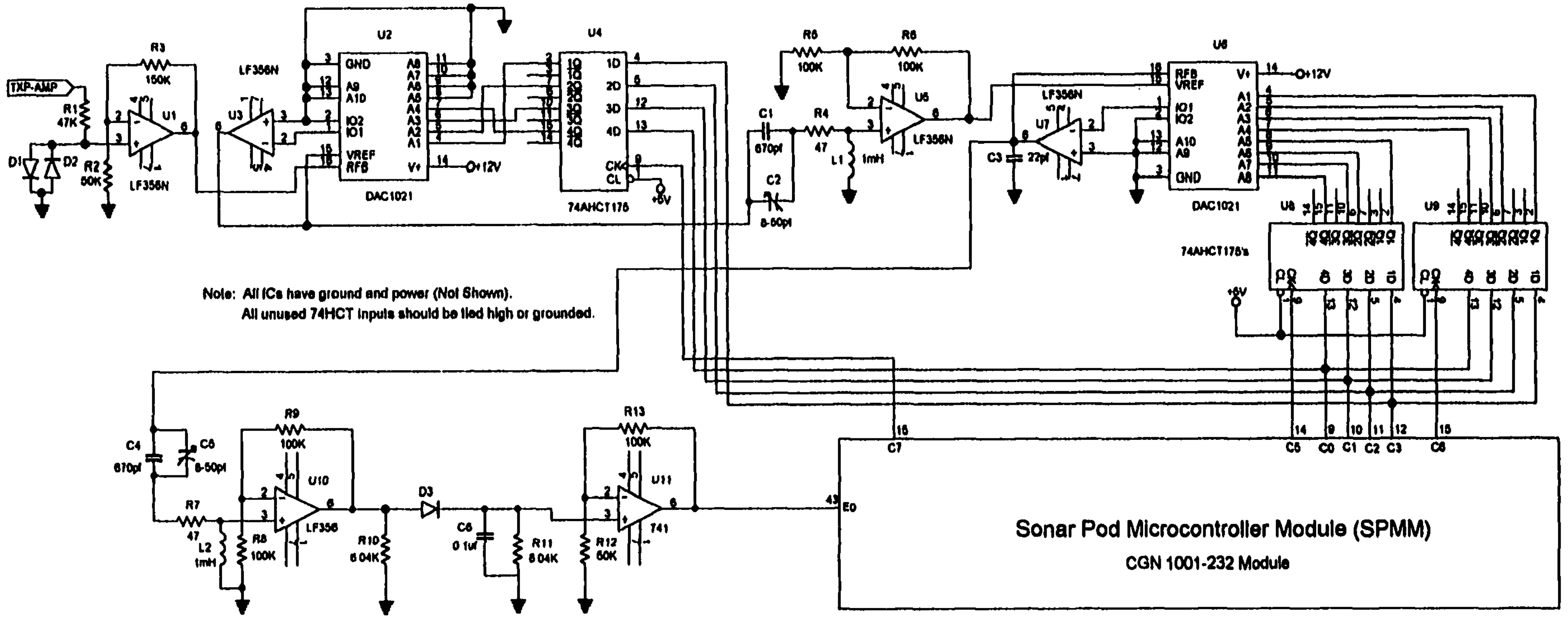


Figure 4

Analog Amplifier-Filter Module (AAFM)

the first ping if measures were not taken to attenuate this signal before sending it to the amplifier.

The first stage of outgoing ping attenuation occurs at the TXP-AMP input of the AAFM. Here a 47K-ohm resister R1 is placed in series with the input. The amplifier side of this resistor is attached to two high-speed silicon switching diodes, D1 and D2, placed back to back to shunt any current passing through the resistor to ground. The 47K-ohm resistance will not affect the strength of the ping signal as it is transmitted and will not create a substantial loss in the returning ping signal as it is directed to an amplifier stage with a 10^{12} ohms input impedance [18]. These diodes limit voltage swings at their point of connection to approximately 0.7 volts. This prevents damage to other solid state components.

A DAC1021 multiplying digital-to-analog converter U2 along with an LF356 op-amp U3 were used as a gain control stage [21]. One reason for placing a gain control stage this close to the input was to accommodate large input signals if necessary without overdriving any stage. Had this stage been placed later after several more stages of general amplification, larger signals could saturate one or more amplifiers prior to reaching the gain control stage. This would act much the same as a limiter in an FM radio receiver, flattening signals over a certain threshold and resulting in loss of linearity.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The value of the gain for this stage is set by the SPMM latching output pins C0 through C3 into U4 a 74HCT175 4-bit latch. Clocking of this latch is accomplished with output C7 from the SPMM. The gain can be adjusted from 1.07 to 16 for this stage [21]. The output of the gain control stage is at pin 6 of U3. Due to feedback, the output impedance is low. This is an important consideration for the filter Q of the next stage.

The first filter stage is connected to the output of the gain stage. Filtering is simply a series RLC circuit with the output taken across the inductor. Unlike radio receivers that must receive a broader band of frequencies, this sonar unit must receive echoes from the transmitted signal at 192KHz with only small deviations for doppler shift of the returned signal due to forward motion of the vehicle [27]. For this reason a simple single peak resonate response is adequate.

One major concern of the design was the ringing present in high Q circuits. This could result in signals not decaying rapidly enough to obtain the desired resolution from the sonar. The solution was to lower the Q by placing a 47-ohm resistor R4 in series with the resonate circuit. The closed loop gain of this stage U5 is only 2 times the Q rise in voltage across the inductor. However, the combined gain of the amplifier U5 and the RLC circuit can be much greater in practice. Two identical stages of filtering were used tuned to the same frequency (192KHz).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since little was known of the characteristics of the transducers employed in this sonar application, or how much return signal to expect from targets of varying density and structure, it was thought prudent to design an amplifier-filter stage with as much gain versatility as possible to prevent the need for redesign. For this reason a stage of attenuation was utilized.

This attenuation stage consisted of another DAC1021 U6 and a LF356 op-amp U7. This configuration has a overall gain (attenuation) from 0.001 to 1 [21]. This stage is an attenuator capable of limiting the signal being fed to the second filter stage. Once again, the attenuator, like the gain stage, has a very low output impedance due to the feedback from U7.

The second filter stage U10 is identical to the previous filter stage. The output of this stage is directed through a diode D3 into both a capacitor C6 and a resistor R11 in parallel. This combination represents the detector stage. This circuit in effect rectifies the signal and places a small barrier potential (0.3 volts) against the signal, which helps to eliminate noise.

The final stage is a non-inverting amplifier U11, a 741 general purpose op-amp with a low pass filter which takes the output of the detector and filters the 192KHz component out, leaving the echo envelope. This signal is directed to the SPMM analog input E0 where it is sampled and converted into a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

one-byte data element. Thus, sets of converted digital information gathered from this output comprise the data packet to be later transmitted to the processor A.

U8 and U9 are 74HCT175 4-bit latches used for latching and supplying data to the attenuator stage. These latches are fed by the low nibble of port C. U8 is the low order latch and U9 is the high order latch. U8 is latched by a pulse from the SPMM's C5 and U9 is latched with a pulse from C6.

Sonar Pod Microcontroller Module

The Sonar Pod Microcontroller Module (SPMM) contains the MC68HC711E9 Motorola microcontroller that controls the operation of the sonar pod. The microcontroller support circuitry was installed as a prefabricated module. This module, the CGN 1001-232 produced by CGN of Sunnyvale, California, contains all of the necessary circuitry for clock, mode selection, RS-232C interfacing, power on reset, pull-up resistors, and power supply filtering [7]. Connections between the SPMM module and the other circuits were made with wire-wrap technology, IDC type sockets, and ribbon cabling.

Power Supply Module

Several circuit modules in the sonar pod require 5 volts for their operation. For this reason a 78H05 regulator is furnished in the power supply circuit. This regulator supplies these circuit requirements and also powers the input of a +12V, -12V converter module which furnishes these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

voltages to the op-amps and other analog devices. The power supply is fused for protection.

Power supply circuitry is not depicted in this writing but follows standard approaches to regulation and filtering and can be reproduced by any competent electrical engineer.

Pre-processor Hardware Theory of Operation

Processor A

Processor A, a MC68HC711E9 microcontroller, is responsible for sending and receiving information to and from the Sonar Pod. In some cases this pre-processor originates commands sent to the SPMM and at other times simply allows host commands to pass through on their way directly to the sonar pod. In either case the serial communications port (SCI) of processor A is connected to the SCI port of the SPMM.

The program running in processor A sends a command to the sonar pod to ping one of the three transducers. The transducer selected transmits a ping pulse and then collects returning data. This data is automatically routed to processor A via an RS-232C serial interface connection. This data from the sonar pod is sent not in pure binary, but rather in ASCII hexadecimal format. In this way decimal signal values of 0 to 255 units can be represented as hexadecimal, by ASCII pairs of $00 to $FF in value.

These data pairs are sent one line at a time with each line representing 20 one-byte values for a total of 40 hex

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

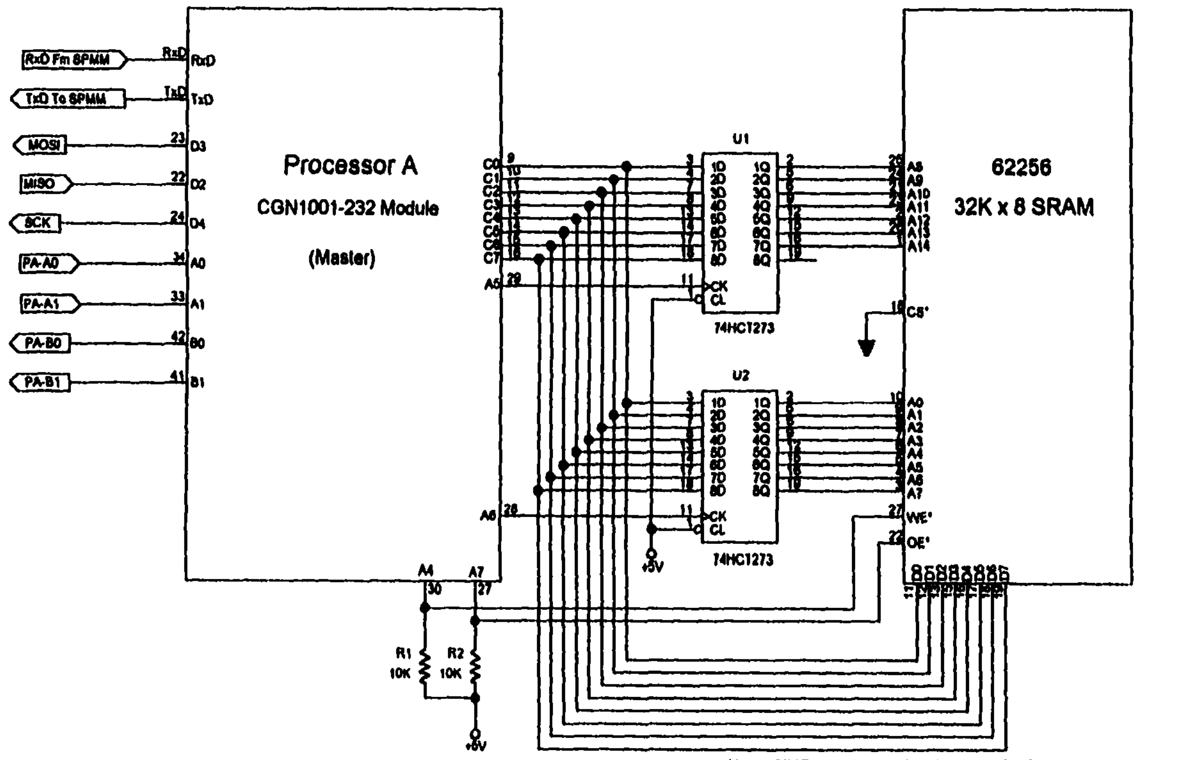
digits. A total of 15 lines represent the full 300 values of data. As mentioned earlier, each value represents one foot in distance from the vehicle to the target. Each line is terminated with a Carriage Return and Line Feed (CR/LF) character set. There are CR/LF sets also before the first line and after the last. The final information sent is the ASCII string "DONE" which represents the finished operation.

This bundle or "packet" of data is stored in processor A as it is received. Figure 5 shows that the storage medium is a 62256 32K x 8 static RAM memory. Processor A's software determines the location for storing each packet. Enough room is present to store all three returning data packets along with a pre-loaded threshold level used as a comparison for determining event generation.

Once data has been stored, it can be retrieved by processor A by reading the RAM memory in a manner similar to the way it was initially written. It should be noted here that the RAM memory is not the main memory for processor A. It is handled as a stand-alone memory accessed much the same way that a microcontroller would access a peripheral device attached to its ports.

Figure 5 shows processor A connected to two 74HCT273 octal latches, U1 and U2, via port C. These two latches set up the RAM address, which requires, 15 bits of address. The lower 8 address bits are latched into U2 and the upper 7 bits are latched into U1. Latching occurs by placing the address

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Note: All ICs have ground and power (Not Shown).
All unused 74HCT inputs should be tied high or grounded.

Figure 5

Processor A

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information on port C and then strobing the appropriate latch via the strobe line input CK. Latch U1 is strobed with A5 and U2 is strobed with A6.

Write enable is furnished to the memory with pin A4 of processor A and output enable is provided by pin A7. Chip select (pin 18) is perpetually grounded to enable the memory at all times. If write enable and output enable are in the inactive state, the RAM device will neither read nor write information.

Writing to the memory is straightforward. Once port C has loaded the address information into U1 and U2, port C is left in the output mode and the data is placed on port C. This places the data at the RAM input pins D0 through D7. Once the write enable strobe goes active (low) the data are written to the address present at the latches.

Reading is also straightforward. The address is loaded as mentioned above. Once the correct address is present, port C goes into the input mode for reading and the RAM memory output is enabled. The data is read via port C and recorded in software. Once this is accomplished the output is disabled and the next RAM operation is ready for implementation.

Processor B

The hardware for processor B also consists of a Motorola MC68HC711E9 microcontroller connected to processor A via the Serial Peripheral Interface (SPI) provided on-chip. The SPI system is explained in detail in section 8 of the M68HC11

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reference Manual [19]. This pre-processor is connected directly to the host computer via an RS-232C interface running at a 9600 baud rate.

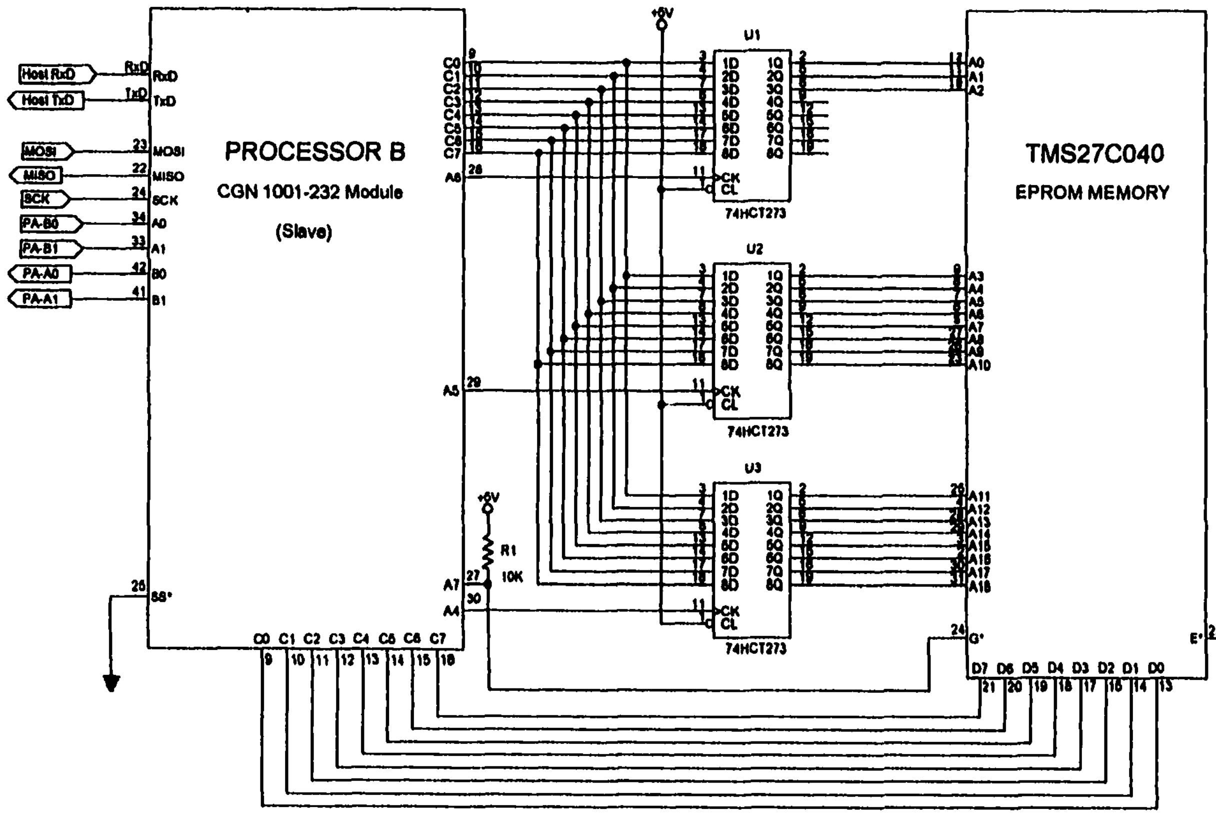
Processor B exchanges data with processor A via the SPI interface. This data represents events generated as processor A compares incoming echo levels against the threshold level present in its RAM memory.

Events arrive at processor B in the form of left-right sonar array angle, transducer number, array up-down tilt status, and range to target. These values are changed into a binary number that is used as an address to look-up pre-calculated data in a large table of values located in EPROM memory.

The EPROM used is a TMS27C040 512K x 8 configuration having 19 total address lines. This is shown in Figure 6. This large address is provided by three 74HCT273 octal latches, U1 through U3, in a manner similar to the RAM memory of processor A. These latches are fed portions of the total EPROM address, one latch at a time, using port C configured as an output. Processor B pins A6, A5, and A4 strobe U1, U2, and U3 respectively. The EPROM is enabled at all times by grounding pin 22, the enable input. Only pin A7 is used to enable the EPROM output lines. Once addresses have been fully latched, port C goes into input mode and the EPROM output lines are enabled allowing processor B to read the EPROM data at the location indicated by the latches.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Note: All ICs have ground and power (Not Shown).
All unused 74HCT inputs should be tied high or grounded.

Figure 6

Processor B

Velocity and Heading Sensor Hardware

Velocity Sensor

It was the initial desire of the author to utilize a readily available flow meter for velocity sensing. A number of boating equipment manufactures supply speed and temperature sensors along with their fish locator products. After purchasing such a product from LEI, it was quickly noted that for slow velocities through the water the device would not function adequately. The sensing principle involved the use of a small magnetically-polarized wheel that was placed on the transom of a boat and subjected to water flowing across the lower portion of the wheel. The water flow turned the wheel where a digital magnetic sensor could count the pulses.

For slow operation of the unit, the coefficient of friction of the wheel itself completely prevented it from rotating. This problem was solved by totally redesigning the mechanism. First, a small turbine was constructed which is 3 inches in diameter and has a total of 7 blades. The blades and attached hub for this turbine were constructed from a small cooling fan purchased at Radio Shack. The blade hub was filled with epoxy to attach it to a stainless steel 0.25 inch shaft that runs through a machined PVC housing. This housing was equipped with non-metallic ball bearings. These bearings are totally impervious to water and may be submerged without fear of corrosion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The shaft was attached to the original multi-poled magnet so that when water flowed through the turbine the shaft was rotated and the magnetic wheel caused the digital sensor to switch its output between high and low. There are 6 pulses of the digital level for every one revolution of the turbine. The rotational speed of the turbine is related to the velocity of water that flows through the turbine. This system generated much more torque than the original wheel itself and proved to be far more sensitive to lower water velocities.

Magnetic Compass Sensor and Microcontroller

The main thrust of this research is centered around changing the locations of echo targets as the sub or AUV moves and turns. Therefore, it is essential to have a method to sense turning of the vehicle. There are several methods for doing this that were considered.

One approach considered was the use of a gyroscope. However, position gyros (which are used to maintain heading information) are extremely expensive. Rate gyros are far less expensive and should work well for this application, but their cost can range from $200 to over $2,500. Another consideration for not using a rate gyro was their tendency to drift, due to the integration of turning rate necessary to obtain position.

One interesting concept was to use the direction sensor not only for this research but for magnetic heading

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information in the submersible. This dual purpose dictated the use of a remote-sensing compass module. The Vector$_{TM}$ electronic compass module manufactured by Precision Navigation, Inc. was chosen. This device utilizes two gimbled induction coils placed at 90 degrees to each other that have variable permeability. This allows onboard electronics to calculate a magnetic heading that has an accuracy within 2 degrees of the actual heading and a 1-degree resolution [22].

This device uses a Serial Peripheral Interface (SPI) similar to the Motorola microcontroller product line. The device has multiple control lines that must be manipulated. A MC68HC811E9 Motorola microcontroller was used. It was connected using the SPI port directly to the compass module. Both the compass module and the microcontroller were placed in a acrylic plastic tube 2.5 inches in outside diameter and 12 inches long.

The acrylic tube was sealed using machined PVC ends and double O-ring seals on each end. The clear plastic allows easy inspection of the electronics. A desiccant was also inserted into the tube and the system filled with dry air to prevent condensation.

The microcontroller used to control the compass module serves a dual purpose. The pulses generated by the velocity sensor are fed to the *IRQ interrupt input of the microcontroller. Both real-time interrupts and IRQ type interrupts are generated. The real-time interrupts count time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the IRQ interrupts count magnetic sensor level change. Software in the microcontroller continuously reports both velocity and heading information. Software listings for this microcontroller are provided in Appendix A.

The RS-232C output of the compass microcontroller is directed to the interface box and then directly on to the host computer on a second channel of serial input (COM2).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

SOFTWARE SYSTEM THEORY OF OPERATION

Introduction to Software

Languages and Tools

There are a total of four microcontrollers and two PC-based computers that require programs to operate. The four Motorola 68HC11 family microcontrollers are programmed in assembly language. The recording host is programmed in QuickBASIC version 4.5, and the processing host computer runs programs compiled with a Microsoft C++ compiler.

Motorola offers not only the AS11 assembler for their microcontroller but also the PCBug debugging tool for aiding the programmer in developing the software. The main problem with using the debug capability of these software tools is that when the microcontroller is embedded in an application that is demanding of its resources, the debug tools cannot be fully implemented. For this reason no debug capability was used on any assembly language program written for this application. The AS11 Motorola assembler was used to create the Motorola ".S19" load modules and either the BUFFALO

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

monitor or PCBUG was used to load the programs into the microcontroller memory [20].

Sonar Pod Software

Command Structure

The sonar pod is operated by a Motorola 68HC711E9 microcontroller. This sonar pod microcontroller is responsible for controlling the sonar array scan stepping motor, the analog gain settings, the ping generator and several other related functions.

Communications with this device are provided through an RS-232C connection with processor A. No sonar pod operation is initiated in the SPMM. Rather, specific commands are sent to the SPMM that are carried out one at a time with the SPMM software. Table 1 lists the possible commands that can be performed by the SPMM's microcontroller. Each command must be properly formatted using a simple convention.

A colon ":" must be used as the first character in a command string. The second character must be one of the commands listed in Table 1 and must be in upper case only. The next three characters necessary for most commands represents a numeric field of three digits.

The purpose of these three digits depends on the command itself. For example, the "W" command's numeric field represents the pulse width of the sonar ping signal in cycles. The range of this field is from 001 to 255, denoting 1 cycle

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 1

Sonar Pod Commands

SPMM Cmd	Nbr Field (N)	Purpose of Command
X	Not Used	Sends Execution to BUFFALO monitor
L	Steps	Rotates Sonar Array Left (N) Steps
R	Steps	Rotates Sonar Array Right (N) Steps
G	Gain Code	Sets the Gain of the Analog Amp.
A	Attenuation	Sets the Attenuation of AAFM Amp.
W	Ping Width	(N) = Number of Cycles of Pulse
P	Transducer	[(N) = 1, 2, or 4][(TX0, TX1, TX2)]
F	(Dummy)	Set for Fresh Water
S	(Dummy)	Set for Salt Water
B	Baud Rate	Converted to Byte for Baud Register

to 255 cycles in length. The Baud Rate "B" command field is converted to a single byte representing the field's value and is loaded into the BAUD register in the SPMM. This allows a wide range of setting for baud rate communications.

The command string is always terminated with a Carriage Return character ($0D). If a mistake is encountered in the string it is not executed; the program returns and looks for a colon to start another string. When commands are executed properly, the string "DONE" is sent out from the SPMM. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is important in certain commands because processor A looks for this string before proceeding.

The Ping command "P" not only sets the transducer to be used but also results in returning the echo data to processor A. Processor A expects to receive a certain number of control characters such as Carriage Returns ($0D) and Line Feeds ($0A) before the data is received. The "DONE" string is important here and must be transmitted by the SPMM. If it is not, then a timeout occurs in processor B and an error condition is recorded.

Initial Software Considerations

The first part of the sonar pod program is the equates section, which is standard for most assembly language programs. Here addresses and constants are defined. This program, located in Appendix A, defines each label being equated.

The RAM variables section of this program defines the address labels and the number of bytes that each occupies. One buffer location, BUFFER, is defined for incoming commands storage.

The START label points to the first byte in EEPROM located at $B600. Here the stack is initialized and the MCU OPTION register loaded. Also a subroutine INIT is called to initialize certain values and conditions. This initialization of the system is called only once. There are numerous memory

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

locations, pointers, and control settings established here. These can be easily identified by reference to the INIT subroutine.

The Main Line Program

As in the other assembly programs written for this research, the MAIN label reflects the beginning point for a loop that is executed perpetually while performing the operations required by the software. MAIN first points to the BUFFER, a buffer storage location where incoming commands will be stored. A CR/LF sequence is sent to processor A via the SCI port. This practice of first establishing a new line goes back to the debugging phase of this software, where only a computer in terminal emulation mode was connected to the SPMM. This cleared the line on the terminal before dummy commands from the terminal were entered for testing.

Next in the main line sequence is the retrieval of a single character using INCHAR. It is then checked to see if it is an "X." If so, then the Motorola BUFFALO monitor located at $E000 is summoned. Once again this is a debugging feature not seen in run-time operation. If the first character is a colon, then validity is established and the next command character is sought. If a valid colon is not found then a loop back to MAIN occurs.

Any new character is checked against a list of possible commands as listed above. If the character is not a valid command, then the program loops to MAIN. If a valid character

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is obtained, then a string of 3 characters is read and converted into a value, NUMBER, ranging from 0 to 255. This numerical value is used with most commands, but if required must be entered even if only a dummy field.

The main line program then branches over each program segment until the correct segment representing that command is located. At this point there are a number of different actions that can be taken.

Rotation Commands

There are two rotation commands common to this systems level. The sonar head can be rotated right or left. The subroutines that accomplish this are RIGHT and LEFT. These subroutines must establish a time delay for steps because the SPMM can rotate electrically much faster than the stepping motor can keep up with the stepping action. After experimentation with the system, the fastest reliable speed was established with a delay equivalent to 1332 loops in the DELY subroutine. This equates to a 4mS delay. This delay subroutine is called with this loop value located in the SDELY RMB memory location.

Once RIGHT or LEFT has this delay value loaded in SDELY, the subroutines load the previously established value of NUMBER into accumulator B on the SPMM. A loop is established and this number is counted down to zero while calling either R_STEP or L_STEP. These subroutines use a Step Table, STP_TABL, to look up the next sequence of phase values for the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

stepping motor. A pointer points into this table that constantly indicates where the current phase in use is located.

One unique feature of the stepping motor system is the fact that a much larger current than the rated value can be used on the motor. Most of the time the motor is idle. This occurs while the transducers are pinging and recording. The current to each phase can be turned off while the head is stationary. At present, rated current is used on the motor, but little is known about the rotational torque necessary when a submersible carries the sonar system and strong water flow is present. The KILL_I subroutine is used to kill the current to all phases of the stepping motor while not rotating.

Before the motor is used for the first time the lines that drive the motor have to be initialized. This is done with the MOT_INIT subroutine. Not only does this establish a beginning phase setting for the drive lines, but it also sets the STEP variable to zero.

Ping Generation and Recording

The pinging of the transducers is done by calling the subroutine PING. This subroutine first resets the PGM circuit by pulling the RESET high and then low (SPMM pin A6). After this is accomplished the PING_CT value in number of ping cycles is loaded from EEPROM and stored in lines B0 through

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B7. Next the PING bit A5 is strobed and the ping is generated.

The PING subroutine is called from the main line program after setting the transducer number (NUMBER) into the RELAY variable and calling RLY_ON. The subroutine RLY_ON turns on transducer multiplexer relay number 1, 2, or 3 with a RELAY variable value of $01, $02, or $04 respectively. When the program is finished with both the ping and echo recording the RLY_OFF subroutine is called and these SPST relays are all opened.

Once a ping is generated, data recording must be instantly initiated. This sonar was designed to be operated in both fresh and salt water. The problem here is that sound travels slightly faster in the denser salt water. Two separate recording programs were constructed in the software.

DATA_FW points to MCU RAM memory at location $0080, then loads data for the 300 feet of range to store. An analog to digital conversion is begun and readings are taken every 790 E-clock cycles of the SPMM. This elapsed time represents the time of flight of a ping to and from a target for one foot of range. Once 300 samples are taken the recording is finished.

DATA_SW is identical to DATA_FW but is the salt water version that takes 775 E-clock cycles to complete one foot of range. These two subroutines are selected from sonar pod commands "F" and "S." The default setting is "F."

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Once data is recorded to the SPMM RAM, a segment of the main line program following the location where the ping is generated begins reading this data, and with calls to PRT_PAIR, bytes are converted to hex representation and sent to the output routine OUTPUT. Return characters and line feed characters are used to format a packet of data into 15 rows of 20 sets of hex digits (40 characters per line). Once the full complement of data has been sent to processor A via the SCI port, the SPMM sends the "DONE" terminating string by calling the subroutine DONE.

Gain and Attenuation

Gain is controlled by taking the GAIN_VAL Gain Factor value originally loaded from NUMBER in the main line segment and placing it into the low nibble of port C in the SPMM after being complimented. This is done with a call to the GAIN subroutine. Once located at the output of bits 0 through 3 of port C, the gain latch strobe for U4 is actuated by a high-level signal on pin C7 of the SPMM. Since only the lower nibble is used, the gain factors can only range from 0 through 15 (000 - 015). The table of actual gain values versus the gain factor used is shown in Table 2(a), along with attenuation factors, Table 2(b).

Attenuation is controlled much the same way as gain. The attenuation subroutine ATTENUATE is called with the Attenuation Factor located in ATEN_VAL. Once again, the location's value originated with the NUMBER value passed by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 2

Gain and Attenuation Factors

Gain Factors

Factor	Gain	Factor	Gain
000	1.07	008	2.29
001	1.14	009	2.67
002	1.23	010	3.20
003	1.33	011	4.00
004	1.45	012	5.23
005	1.60	013	8.00
006	1.78	014	16.00

(a)

Attenuation Factors

Factor	Attenuation	Factor	Attenuation
000	0.0000	128	0.5000
001	0.0039	144	0.5625
016	0.0625	160	0.6250
032	0.1250	176	0.6875
048	0.1875	192	0.7500
064	0.2500	208	0.8125
080	0.3125	224	0.8750
096	0.3750	240	0.9375
112	0.4375	255	0.9961

(b)

the command itself. Unlike GAIN this subroutine has to load a full byte in port C and strobe two latches, one for U8 and the other for U9. Values for 18 if the possible 256 attenuation levels are shown in Table 2(b). The attenuation is equal to the value (.0039063) times the attenuation factor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus, for an attenuation factor of 80, the attenuation is 0.3125, which can be found in the table.

Processor A Software

Introduction to Processor A Software

This processor is used to collect data from the sonar pod SPMM and compare the data against a pre-loaded threshold of values for each foot of range sampled. Since the threshold information is in RAM rather than EPROM, it must be loaded prior to use. Although the sonar pod can change its gain to increase or decrease the received signal, the threshold loaded can reflect much about how the signal is to be analyzed.

The threshold can be simply 300 bytes of the same value or the threshold can be set in a non-linear manner so as to lower the threshold values for distant targets that have lost much of their energy by the time they travel the longer distances. These are choices that can be made flexible by placing this array in RAM memory. However, a mechanism for loading these values into RAM must be provided.

Initial Start-Up of Software

The listing found in Appendix A for the processor A software shows a standard assembly language layout. First, equates define values and address locations to be used in the program. These mainly define locations in the control block

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where the Motorola 68HC11 family microcontrollers control all of the various sub-systems [19].

Following the equates section the RAM memory allocation begins. This area of the program defines variables and buffers used by processor A. Two such buffers should be noted. First, the SPR_BUF and its pointer, SPR_PTR, are used for the SPI buffer system for receiving data from processor B. The SPT_BUF and SPT_PTR represent the SPI transmit buffer system.

Located at address $B600 in EEPROM, the program begins by setting up the stack and OPTION register. Initialization is performed as a single call to the subroutine INIT. This subroutine sets up initial values, control information, and pointers for the software.

Main Line Program

The MAIN location in the main line portion of this software is an entry point to a loop. The first thing that occurs in the main line program is an examination of line B0 (request to send from processor B). If data is ready to be received from processor B then the RECEIVE subroutine is called to receive the data.

If no new data is available, the software checks the FUNCTN variable for an "R" value. If there is an "R" in FUNCTN, then the program continues running by calling RUN_SNR. If no "R" is present, then the software loops back to MAIN and begins again. If a new command comes in from processor B, or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

perhaps from the host computer via processor B, after reception, it is analyzed to determine its nature and destination.

Commands are of two basic types. Any command that starts with a "<" character is intended for processor A. Any command intended for the SPMM in the sonar pod begins with a ":" character. Colon-originated commands are defined under the sonar pod software section. Table 3 shows a list of possible commands that can be identified by the main line program.

Each command listed in Table 3 has a program segment associated with it that carries out that function. These segments are part of the main line of the program and are not called as subroutines. The main line program tests the command for each and if there is not a match then the next program segment checks and so on. These segments are self-explanatory by observing the program listing.

The CHK_COL segment of the main line program is executed if the "<" character is not identified as the first character in the command string. If there is a colon, then this segment simply relays the command to the sonar pod SPMM.

The commands for head rotations right and left and also ping generation must be sent to the sonar pod in an ASCII string format. For the sonar pod to perform one of these operations it is necessary to send the command string from either the host (non-run-time mode) or from processor A (run-time-mode). There is a need for processor A software to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3

Processor A Commands

Char	Definition of Command
X	Vectors execution directly to the BUFFALO monitor system.
R	Runs the program. This sweeps the sonar head and gathers returning echo data.
I	Performs an initialization on the sonar head.
C	Performs an initialization then a centering of the sonar head.
U	Manually tilts the sonar head to the up position.
D	Manually tilts the sonar head to the down position.
B	Boot BUFFALO Monitor (Not needed since program modifications).
F	Sets fast baud rate for Serial Peripheral Interface (SPI).
S	Stops the head motion and recording of data from the SPMM.
P	Point to threshold area of RAM memory. (Before loading threshold information.)
T	Transfer threshold information in string form to RAM memory.
e	Generate error report. (Not used currently in run-time operation.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

generate these commands. This is done with a subroutine called MESSAGE. First the IX register is loaded with a pointer to an ASCII string (command). Then the MESSAGE subroutine is called. This routine takes the pointer in IX and starts transferring characters pointed to by IX to the output device, the SCI port connected to the sonar pod.

A set of predefined Form Constant Character (FCC) strings exist at the end of the source program that can be pointed to and transferred to the sonar pod as commands.

Head rotation to the far left position is accomplished through the ROT_STP subroutine. This subroutine tries to rotate the head 500 steps to the left. This results in having the head hit a mechanical stop. After hitting this limiting device, no further motion to the left is possible. This initializes the head position, since no limit switch is built into the system to automatically stop the motion.

Head tilt is accomplished with calls to one of two subroutines, HEAD_UP or HEAD_DN. These control line B7 of processor A, making this line high for head up and low for head down positioning. This bit controls a relay to supply 12 volts to the pneumatic valve if the bit is high and 0 volts if the bit is low.

Running the Sonar

RUN_SNR pings the transducers, records and thresholds the data, and sends event information to processor B. This subroutine points to the RAM memory where returned data from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

echoes will be placed. The IX register is pointed to the messages for generation of pings, and MESSAGE is called to send the commands. Pings are then generated by the sonar pod. The PRC_PING subroutine is where data is collected into the RAM memory pointed to by the IX register as the data returns from the sonar target.

The PRC_PING subroutine takes incoming data from the sonar pod and places it into the memory of processor A. Here the ping command selected is sent to the sonar pod SPMM and the subroutine TO_MEM is called to collect data into the RAM memory. Once a full collection of data has been made representing the 300 values sampled, the PRC_PING subroutine calls CHK_DONE to see if a "DONE" indication was received. If this string is found, no error is generated.

The above-described process is repeated three separate times, one for each sonar transducer. When this is accomplished the ANGLE value is adjusted to represent a new left right angle position and PRC_ROT is called to rotate the head to the next position. If at this time the head is already at its far right position, then the rotation is to the far left starting position and the ANGLE value is cleared to reflect the initial far left angle position (0).

The next step in the RUN_SNR sequence is the data processing step. PROCESS is the location in the RUN_SNR subroutine that takes the RAM data and compares it to a stored threshold located inside the same RAM memory device. PROCESS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sets up pointers for a call to THRESH where the actual threshold comparisons are made.

In THRESH the data elements still in hex form are compared to stored threshold. The variable COUNT represents the range count as the system starts at the closest range and looks outward to the 300 foot limit. AB_CTR is a variable that counts the number of feet of range that continually break the threshold. Such an echo could be representing an object with much more depth than a single probability sphere can represent. In this case every few feet of constant over-threshold data return can generate a new event.

In some cases, repeated closely-spaced echoes could generate multiple events. For this reason the CX_CTR keeps repeated close crossings of the threshold from triggering multiple events. This count is used to space the occurrence of close events.

Since both stored echoes and stored threshold information is in hex format and conversion takes extra time, the comparisons are made in hex format directly. First, the high order hex digits for the two values are compared. If there is one clearly higher than the other there is no need to compare the lower hex character. However, if the two high order characters are identical then it becomes necessary for a comparison of the low order characters. The threshold is not considered crossed unless the echo data is greater than the threshold value.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It should be noted that when a ping is generated the outgoing ping signal is sent directly to the AAFM circuit where it is clipped for protection of the first stage but still possesses an extremely strong signal content at 192KHz. The ping signal continues even after sampling begins in most cases. This results in values of $FF being returned for the first several samples. Even after the ping has finished ringing in the RLC circuits, high level samples continue and thus perpetuate this effect for a few more samples. For this reason, thresholds should be set to $FF for the first 10 or more samples.

Event Formation

Events are generated in this subroutine that are sent via the SPI port to processor B. These events are in the form of formatted strings. An example of this string is shown in Table 4 along with a legend for decoding.

Threshold Commands

Commands coming from the host computer traveling through the pre-processor chain such as "W" or "G" are intended for the SPMM to set parameters or generate pings. These commands begin with a colon ":". Any command traveling down the chain with a less-than symbol (<) will not be sent to the SPMM, but rather to processor A. There are two such commands. The "p" command when issued by the host is echoed by processor B to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processor A and causes an internal pointer to locate to the start of the threshold storage area in the RAM.

Table 4

Event String Formatting

Example: >E2030C4 <CR>

Character	Description
>	Identifies start of event string.
E	Always present at this position.
2	Transducer number (0-2)
0	Head position (0=UP, 1=DN)
3	Horizontal angle (0-5)
0C4	Hex Range (0C4 = 196 feet)
<CR>	($0D) RETURN character

The "T" command when issued, with ASCII hexadecimal data following, instructs the software to load the RAM and advance the threshold RAM storage pointer that was originally set by the "P" command. An example of both of these commands is shown in Table 5.

After the above commands are issued, the pointer has moved from its initial position to a position 8 bytes higher in RAM memory. Each recorded threshold value takes 2 bytes as the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information is stored in hex format. Without issuing another "P" command, the "T" command can be issued over and over until the RAM has its 300 values of threshold information. The lower limit for the "T" command is one ASCII hex set.

Table 5

Point and Threshold Commands

P<CR>	(Points to start of threshold RAM)
TF3A4127E<CR>	(Loads $F3 $A4 $12 $7E into memory)

Serial Peripheral Interface

The communications between processor A and processor B occurs via the Serial Peripheral Interface (SPI). This interface is used by many Motorola microprocessors and microcontrollers to facilitate fast and effective serial communications between a master and slave type device [19]. Processor A was chosen to be the master and processor B the slave. These devices were hard wired in this configuration.

It is necessary to provide handshaking between these two microcontrollers. Several of the B output lines were dedicated on each MCU and connected to corresponding input lines on the opposite MCU. This system allowed either device

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to request data transfer. Care was taken to not allow "deadly embraces" between the two units when both had requests at the same time. This configuration, although somewhat complex, proved more than adequate for communications in both directions.

Processor B Software

Initial Considerations

The composition of the processor B software program begins with the equates section. This section sets up values and addresses for system reference. Following the equates, originated at $0000, is the RAM variables section. This is where buffers, variables, and pointers are located. In this area of RAM memory the FIFO_Q buffer is located. This buffer is a queue capable of buffering serial output to the host. When events are being generated faster than they can be sent at 9600 baud, this buffer takes the overflow and spools the data until the system can catch up with the rate of incoming data.

The initial start-up location in EEPROM begins at $B600. At this location a one-time set-up of stack and OPTION register is performed. Initialization occurs at this point.

The INIT subroutine is called before any data is transferred. This is the initialization subroutine which is responsible for setting all software parameters. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

subroutine sets jump vectors for interrupts, pointers, initial values, and control registers. It is called only one time.

Main Line Program

The MAIN Label in the processor B software identifies the start of the main line Program. MAIN is the beginning of a loop that is performed repetitively. The first action taken in MAIN is to check the SCI_RDY flag for indication of an SCI command pending from the host computer. Commands from the host may be of two types. If the command has a colon ":" as the starting character the command from the host is a sonar pod command that will be relayed through processor A to the sonar pod. If the command begins with a "<" character then it is intended for processor A as a processor A instruction.

If a command is present from the host, the software branches to the SCI_IDR location for handling. If there is no indication of a host command, then the MAIN routine checks SCI_RDY, the status indicator for the SPI system, for any information such as an event arriving from processor A. If there is data from processor A, then GET_DAT is called and data is retrieved. If no host data or processor A data is available, then the MAIN loop repeats.

The GET_DAT program segment obtains data received from the SPI and acts upon it. This segment looks for proper formatting of the incoming stream and records an error if not in the correct format. Two types of data are received from the SPI port of processor A. Events are the normal element of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

transmission. However, upon request, processor A may send an error report of its status to processor B for relay to the host. This facility is included in the software but not utilized at this time.

The SCI_IDR program segment is responsible for retrieving the commands sent from the host computer. Since no command is processed in processor B but is simply relayed to processor A, any data coming into this program from the host is automatically relayed to processor B. This program segment is responsible for taking data from the SCI_BUF buffer and loading it into the SPI transmit buffer, SPT_BUF.

Interrupt Service Routines

The SCI receiver and transmitter use the same SPI interrupt service vector. The SCI_INT interrupt service routine first looks at the RDRF flag and then the TDRE flag in the SCI system to see which operation is being requested. If data is available then the SCI_ISR is serviced. This receives the data and places it in the correct buffer location. When finished, or if there is no received data available, then transfer is made to the location that checks
SPOL_CT to see if the output system is sending data. If data is to be sent back via the SCI to the host, then the program waits until it can send data by checking the TDRE flag.

The SPI_ISR is the interrupt service routine for the SPI system. This routine handles data coming in from processor A by generating an interrupt every time processor A sends a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

character. This routine will do one of two things. If output B0 from processor A is high (request Acknowledge) then processor B has the attention of processor A for the purpose of data transfer to processor A. This will continue until processor B lowers its B0 output line to release processor A from its obligation. If no acknowledge is present, then the SPI interrupt can only mean that processor A is sending a command or data to processor B. If processor A pulls its line B1 high during a character transfer, this routine resets the SP_PTR to the location of the first position in the SPR_BUF buffer.

Table 6 shows the basic connections between processor A and processor B. Six pins are listed for both processors A and B.

Table 6

Processors A and B Connections

Pr A	I/O	Pr B	Definitions
B0	0 -> I	A0	Acknowledgement that P-B can send
B1	O -> I	A1	Tells P-B to reset the rec. buf. ptr.
B2	O -> I	A2	Not Applicable
A0	I <- O	B0	Requests to send data to P-A.
A1	I <- O	B1	Not Applicable
A2	I <- O	B2	Not Applicable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The TRANSMIT subroutine is called by the main line program after loading the SPT_BUF SPI buffer. This routine raises line B0 from processor B to processor A's A0 input pin. The purpose of this is to request that data be transferred from processor B to processor A. This subroutine will hang execution until processor A sends an acknowledge on its line B0 to processor B's A0 line, along with SPI rotations, until processor B releases the request for transfer.

Other Communications Subroutines

The PUT_SPOOL subroutine is used for placing characters of data or other command information into the output buffer to be spooled to the host computer. This routine is a virtual output routine for all data going to the host. To assure that no characters are sent out-of-turn, this routine is called whenever any output information is to be sent to the host computer.

The SPOOL subroutine when called sends a single byte found in the output buffer (FIFO_Q) to the output SCI port. If there is no data in the queue then this routine must not be called. The SPOL_CT is changed each time a character leaves the buffer. When this counter is zero there is no more data to be sent to the host computer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Event Handling

Subroutine MK_EVENT sends a string to the host computer containing the X, Y, and Z locations of an event generated from a threshold crossing which originated in processor A. Along with this coordinate information the event also contains the range information in hexadecimal necessary for the host to determine the size of probability spheres to be generated in the data structure.

The first thing that MK_EVENT does is to formulate an address to be used to look-up the pre-calculated cartesian coordinate information found in the EPROM connected to processor B. The MAKE_ADR subroutine does this. This subroutine looks at strings entering the SPR_BUF from processor A. Once this buffer is loaded with a new event, a two-byte address into EPROM contents is formed.

Once an EPROM address is formed based on the event generated in processor A, the EPROM contents are read as ASCII information. This information is loaded into the output buffer in the proper order and transmitted along with range information also in ASCII. Range information is formed separately with the FIG_RNG subroutine which takes the hex range information found in the event and converts it to a three-byte ASCII decimal.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Other Event Handling Subroutines

The subroutine OUT_XYZ sends five characters to the SCI port from the EPROM. This subroutine is called by MK_EVENT several times in the formation of a complete event string to be sent to the host computer.

SET_ADR sets the medium and high bytes of the EPROM address into the address latches. SET_LADR sets the third or lowest order byte into its address latch. These routines are basically latch drivers.

GET_MEM loads a byte from the EPROM after setting up the address latches with SET_ADR and SET_LADR. A series of other subroutines enable and disable functions on the EPROM and are responsible for generating strobes etc. The subroutines can be seen in the source listing for processor B in Appendix A.

Recording Host Software

Introduction

A computer was needed to ride on the surface platform above the submersible portions of this sonar system to gather data. This system does not require great speed or storage capacity either in RAM memory or hard disk space. Its main function is to create files of sonar data to be taken back to the laboratory where the processing host can digest the data and create data structures.

This computer should possess several attributes. First, the system should be physically small. A laptop computer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

would have been ideal for this application; however, one was not available to the author. The second choice was a small light-weight PC capable of being mounted securely onto the instrument platform.

The second attribute that the gathering host must have is the ability to input and buffer serial input from two input channels (COM1 and COM2). No data can be lost while the operating system and data gathering applications programs pause to write data to the hard disk file or complete other functions.

The third attribute for the data gathering computer is enough speed to keep up with the volume of data to be recorded without generating buffer overflow errors which would both corrupt data and halt the gathering operation.

Hardware and Software Used for Recording

The choice of computer hardware for this application was a Dell model 333s/L 386-SX desktop computer. This computer was equipped with MS-DOS version 6.22, Microsoft QuickBASIC, and data recording programs written in QuickBASIC. This worked well in both preliminary tests and actual field trials.

The program source code for RECORD.EXE and LOGIT.EXE, the two data recording programs, is furnished in the Appendix C. These not only record sonar data but are menu-driven and allow easy access to functions such as Gain, Attenuation, and Width for control of the SPMM located inside the sonar pod. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ability to easily shift baud rate communications between processor A and the SPMM was also facilitated. Featured in this software is the ability to start and stop the sonar head movement and data gathering commands originating in processor A. The software is written to be as user-friendly as possible, as the environment of use puts extra burdens on the technicians operating the equipment.

Processing Host Software

Introduction

The host computer must be capable of fast operations including floating point calculations. A 100MHz Intel Pentium-based computer proved to be more than adequate for meeting the necessary requirements for a non-real time evaluation. Speed is not as important in this phase of the system evaluation because the computer can take as long as necessary to process the data once gathered. However, much effort was undertaken to construct the host software to optimize for speed so as to make the system more portable to an actual end-use application which is planned for later.

Much consideration was given to the choice of languages for use in the host processor. Both speed and transportability to other operating system platforms entered into the decision. Windows 95/98 by Microsoft is reported to be a less stable system than UNIX and its derivatives such as LINUX [9]. LINUX was ultimately chosen because the author

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

believed it would be easier to implement in a submersible-based sonar system.

Microsoft Visual C++ Version 5.0 was licensed to the author. A compromise was made in regard to the use of C or C++ as the host computer language. If the host software were written in C++, and a C based UNIX/LINUX compiler was available later for the end-use host, a large scale rewrite would be required to transform the software to the smaller C subset. This is especially true for converting structures only present in C++ back into usable C code. Converting classes and objects into C language would facilitate a major rewrite.

Another major consideration was speed. In general, C language code executes faster than the more complex C++ code especially when objects are used [1,11]. A compromise between the two approaches was made. The Visual C++ compiler was used for the application. Although a few convenient features found in the C++ language were used, mainly in the area of console input and output, for the most part the code was written in a form that could be easily modified for transporting to a C-based machine platform. The Host.cpp program is listed in Appendix B.

Host Data Structures

The space in the projected path of the AUV or submersible was mapped in a 3-Dimensional "char" type array called Water_Space. This occupancy grid array was dimensioned as 151

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

elements along the positive X-axis, and 141 elements comprising both the negative and positive portions of the Y-axis and Z-axis. This gives a total space of 3,002,031 one-byte character type locations. It is assumed that any element of this array is empty (no sonar targets) if the element has a value of zero. All locations in this array are initialized to zero before events are entered into the host software.

To generate a sonar target representation in the Water_Space array from an event the software must "color" an area of array locations with non-zero values. It was decided to use spherical shape colorings, although other geometric shapes could be employed. The spherical shape facilitates featureless rotations. Any other shape could not be rotated without reference to orientations other than just its X, Y, and Z location.

These spheres cannot all be generated to have the same physical size in the occupancy grid array. Close ping returns from short ranges indicate a volume of probability smaller than that of distant returns. Highest resolution is obtained at the shortest ranges [3]. This is because the sonar beam fans out creating an 8 degree cone (3dB down points). The height of this cone is the target range, with the center of the generated sphere located at the geometric center of the cone base.

Since spheres are all solid (all inside volume colored) small spheres color the same array elements as larger spheres

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with the same center coordinates. The only difference for large spheres is that they have more array locations to color. Thus, when constructing a sphere centered around its X, Y, and Z location in the array, the array elements closest to the center are colored first and then coloring can proceed outward.

An algorithm could have been developed to generate the coloring locations in much the same way that coloring is done in 2-dimensional blob coloring of machine vision images. However, algorithms can take large amounts of processor time and this can seriously slow down the event entry and relocation times. It was decided to pre-calculate the locations of all affected elements and store these in a table for look-up. This greatly speeds up the process of developing these values on a real-time basis.

A straightforward approach was used to generate the sphere coloring look-up table values. A program, VECTOR.BAS, written in Microsoft QuickBASIC was used. This program iterated nested FOR loops for X, Y, and Z variables from negative values through zero into positive values. Each value (X, Y, and Z) was considered the end point of a vector beginning at the origin. The length of this vector was calculated and this value along with the X, Y, and Z endpoint values, in that order, were placed into an ASCII data file. Magnitudes for vectors range to values larger than necessary for the largest sphere to be generated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Once this data file of vector magnitude versus X, Y, and Z coordinates was generated and stored, it was sorted using the standard MS-DOS sort routine. This sort routine sorts numerical values in the order of the ASCII character set [17]. The sort file output contained a listing of magnitudes (radii), along with X, Y, and Z endpoints. All are in ASCII character form. This file contained all necessary information to generate spheres of any required radius.

Sphere generation is based on coordinates of the sphere center along with a radius. Thus, looking into this sorted vector file table, starting with the first record, the host program can begin coloring each array element until the desired sphere size is obtained. All that is needed is a method for stopping the sphere generation when the desired uniform size is obtained. This is also done in tabular form.

The SCAN_SPH.BAS program was then used to create an index file from the magnitude values of the Sphere. Program FIND_SZ.BAS is used to generate the Entry.Dat file of endpoint locations from the two previously generated files of data. These QuickBASIC programs are included in Appendix C for a better understanding of the process used.

The sphere build endpoint locations are maintained in the host program as a single dimension array called RngPts. Once range is determined, a look into the RngPts array will return a number corresponding to the last location in the Spheres array to be used in the sphere build operation. This method

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of constructing spheres is direct and can be altered by changing the values in the Spheres and RngPts arrays. This is accomplished simply by changing the data located in the "Sphere.Dat" and "Entry.Dat" files.

Host Computer Software

Layout

The layout of components for the processing host software is straightforward and typical for most C language programs. First, the headers are declared, then the constants. Following these declarations are the file declarations for file types. The global variables are then declared.

It is easy to see that a large number of global variables were used for this program. The main reason for the larger use of global variables was to speed the processing by not having to transfer data with function parameters every time a function was called. This saves some time in the execution and uses a minimum of additional memory.

The global array variables are declared including the Water_Space array, the main array for the occupancy grid. The Spheres array containing sphere generation values and the RngPts array containing entry points to the Spheres array are included in this section.

The events are maintained in a series of arrays. The X, Y, and Z coordinate points are maintained in the X_Curr, Y_Curr, and Z_Curr arrays. The SphRng is the event range array. This array is used to redraw the spheres to scale with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

their original size. The Tm_Exit array keeps a numeric time value which represents the event formation time plus the EvLife constant value. This is used to eliminate old values from the table. The EvLife constant is the number of seconds that the event is to live starting from the event generation time.

Next are found the function declaration sections of the program. These functions are defined in the source listing in the Appendix B. The definitions are found in comments directly above each function definition.

Host Main Line Program

The main line program section of Host.cpp begins with local variable definitions. Next, housekeeping operations do things such as opening files with the Open_Files function, and filling initial values into arrays with the Fill_Tables function. Initialization of the Water_Space array elements to zero is done with the Init_Array function.

The main line program then performs a "file synchronization" where two types of data are synchronized so that they can be read together. Since data is gathered and then processed at two different times, the storage of the data requires it to be in a file format. For input of the non-real-time data the processing host needs only to open and read data files.

Two files of data are required for input of gathered data. The Events.Txt file contains event information from processor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

B time-stamped and stored in the data gathering operation. File DirSpd.Txt contains heading and velocity information formulated and transmitted by the microcontroller located in the compass housing. This data is also time-stamped by the recording host computer.

The Events.Txt and DirSpd.Txt files are created under a different set of names. This is because each time the recording host program is executed it creates a new set of files to hold the events and heading-velocity information. This can occur many times in field testing on the same day. To prevent overwriting of an existing file each time, the file is opened and given a different program generated name. The select file set is renamed before presentation of this data to the processing host.

Table 7(a) shows a short but typical listing of the ASCII text to be found in the Events.Txt file. Table 7(b) shows data that is typical to the heading and velocity file, DirSpd.Txt. These two files must be read in a coordinated manner so as to synchronize the timimg of both event generation, and data indicating velocity and heading information. The host software could not accurately process information if these two files were being read with each record representing a different time.

This synchronization is accomplished in the main module after files are opened and basic initialization is performed. Once synchronization is accomplished the host computer reports

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the operator via the console output the current event time and also the current heading and velocity time for verification. At this point the program execution enters a read and process phase. This is where data are read and compiled second by second.

Table 7

Recording Host Data Samples

```
>E114036                         C024
 52.69 11.19-3.766R54            [02:14:39]
[02:14:39]                       C023
>E11004B                         [02:14:39]
 70.30-25.58-5.231R75            C023
[02:14:40]                       [02:14:39]
>E1110A0                         V1A
 156.1-33.18-11.16R160           [02:14:39]
[02:14:44]                       C022
>E112053                         [02:14:40]
 82.59-5.775-5.789R83            C021
[02:14:45]                       [02:14:40]

          (a)                        (b)
```

A function called Clock_Tick is used to create one-second steps in the program execution. However, if data is being logged for later plotting of echo structures, the logging operation takes considerably longer than one second and results in the loss of truly real-time operation. As mentioned earlier however, this is not a problem since real-time operation is not needed for data evaluation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the host software is executed without requiring it to store plotting information, it uses a one-second loop that performs an update on the event table on every iteration. Several things happen in this loop. First, the boolean variable MoreData is checked to see if more data is available to be input. If more data is available, it is read. If not, the event table continues to be updated but the reading stops.

Assuming that there is more data to be input, the program enters a program segment that first does table maintenance. The Maint_Table function is called and the event table is updated. This function is performed only once per second, even if there are more than one event in that time period.

Next, if there is a time match between current system time (Tstr) and the last read event time, the Place_Sphere function is called. This generates a sphere to represent the new data that has just been read. The center of this sphere is X_Evnt, Y_Evnt, and Z_Evnt. These variables along with Range_Evnt, the range for the event, and the constant "1" representing the number to be added to each sphere element are passed to the function.

After the placement of a new event-generated sphere in the data structure, the Table_Event function is called to log the event into the event table. The parameters used here are the same as for the Place_Sphere function with the exception that event time must be recorded. Thus, Tstr is also passed to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

identify current event time. An example of this string, "09:04:38," would represent an event captured at 9:04:38 a.m.

After executing the above functions a slice through the Water_Space array is generated on the console in rough character graphics to give the operator a look at what is located in, and slightly below, the Z-axis plane where Z is equal to zero. The operator is then prompted as to whether a slice should be made. If so, a detailed set of all sphere colored points are logged to a data file for later plotting of a scatter plot. This plot depicts the slice with much greater detail than the rough character graphic shown on the console.

If the MoreData boolean variable is false, this would indicate that there is no more data to be read. In such a case, this main line program loop described above would do all of the same functions with the exception of the Place_Sphere and Table_Event calls. These are only needed for entering new events and thus not applicable in this case.

When in the process of reading and processing new events, an event is input whose time does not match the current time, variable Time_Match becomes false. At this point the StrTmInc function is called to increment the string time variable used to identify the next possible event time being read.

LinTime keeps what is called linear time which is a computer integer value of time used for event aging. This is done with the MkTimeInt function.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

At the point where time changes, the seconds variable is updated and execution continues for the next second. The functions and loops are inside a while loop constructed to be perpetually enabled. Thus, the program continues to run until terminated with a control-break entry from the console.

Host Computer Functions

Function Event returns event information from the events file. It is called with the Tstr time variable equal to the time of the event that the main line program is looking for. The first time that Event is called the Tstr may be equal to a null string in which Event will return the first time found in the events file.

The Event function returns a boolean type that reflects the availability of data. Global variables affected are X_Evnt, Y_Evnt, Z,Evnt, and Range_Evnt. These variables defined earlier represent the cartesian coordinates of the event center along with its range.

HeadVel is a function that is called much the same way as the Event function. However, it returns both vehicle velocity and also vehicle heading. These variables are floating point types. The heading is an average heading for the plurality of heading samples taken in the specified second. The velocity value (turbine sensor) is reported less frequently than heading and thus the returned value will be the last reported value for that seconds time interval. The returned value is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

boolean and reflects data availability like the Event function.

The function Place_Sphere is used to either place an event sphere in the Water_Space array or to remove a sphere from the array. To place a sphere the function is called with a (1) value passed to the character variable Incr. This adds one to each sphere element. If called with this value equal to (-1), then each array element is decremented, resulting in the removal of the effects of previous sphere generation on the array.

The function is also called with X_Pos, Y_Pos, Z_Pos, and SpRng as parameters. These are X, Y, and Z positions for the sphere center along with a range value to size the sphere.

Rotate_LR is a function that rotates points in the cartesian coordinate system based on an angle in radians passed as its single parameter. Since rotations are only about the Z-axis the value of ZP is not effected. However the function operates on the global X and Y values XP and YP. This rotation is based on a linear algebra computer graphics technique [14].

The return value for Rotate_LR is boolean and represents whether the rotation moved the points outside the boundary of the Water_Space array.

Translate is a function that moves the vehicle forward in the Water_Space array by moving all associated event sphere locations closer to the vehicle (toward the negative X-axis).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This is a very simple function. It merely adds a variable value called Feet to the XP variable. This effectively translates the event spheres toward the submersible or AUV assuming forward vehicular motion.

Most arithmetic operations carried out in the host software involve only simple integer additions and subtractions. However, once every second the event table must be updated with both rotations and translations for each entry. These operations are floating point operations and must be done at high speed.

The sine and cosine functions come into play from a speed standpoint also. Many times these are calculated by evaluating a polynomial of degree five or higher [16]. If the angle of rotation is handled as an integer by not averaging a set of angular readings, this process can be reduced to a look-up table for an increase in speed.

Table_Event is used to load a newly generated event into the event table. It must be called with X, Y, and Z coordinates, range, and a time string. The new event is placed in a series of arrays that represent the event's values. From this set of data locations future rotations can occur. The Tm_Exit array element dictates the linear time when the entry will be removed from the table.

The MkTimeInt functions receives a string in the form of "09:03:38" and returns the value of the string based on hours equaling 3600 seconds, minutes equaling 60 seconds and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

string value Seconds being equal simply to seconds. These values are then added to form linear time.

OpenXYfile creates the slice file for samples of the Water_Space array. The first two characters of the file created are always "XY." The second two characters represent the hours in a two-character sub-string pulled out of the Tstr time string. The third two characters represent the minutes, and the last two represent the seconds. Naming a file in this manner tags it as to exactly when the data was recorded. The file extension is ".Txt" for viewing purposes. CloseXYfile is the function that closes this file.

XY_Log is a function that scans the Water_Space array and stores a set of X and Y points into the file opened with the OpenXYfile function. There is no parameter returned. This file is furnished a depth to begin the slice. This depth is usually zero. However, as the slice is developed, each X-Y location in the array is summed for 30 Z-values below the stated depth. This results in the formation of a "thickened" slice equivalent to 60 feet of water depth. This better represents the Water_Space array structures through projection onto the Z-equal-zero plane.

The Conv_Vl function is to take a string comprising hexadecimal numbers from the velocity sensor and convert this pulse information from the turbine wheel into actual velocity in feet per second. This velocity value is to be returned to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the program. The value returned should represent the floating point value of the velocity.

Currently no table of conversion values from pulses per time interval to velocity has been constructed. Neither has a constant been developed that represents a quasi-linear relationship between these two values. Currently a "Forced Velocity" is used while processing the event information. Thus, known velocity is input into the processing host software representing the steady velocity of the instrument platform.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

TESTING AND EVALUATION

Testing Purposes and Objectives

Introduction to Testing Philosophy

The major objective of this research is to obtain proof-of-concept validation for a new approach to sonar obstacle avoidance. This research is not intent on laying the ground work for commercial production of these systems or creation of a manufacturing model. This is a specialized sonar concept with a limited number of end-use applications, but one that when needed can possibly yield an ability not currently seen in the field.

The evaluation of this research was conducted with the intent to either point to more research into the basic concept or to establish that the approach is lacking in merit. One purpose of this research is to demonstrate a working model of the overall system. Further research can and should refine the quantitative analysis of the applied concept for qualification in various roles in existing and emerging marine technology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For the most part, the evaluation of this concept is based on graphical imagery which leans toward subjective evaluation. This approach is much the same as it has been for evaluation of most graphically-based sonar systems for many years.

Instrument Platform Development

Need for an Instrument Platform

Much consideration was given to the field test phase of the testing procedure. Many factors were considered in configuring the testing scenario. The seemingly obvious test would be to attach the sonar systems onto "The Vindicator," a one-person submersible, for actual end-use testing. This approach was ruled out for several reasons.

Operation of this submersible requires much support from a crew of workers knowledgeable in the launch, operation, and recovery procedures used. Although The Vindicator has made approximately 15 dives, the launching and retrieval for this 3 ton submersible is still tedious at best. The extra workload necessary for field testing the sonar while operating the submersible could prove to be too large of a burden. Later, after the sonar systems have been proven and refined, the transition to the submersible application will require far less effort.

Another factor involved in not using a submersible for evaluation is that timing for such an actual end-use test would cause the test to be performed in the coldest season of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the year. This would prove virtually impossible for the launch crew with the seasonal water temperatures in Northern Louisiana, Arkansas, or Texas where the vehicle would need to be launched.

It was decided that the best approach to evaluating the sonar system was to use a surface craft, either a boat where the sonar would be attached below the water line, or a specially created floatation system. The approach taken was to construct a small boat, with as little water drag as possible, to float the instruments needed.

It seemed logical to also include radio control as an option, as the platform was given the capability of self propulsion. In this way the platform could be controlled remotely or attached to a surface boat with its own propulsion.

Design and Construction of Instrument Platform

This radio-controlled boat constructed specifically as an instrument platform is slightly under 8 feet in length and 27 inches wide at the beam. A steel frame in the form of a double "U" was constructed so as to attach to four mounting points on the top of the instrument platform. This frame is therefore not attached to the water-tight underside of the platform, yet provides an undercarriage for mounting the submerged portions of the sonar system. This frame was made

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

easily detachable so as to allow convenient transportation to testing sites.

A 14-pound thrust 12-volt trolling motor was obtained for propulsion. A Whistler model PP300AC 300-watt voltage inverter was installed to provide 120 VAC power to operate the recording host computer and monitor. Radio control was accomplished through a 1970's era Citizens Band Radio Control (R/C) transmitter and receiver combination coupled to a controller and servo driver developed specifically for this application. This system is based on pulse width modulation (PWM) for servo position control.

The controller/driver uses an embedded microcontroller to detect channel pulse width from the R/C receiver and provide drive to a servo motor combination having far greater torque than would be seen in a small radio control model servo. This servo system utilizes digital sampling and feedback for proper positioning of the trolling motor thrust. Relays were employed to adjust the speed of the trolling motor. Available speeds are stop, slow, medium, and fast.

A small scuba compressed air tank commonly referred to as a "Pony Bottle" and a pneumatic 4-way air control valve were located on-board for driving the sonar head tilt cylinder. Much care had to be taken with the compressed air valve-cylinder system so as not to jar the sonar head repeatedly as the cylinder was actuated. It was a concern of the author to limit the high acceleration impact that occurs when the sonar

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

head moves up and down. An exhaust restricter valve along with a flow restricter were used to cushion the movement. This worked well in and out of the water when tested.

The interface box containing processors A and B was mounted and powered by the same 12-volt battery system used for propulsion and steering. This box contained a relay used to control the action of the 4-way pneumatic valve.

Below this instrument platform the submergible portion of the sonar system was mounted. Cabling was routed through clear vinyl tubing to maintain water-tight integrity. This cabling was connected to the interface module and power systems of the platform. Pneumatic lines consisted of 0.125 inch diameter nylon tubing. This tubing runs over the edge of the platform and connects to the sonar head positioning cylinder.

Testing Prior to Field Trials

Initial Systems Testing

No complete system of this size should be constructed and placed in a field test environment without first having each module evaluated as to its performance and fitness for system inclusion. This sonar system consists of various smaller systems and sub-systems of both hardware and software that must work well together. If any component fails to meet its standard, the final performance of the overall system produced will be inadequate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Testing a system of this size dictates testing of individual modules prior to their connection into the final product. This was accomplished, one sub-system at a time, in most cases. The sonar pod systems were evaluated one at a time and then as a whole by demonstrating that the sonar pod responded to commands as required. Testing of this system was performed by simple connection to a PC with terminal emulation software. All commands were tested and results evaluated.

The initial testing for processor A and processor B was not as simple as testing each section individually. These two MCUs were connected via the SPI ports that exist on the Motorola 68HC11 series microcontrollers. Much of the testing of this pre-processing system was performed by connecting separate computers, operating in terminal emulation mode, to each MCU.

Data and commands were passed back and forth and error conditions were observed. This was a formidable task in that no debug software could be employed. It proved very difficult to locate problems when this larger logical block of hardware and software did not perform to expectations. Several problems were found that took considerable time in the logical probing of the system to isolate and correct.

Dummy data were collected by the recording host representing both events and heading-velocity information to insure that this data was being recorded correctly and that the time stamp was working correctly. Initial testing of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processing host software was done in small increments. Functions were tested and linked together in a building process common to software development.

Pool Testing of Sonar Pod

Many smaller system tests have been performed during the development of the sonar hardware and software. Simulated sonar signals were passed through the analog amplifier system and recorded. Head scanning was observed and numerous other tests validated sub-system performance. However, actual underwater tests of the sonar module were not performed until early January, 1999.

Initial sonar pod tests were performed in a swimming pool located in the Health and Physical Education building on the Louisiana State University in Shreveport campus. The purpose behind this testing was not to record quantitative information on the sonar pod, but to determine usability as designed for later open water tests. As mentioned earlier, little information on the sonar transducers used was obtainable. Perhaps the reason for not making this information available was that it was proprietary in nature based on the vendor's company policy.

Since little was known about the sensitivity of the transducers used, it was essential to obtain a subjective evaluation of the usability of the sonar pod system. This evaluation revealed several valuable pieces of information on the operation of the sonar pod.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The necessary gain and attenuation settings were not known until pool testing. The level of baseline noise at the best gain and attenuation settings was assumed to be low but unknown. The relative amplitudes of returns from smaller targets compared to the "solid wall" pool echoes were also not known.

This test began by locating the entire sonar pod assembly including angle iron frame, head tilting mechanism, rotating transducer array, and electronic enclosure in the pool. This pool slopes very gradually in the shallow end but once a depth of around 5 to 6 feet is obtained the slope of the pool becomes abruptly steeper. This was the location where the assembly was placed, roughly half of the distance between the shallow and deep ends of the pool.

The assembly was placed on a plastic table 25 inches from the pool bottom. This placed the sonar pod center approximately 35 inches from the water's surface. The transducer array was pointed out into the deep end of the pool such that no portion of the 8-degree beam projected from transducer number 1 (center transducer) would hit either the water's surface or the pool bottom.

The sonar pod was connected directly to an Intel 386-SX based computer executing Kermit, a terminal emulation software program. Power was supplied by two standard laboratory power supplies: one supplied power for the rotation of the sonar head, and the other supply was used for powering the sonar pod

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

electronics. The gain factor was set to 8 and the attenuation was set to 180. A pulse width of 80 cycles was chosen.

The first ping was generated and sent to transducer number 1. The echo signal was observed for the entire 300 feet of range. Much to the surprise of the author, a screen of $FF values were displayed indicating that the signal had saturated the amplifier system. The gain was lowered and again a ping was generated. Virtually the same thing happened with signal strength being so large that it overloaded the amplifier. A pattern of lowering gain and raising attenuation continued until a very well-defined series of echoes were observed with $00 echo values displayed where there were no direct echo returns.

Both the author and his assistant became submerged sonar targets for ping returns at various positions in the deep end of the pool. In every evaluation it was noted that strong echoes, far above base-line noise values were observed. It was also noted that the resolution of the system was good. It was postulated that in the initial (high gain) trials, very small signal values in the form of echoes from the waters surface, pool surfaces, and scattering of the sonar beam from various conditions resulted in overloading the amplifier. There were no tests of the claimed 8 degree cone patterns made.

The optimum settings for the pool dictated gains at the extreme low end of the available range and attenuations in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

moderate to high range. There was no doubt that from this preliminary evaluation the sonar pod would perform satisfactorily with respect to base line noise levels, adequate overall gain ability, and dynamic range of echo signal return.

Initial Platform Checkout

Initial testing in a laboratory setting is mandatory for success in the field. However, the concept of mounting previously functional blocks of any system on a new testing platform needs research prior to taking the platform into the field. All major platform mounted systems were loaded into and onto their designated positions. Power was furnished by the on-board 12-volt lead-acid battery to operate all systems including the voltage inverter furnishing 120VAC to the computer and monitor.

Head position was checked and calibrated and preliminary tests of the system were performed. Several problems were identified and corrected. Radio control operation was verified with all on-board systems powered and functional. There was concern that the use of the Citizens Band Radio carrier frequency in the 27 MHz range might allow the introduction of interference from radiation generated by the computer or microcontrollers. This had the potential of causing problems with the platforms steering system. No such problems were encountered and after a few minor modifications all systems were ready for field testing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Open Water Field Testing

Problems and Solutions in Field Testing

There are many lakes in the North Louisiana area where the author is based. Since most of the terrain is relatively flat in this area of Louisiana it becomes difficult to find a body of water deep enough to echo in a true 3-dimensional manner. Even in deeper open water using a surface mounted platform rather than a true underwater vehicle will allow only sonar observation in forward and downward directions.

Observations at the steeper downward angles can return random sonar patterns pertaining to obstructions located at or near the lake bottom. Such structures are of no known geometric shape or pattern, thus making their echoes indeterminate in evaluating performance.

It was believed that the best approach to evaluating this sonar system was to choose a local lake with few obstacles close to the water's surface that can return echoes. If more than one field testing session was needed, the same lake could be used without having to travel long distances transporting equipment and personnel each time.

At such a location, the planned procedure was to allow the sonar to scan in a normal manner. The platform was to be moved in reference to a singular fixed obstacle while data were recorded. The recording host would, in such a test, be active in recording returning echoes due to bottom soundings

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and reflected surface scattering. This need not be a problem, since the data are first recorded and later presented to the processing host. The recorded data files can be "filtered" of all echoes that do not pertain to the location and range of the target object.

If only transducer number 1 in its lower tilt position has an unrestricted echo from the selected target, the processing host software can still identify the position and range of the target and observe its motion in the Water_Space array. It was assumed that if this approach worked well for a singular transducer there is no reason that the other transducers would not work in a true 3-D environment.

Procedures for Field Testing

The first criterion for a successful field test is the choice of location. The location must have at least enough water depth for one sonar beam to travel to the chosen target. This could be done in as little as 15 feet of unobstructed water. This would potentially give a range of greater than 100 feet to target. This should be enough range to keep the events in the data structure for a period long enough for evaluation.

Numerous passes on the target can be performed head-on, turning left, and turning right. Various rates of turn can be employed. The optimum target is a singular obstruction such as a bridge piling, power line pole, or tree. These objects for the most part represent vertical columns from the lake bed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

up past the waters surface. Echoes from objects at greater ranges may not be a problem.

Testing of Host Computer Program

Since the host computer program is not required to run in real-time operation for this study, the evaluation of processed sonar data becomes much easier. In a real-time use of this system, data cannot be both processed and evaluated without the subjective evaluation of a visual display system which is beyond the scope of this research.

As data are read inside the laboratory from a pre-recorded set of files created on the instrument platform, sonar structures are created inside the Water_Space array. These structures cannot be evaluated without suspending operation of the software to observe what is happening in the array. This is accomplished with the use of the Slice function in the host software.

First, data reading is suspended, then the user is prompted. At this time the user enters a depth. This depth figure is not the depth below the waterline where the data are taken, but a Z-plane slice 60 feet thick through the Water_Space array starting at the level specified for the slice and extending downward. While normal operation of the host program is repeatedly suspended, the operator may take multiple slices of the array.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Rather than generating a raster graphic image of the Water_Space array a file, previously described, is opened for output that captures the Water_Space array as a series of X and Y coordinate points. As the slice is created by scanning the array, any array element that is non-zero generates an X-Y pair of data for this file. Once the slice is finished this file contains all of the points that are non-zero in that slice. This file is later printed with the use of DataFit Version 6.0.10, a plotting software package by Oakdale Engineering, to generate a graphical depiction of the slice.

Field Test at Cypress Lake

Available Locations and Facilities

A boat dock with power outlet, a jon boat with trolling motor, and the assistance of three individuals was available on-site for testing. This first open water test of the instrument platform with sonar system was performed on January 24, 1999. The weather was mild and the maximum wind speed was estimated at less than 10 miles per hour.

The equipment was transported to a location on the Northeast shore of Cypress Lake, northeast of Shreveport, Louisiana. This lake was once Cypress Bayou before the damming of the bayou creating a man-made lake. Bathymetric mapping of this lake reveals that most of the lake is shallow with the exception of the old channel which can be deeper than 20 feet [29]. However, the channel is narrow and winding,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

making the use of this deeper part of the lake difficult. The lake region where the testing occurred was approximately 13 to 15 feet in depth.

Testing Procedure and Test Results

The equipment was attached to the instrument platform before launching in the shallow water of the pier. The launching went smoothly, but after launching in this shallow water, it was noticed that the sonar system was in contact with a submerged object or else the silty lake bottom. The platform was slowly maneuvered to a position where the radio-control system could operate safely. The propulsion was initiated, and the platform was moved out into deeper water.

Recording was started and a series of passes were made on an artificial target constructed from 4.5-inch outside diameter PVC pipe. This pipe was weighted at one end and placed in approximately 13 feet of water where it stood upright resting on the bottom for the duration of testing.

Since the recorded values are time-stamped by the systems computer clock, it was an easy matter to correlate the action of the platform to the data gathered. An observer recorded the beginning and ending of each pass, the angle relative to the target, and any other information that was pertinent.

Although the radio control system worked well, it proved to be less stable than expected in obtaining either straight tracking or slow turning as the platform approached the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

target. It is believed that as the trolling motor vectored its thrust left and right of the center line, the stern was free to rotate left and right while the platform was not as free to move forward due to the high drag of the centrally located sonar assembly. Later tests of the platform without the underside drag of the sonar pod assembly proved much more stable.

More stability and control was obtained by using the jon boat to tow the platform at its side. The jon boat's trolling motor provided more than enough forward velocity for the trial. Numerous passes on the target were made. However, due to the recording host software not indicating the number and position of generated events, little was known at the time about data taken.

Once this Cypress Lake trial was completed and the sonar was being examined for leaks it was noticed that the sonar head was not aligned as expected. Upon closer examination it was noticed that the sonar head was perhaps 30 degrees further left than normal. The presence of lake bottom silt indicated contact with the lake bottom. It was surmised that this contact must have occurred on initial launch of the platform and was present during all data recording. There was no way to visually inspect the sonar head during the testing due to the opacity of the water.

Upon analysis of the data no identifiable echoes were present for the approaches and turning that occurred in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tests. A wide ranging set of echoes were recorded that seemed almost random in nature. Upon correlation of these echoes with the recorded platform position at that time, echoes were breaking the threshold, generating events that could not have represented true obstacles.

After considerable thought as to the origin of these echoes from locations void of obstructions, their source became obvious. These were apparently reflections from fish as seen by the sonar head pointing away from the actual target. The threshold had been set to values lower than $10 which created crossings from the small echoes generated by marine life.

Modifications to Equipment and Procedures

Several things were learned on this first attempt to obtain data. It became obvious that thresholding is more critical than at first thought. Thresholds must be set high enough to avoid most fish and other marine life but low enough to see obstacles that would present a problem for possible collision with an AUV or submersible.

Along with the ability to adjust thresholds more carefully, there is a decided need for a method of monitoring returning echoes for frequency, range, and sonar head angle. Simply knowing that events are being recorded by looking at the file size of the files created is not enough. For this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reason a small modification was made in the recording software written in QuickBASIC.

This software modification was simple, but potentially valuable, in monitoring the echoes as they are generated. The RECORD.BAS program was left intact, with the exception of moving the actual recording function to a separate software module. This module, called LOGIT.BAS, clears the screen of the recording host and draws a line across the bottom and down the middle of the screen. Left side echoes are depicted on the left side of the screen with right side echoes depicted on the right side of the screen.

Characters that range from 0 to 5 are placed on the screen at locations that reflect the occurrence of target echoes. Locations for these character placements start at the lower middle of the screen, representing close targets, and fan out as they get to the top of the screen in ray-type patterns. The six rays that are generated represent the six horizontal angles of the sonar head. The range is not given in the length of these rays, but as the total Y-axis height above the screen bottom.

This visual depiction is simple, but useful, to see where objects are being sensed. The software filters all but the low head position from transducer number 1. This is believed to be the only usable combination for water as shallow as in the test conditions. The screen can be cleared at any time by striking the "C" key on the keyboard. Characters are placed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

directly over existing characters and must be removed periodically to observe new information being reported.

Field Test at Gold Point

Available Locations and Facilities

Gold Point is a peninsula surrounded by an oxbow lake of the Red River located approximately 7 miles North of Downtown Shreveport, Louisiana. The land encircled by the lake is owned by the John David Crow family of Shreveport, whose permission was obtained before using the facilities. There is a recreational home, boat launch, boat house, and pier on the property.

One measurement in the lake center at the southern end of the oxbow showed approximately 15 feet of depth. As a former portion of the Red River, this lake has no timber and few if any underwater obstructions. The water was clear enough to observe the action of the sonar head as it scanned. This was an advantage over the Cypress Lake test, as it constantly showed that the sonar head was aligned and functioning.

The instrument platform was transported to the site, along with a 14 foot jon boat equipped with a 32-pound thrust trolling motor. The sonar pod was mounted below the instrument platform which inturn was connected to the side of the jon boat for towing. Extension cords for power, CRT monitor, and keyboard were employed to allow the monitor and keyboard to be placed in the boat rather than on the platform.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Initial testing was performed using the recording host computer in terminal emulation mode. Several pings were transmitted and echoes observed on the screen in the form of hexadecimal return levels. The sonar pod worked without problem. However, when the pre-processor chain was connected and recording initiated there were too many echo returns recorded. Various thresholds were attempted with few positive results. Finally with the gain code set at 001, attenuation set at 180, pulse width set at 40, and threshold GP11 formulated and installed on-site, the system began generating echoes based on sub-surface features.

The first sets of recordings were made by motoring from the lake center to the shore lines. In some cases, piers and facilities were used and in other cases the opposite shore line, with its gradual slope upward, was used. It was clear that echoes were being recorded and being charted on the screen in patterns reflecting underwater geometries.

A second set of tests were performed using the PVC pipe target used on Cypress Lake. This 4.5-inch outside diameter pipe rested upright on the lake bottom in approximately 15 feet of water. Although echoes were reflected by this target, it did not perform as well as expected. Even at close ranges the sonar would apparently not record all reflections.

It is believed that the smooth round shape offered a low effective cross section to the sonar signal with most of the pipe reflecting the signal either left or right. Most

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

probably an object with a rough surface such as a tree or power line pole, would have scattered the signals making the detection much easier.

Approaches to this pipe target were made head-on with slow turning either left or right. Other tests were made by stopping the motion of the platform and allowing the sonar head to scan as the platform was rotated. This was done to test event rotation.

Since the water velocity indicator had never been calibrated, a procedure was performed to calibrate it. At the end of the sonar target runs, the boat and platform were steered into a path parallel to the boat docks to observe a time and distance measurement set. The boat dock area has a boat house supported by six large wooden pilings in a rectangular configuration. By sighting down the two end pilings, it is an easy matter to time the distance traveled between the ends of the boat house. Distance between these pilings was measured with a steel tape.

This procedure was performed with two passes in opposite directions. Assuming that wind would make a small difference in speed, the two times were averaged. These values for time were divided into the distance between the pilings to give velocity. This procedure was performed at both the slow and fast speeds of the trolling motor. The resulting average velocities were 1.25 feet per second for the slow speed setting and 2.13 feet per second for the high speed setting.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Some random and non-repeating echoes were noted at times even when the sonar was pointed directly out into the lake where no obstructions exist. It is believed that once again echoes from fish create these spurious and unpredictable sonar returns. A fish swimming perpendicular to the direction that the sonar transducer is pointed could provide a moderate sonar cross section. For the most part these echoes were not a problem in analyzing performance.

Equipment Considerations and Allowances

No problems were encountered with the sonar pod or head rotating assembly. The turbine water velocity indicator worked well initially, sending its velocity information in the form of counts per time interval back to the recording host computer. On long straight runs at the target this indicator provided stable constant level output. However, later analysis of the data revealed that this indicator had suddenly stopped turning roughly half-way through the testing period.

The cause of the water velocity indicator failure is believed to be aquatic vegetation that collected on the sonar pod and undercarriage. This turbine assembly was on the opposite side of the instrument platform and not visible to the operators. When the sonar system was later removed from the platform, much debris had collected. There was no apparent problem with the mechanism other than entanglement with the vegetation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It was the intention of the author to use actual observed velocity from the time and distance measurements and correlate this with recorded turbine speed. In this way the recorded velocity information could be automatically injected into the data furnished to the recording host computer. Since this could not be done due to the failure, the recording host program was modified to allow the entry of velocity at the beginning of each run. This velocity was calculated directly from the time and distance measurements for the speed setting used.

Selection of Transducer and Head Position

It was decided that the center transducer in the head-down position was the best choice for usable data. Since there was enough wind at the site to cause small wave action, the head-up position would have returned close echoes that would have had to be filtered out before presentation to the processing host computer.

Since every other sweep yields a head-up positioning and thus contributes nothing to the data, it was decided to allow the head to remain low and to use this data later with minor modification. With this approach the sonar head can yield data on each scan. The problem, however, is that the pre-processor pipeline software believes that the head is tilting up and down on every other pass.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The solution to this problem is simple. The head-up position is 4 degrees above the horizontal and the head down position is 4 degrees below the horizontal. The transducer used, sweep angle, and range is identical. When the look-up table located in processor B is entered the X, Y, and range values are identical for the target event. Only the Z value is incorrect, but only by its sign. If the head is down continuously but the software indicates a head-up position, changing the sign of the Z value, as recorded, reflects an accurate measurement and allows data on every sweep.

Filtering of Data Prior to Processing

The data gathered from the Gold Point field trial consisted of 13 file sets. Each set contained an event file (SN1_XX.Txt) and a velocity and heading file (SN2_XX.Txt). The "XX" values contain a unique computer-generated number used to keep one file from overwriting the other as data are taken. These files represent raw data which must undergo filtering before being renamed as Events.Txt and DirSpd.Txt for processing by the host computer.

The first step in filtering the data is to modify the carriage return and line feed character sequence to make the data readable by the processing host program. Also, the time stamps are made more concise by removing the word "TIME" from the data strings and placing the time stamp in a more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

accessible position for reading. No data is altered in any way by this process.

The next step in the filtering process is to remove all data not generated by the center transducer. Thus, the filter programs written in QuickBASIC must look at each event and remove events that do not have a "1" character in the transducer location. This shortens the total event file size considerably.

The last step in the filtering process is to search the event file to find head-up events for conversion to head-down events. Each event in the file that needs modification will have its head position changed in the event string. Also, the next sequential string containing the look-up table values will have a space representing the sign of the Z value. This is changed to a negative by insertion of a minus sign "-". This reflects the true head condition for all data recorded.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

CONCLUSIONS

Research Findings

Introduction

The system as a whole functions as intended. This research effort validates the proof-of-concept for this approach to sonar collision avoidance. Both hardware and software designs create a functional system, that can be used as a model for future development.

The only major limitations to this research were based on available locations for testing. This resulted in a more restricted approach to the testing of the full capabilities of the system. Location constraints on testing prevented the system from being evaluated in a true 3-D environment due to water depth and target availability.

Usable Data File Sets

Of the 13 file sets taken at Gold Point, six showed merit. The others did not contain enough echo information to construct a usable pattern. In some cases the straight-forward approach to the sonar target placed one sonar beam to the left and the other to the right of target with the low

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cross-section geometry of the target itself located at the 3dB down points for both beams. At large distances this resulted in what is believed to be a mix of actual target and fish echoes.

Two types of file data yielded the best results. The runs made at low speed toward either bare shore line or dock complexes resulted in the most echo returns and depicted structure at range. The other type of basic test that yielded good results was the stationary rotation test. This is where the boat and platform have little or no velocity, except for small amounts of wind drift. Here the platform was rotated slowly facing the PCV pipe target.

Methods of Data Presentation

Data runs lasted from less than one minute to several minutes in length. The presentation of this data will be based on a series of array slices taken during operation of the processing host software. A small number of slices are presented, taken at several second intervals. This allows the depiction of probability spheres in the Water_Space array created by incoming events.

When a slice is taken, the processing host software creates a file and loads the contents of each non-zero array sample as an X-Y pair separated by a TAB character. These pairs are used to create a scatter plot using DataFit software. All slices begin at the Z-equals-zero plane, but

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

actually represent any non-zero element as deep as 60 feet below this Z-equals zero-plane. This allows for a Z-axis projection which better defines the shape of all spheres below the Z-equals-zero plane.

The time of recording is furnished with each slice depicted. This time element is critical to the visualization of the process. These times also provide an easy reference for what each slice represents.

The plotted data requires numerous pages and thus is located for reference in Appendix D. The reference to each graph is given by type of observation and time.

Approach to Dock Complex

The first file set chosen for graphical presentation in this paper represents a slow speed run toward the dock complex consisting of the boat house and pier. This approach was not made head-on but rather at an estimated 20 degrees to the right of head-on relative to the shore line. The sonar instrument platform was aimed directly at the large corner piling of the boat house. Beyond this set of outer pilings, the sonar signals can travel under the boat house and strike the lake wall some 20 to 25 feet beyond.

This approach began at 02:27:39 with the starting of the recording host computer, and the platform being turned to heading. The event file, however, was set to start processing at 2:30:14. This was accomplished by temporarily removing the previous events from the file before the processing began. In

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

this way the events handled by the event table grow as time progresses. This allows the reader to more clearly observe the formation of the structures in the Water_Space array.

It should be noted that this depiction used a sphere longevity constant (EvLife) of 36 seconds. This is relatively long, but necessary to show movement of spheres in the Water_Space array over longer periods of time. Longer longevity of the spheres also is useful when some areas that are being sounded do not return solid echoes each time. In this way images can be constructed by integrating larger amounts of data.

Observing the data in Appendix D, one can see that at the 02:30:16 slice, only one sphere was present on the graph. This sphere was seen to have moved downward in the 02:30:21 slice by approximately 6 feet, about the distance traveled in this 5 second interval at a 1.25 feet per second velocity. Also, in this slice four more spheres have been generated roughly in a horizontal line.

In the 02:30:26, slice several new spheres are shown. One sphere in the upper left and another on the far right are clearly visible. Once again the pre-existing spheres have translated downward at a constant rate.

Slice 02:30:31 shows a very interesting development. At least one more sphere has joined the original horizontal line, but two new spheres can be seen forming a second horizontal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

line of sonar echoes, roughly 20 feet beyond the first line. Again, overall translation of existing spheres can be seen.

In slice 02:30:36, the left half of the original lower line shows some development from new events. A "thicker" characteristic can be easily seen in this region. Also, the new line in the upper right shows at least one new event.

The last slice shown, taken at 02:30:41, shows a further building in the top (newer) line along with continued translation of the original (lower) line. This clearly shows what looks to be two separate structural developments. This was puzzling to the author before realizing the nature of the dock and boat house complex.

It is believed that the first of the two separate structures occur from the outer pilings of the boat house and dock complex. As the platform closes on the dock complex, the sonar at these closer ranges had the resolution to see beyond the pilings and under the docks and boat house to the lake wall.

Zero Velocity Turning

This test was initiated to test the turning of the platform and its effect on sonar returns. This test was also successful in that it showed the best set of echo returns from the PVC pipe target. This test shows a gradual turning to the left that slows toward the end of the samples. Again, this is an excerpt taken from a larger file.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The processing was started at 03:06:48, with the first slice taken at 03:06:49. This slice shows a stationary event located approximately 50 feet in front of the sonar array and about 4 degrees to left of center. This was generated from event E112033 generated at 03:06:49. Slice 03:06:57 shows a slight rotation in a right hand direction to approximately the center position (zero degrees). The range for this slice is constant. No events occurred between these first two slices.

Slice 03:07:05 shows the next event, E112030, occurring at 03:06:58, and a third event, E11302F, occurring at 03:07:00. By this time, the first event has been removed from the event table. These two spheres were generated from events occurring at a 2-second interval from two adjacent sonar head angular positions.

Two more events occurred before the 03:07:13 slice was taken. These events are the only ones left in the table when this slice is taken. The two previous events have expired. Another event (E114029) occurs at 03:07:20 and is the last displayed event. When the last slice is taken at 03:07:21 two events remain in the system.

It is interesting to note the position of the sonar head when each of the events were recorded. Consider that the far left-hand position of the sonar head is angle "0" and the far right-hand position is angle "5" thus yielding 6 different angular positions. The recorded events show a sequence as the platform turns from right to left. This sequence is {2, 2, 3,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3, 4, 4, 5, 5} including the next two events that were not graphed. It is clear that each center transducer in the low tilt position captured the target echo twice in the order of the turning.

Although there are two examples of rotation in this sequence, the easiest example to see is the comparison between the slice at 03:06:49 and the next slice at 03:06:57. The rotation values can be confirmed by observing the DirSpd.Txt file for the times of these slices. At 03:06:49, the heading information showed an average value of 94 degrees. At 03:06:57, the heading information showed 90.25 degrees. The difference between these two angular positions is 3.75 degrees. Since rotation was toward the left, the array spheres should rotate to the right.

To better demonstrate rotation, the events file for this example was taken and truncated to represent only the first event (E112033) occurring at 03:06:49. A very high longevity was assigned to the processing host software and another run generated from the same DirSpd.Txt file was begun. This simply demonstrates the rotational process. A series of slices were taken and provided in Appendix D.

One can observe the rotation by viewing the sequence of slices. The amount of rotation averaged 17.4 degrees from the DirSpd.Txt file data for this period. The rotation from the graph measured with a simple protractor over the same period was 18 degrees. The small (0.6 degree) error can easily be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

explained as a combination of measurement error and quantization error.

It should be remembered that the actual magnetic heading is 180 degrees from this heading due to the deliberate installation of the Vector Compass Module in its present orientation. This has no effect on the changes in heading important to this data.

Comments on Results Presented

Although the data presented is a select sample of data taken, the sonar system is seen to operate as intended and to produce usable results. The processing host software serves as a usable model in proving that the concept is useful. Also, a software template for the processing host program has been established showing a workable approach to processing of collision avoidance sonar data.

Strengths of Current Approach

General Considerations

All aspects of the research findings demonstrate that this approach has merit in providing another dimension to collision avoidance for underwater vehicles. The goal of proof-of-concept was realized. It is believed that this research can lead the way to a much-improved system with higher resolution and better processing capability.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hardware Strengths

The sonar pod proved to be a robust design. The equipment remained water-tight and was shown to be rugged, except for the fact that the sonar head became displaced by a collision with an underwater structure during the Cypress Lake field test. The rotation mechanism proved capable of accurate transducer array positioning without the need for feedback, using a stepping motor and a fixed rotation limiting device. The sonar pod mounting frame physically was well-matched to the submersible attachment locations proposed as a future mounting point.

The relay multiplexer scheme had good utility. This was a major concern of the author when deciding to use relays rather than solid state switching. This approach proved to be a good choice due to the low resistance in the ON state and the almost infinite resistance in the OFF state. Relay life should be greatly extended because the switching only occurs with no voltage or current present. This was a good and workable approach.

Gain control provided a range of gains that were more than adequate for reception of week sonar return signals. The approach of providing both gain control and also attenuation allowed for a device that did not require re-design in any form due to results obtained in later testing. At optimum gain and attenuation settings, intrinsic noise levels were low, many times less than typical return signal levels.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The concept of using a microcontroller to control almost all aspects of the sonar pod also worked well. Gain settings could be controlled along with sonar pulse width in cycles. Another strength of the sonar pod design was the fact that all control over this system was maintained by a bi-directional serial port. The only other pod connections necessary were power. This self-contained modular design manifests a great deal of versatility in that it is easy to connect to almost any computer system.

The pre-processor system, consisting of processors A and B, worked well together, generating events with accurate information obtained from look-up table processes. The interface box housing these components was small enough to include in a submersible and used small amounts of power due to the CMOS components of both the MCUs and the 74HCT logic family employed.

The use of two separate host hardware systems proved to be a good choice. Data gathered by the recording host computer could be processed again and again with the processing host computer. This was a strong approach to evaluating the overall system.

Software Strengths

The entire software system from sonar pod through processing host computer was based on modular design. Each software sub-system performed a dedicated task, with the exception of the pre-processing chain which used two separate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

MCUs with programs closely associated by both hardware and software.

The sonar pod used a primitive command structure with simple instructions associated with numerical parameters to facilitate all necessary pod operations. The simplicity of the sonar pod's software is its major strength. This simple command structure makes it easy to interface with almost any system possessing the ability to generate strings and transmit them using a serial interface channel.

The EPROM look-up table used in the pre-processor chain proved to be a good approach to saving processor time for the processing host computer. The interface between processor A and processor B, although a complex one, proved to be reliable in every test.

In regard to the host computer, the choice of C/C++ language was sound. It is not known how much execution speed was gained by not developing objects. However, two professionals queried supported the approach to obtaining faster execution speed by not developing classes and objects as found in C++ [1,11]. Also, the question of re-writing for a C platform makes the author believe that the chosen approach constitutes an overall strength for this system.

Using a Dell Dimension XPS Intel Pentium 100MHz computer as the processing host proved more than adequate for construction and mobility of probability spheres through the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Water_Space array. Speed was no problem for this relatively slow computer as viewed by current computer speed standards.

Weaknesses and Methods of Improvement

Threshold Adjustments

The most serious problem for this system is determining the best threshold for a given water and terrain scenario. This would have been a problem at the Cypress Lake test even if the sonar head had not been misaligned. It was also a major problem at the Gold Point test site.

In theory the use of a threshold look-up table is a sound concept. In practice it is slow and cumbersome to try various thresholds one at a time. The operator needs to see the sonar return amplitudes for select pings to determine the threshold levels needed. This requires a reorientation of the electronics on the platform used and takes much time and effort. When one is in a boat located in the middle of a lake performing this re-arrangement, along with attempting new threshold generation and installation, it becomes a formidable task.

There are several approaches to solving this problem. One approach conceived earlier but not researched would be to use a capacitor and resistor to generate an exponential curve that could be varied in both time constant, initial value, and initial start time. The output of this circuit could be triggered with software from processor A and placed on the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

input of one of the other analog to digital inputs of the 68HC711 MCU. This real time set of values following the actual exponential loss in the sonar return could be adjusted in real time and used to easily set threshold. There are many ways of adjusting threshold that would prove easier than was used for this research.

Underwater Currents

One weakness that has not been addressed previously is the inability of the present system to deal with underwater currents. These currents can be neglected in many cases due to their low velocities. However, ocean currents measured in the Gulf Stream can be as high as 2.5 meters/second [15]. Even with smaller currents it is easy to see how large errors in estimated position can occur.

The effects of underwater currents can be compensated for with simple resources from the onboard host computer. Currents can be represented by horizontal translations with no turning. If both the direction and velocity of current are known these can be easily factored into the computations by treating them as translations with both an X and Y component. Thus, at the time that translations are made to update the event table, translational values for water currents can also be introduced.

Unknown currents can be identified by comparing the forward velocity of the vehicle, as measured by the velocity sensor, with values obtained from doppler sonar which can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

depict velocities relative to the floor of the lake or ocean. This is somewhat similar to an airplane flying in windy conditions. The pilot knows the aircraft heading and velocity relative to the air itself. If a ground based radar operator gives the plane's direction and speed relative to the ground it is theoretically possible to calculate the velocity and direction of the wind.

Turning Error Compensation

When an AUV or submersible turns, it has some lateral motion. This can be considered as a velocity component perpendicular to the longitudinal axis of the vehicle. This effect can be small or pronounced depending on the many factors to be considered for a given submersible or AUV design. If this effect is not pronounced, there should be little or no need for compensation in the computer software processes. In many cases the sub can be considered as moving in the direction that the sonar unit is pointed (forward).

Turning error can be sensed by using a vane type water direction sensor. This simple device, similar to a weather vane, could be attached to an angular displacement sensor to translate angular information back to the processing host computer. Here simple procedures based on trigonometry could take the velocity magnitude value and break it down into its components for processing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sweep Angle

It is believed by the author that a slightly wider sweep angle could prove beneficial. Currently the sweep covers 48 degrees of total forward looking angle. A modification of another 8 degrees of sweep would place the center position looking straight forward with three positions on each side of center. This modification would be easy to perform. However, it would, along with other software and hardware modification, require the recalculation of the EPROM data look-up table for processor B.

The head tilting mechanism worked well but will prove to be more difficult on a submersible installation, probably requiring a hydraulic or pneumatic system for driving the head tilt cylinder. An array of six rather than three transducers would have been a better approach in eliminating the need for the cylinder and its drive system. One reason this approach was not used in the current design was that with each transducer being 2.25 inches in diameter, the total array height would have been 14 or more inches high. The transducers could have been placed in a cluster with some mounted side-by-side. The cluster could then be swept side to side and data taken with no tilt axis.

Pre-processing Versus High Speed Host

Although the pre-processor pipeline worked well in this research, it added some extra complexity in the form of two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

embedded microcontrollers. These microcontrollers could be replaced by a faster host computer. This would made the overall installation smaller and less complicated. The look-up of X, Y, and Z coordinates can still be employed to insure fast conversions. With computer speeds increasing each year it would seem a logical approach to eliminating this extra component of the system.

Another approach to system improvement would be to employ dual processor computation. Many of these systems have two separate Intel Pentium processors. Here speed would be greatly increased as one processor could do the job of the pre-processing chain and at the same time perhaps share its resources with the second processor doing data structure updating.

Suggestions for Future Research

Visual Display Technology

One logical extension of this research is the application of the data structures generated by this approach to a 3-D or quasi-3-D sonar collision avoidance display for a submersible. It is obvious that this research ends when the data is located inside the Water_Space array. But obstacles inside an array are useless unless the submersible pilot can visualize the objects with perception and understanding. Perception can involve other senses as well. Audio in the form of a verbal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

machine generated speech to warn of collision is a possibility.

It has been suggested to the author that research into the psychology of visual perception would be a logical extension to this research. This is a formal area of expertise outside the realm of the author's current knowledge.

Upgraded Transducer Capability

More research can be pursued using more and higher resolution sonar transceivers. However, higher resolution requires larger quantities of memory to depict the data structures. With increased data structure size, speed of computation will decrease requiring a faster host computer.

Research aimed at pushing the limits of current technology in this area would be expensive and require better laboratory facilities than those available to the author. The ground work has been laid for this type of research by establishing proof-of-concept in the creation of a working model. Also, it should be mentioned that the current level of resolution and computational ability is sufficient for many current applications.

Z-Axis Rotation and Translation

This research utilized a single left-right rotation axis and a forward translation capability to update events in the data structure. The logical extension to this research is to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

include an up-down (Z-axis) rotation ability along with translation in this axis. This should not prove to be a difficult task, using the concepts currently put forth in the host software.

It should be noted that there are some basic differences between existing rotation and translation operations in the current software compared to what would be necessary for extending these operations to the Z-axis. From a vertical angular frame of reference, submersibles seldom point in the same direction as they move through the water. This can cause more complex considerations for Z-axis inclusion in the sonar calculation scenario.

Most submersibles use bow and/or stern planes. The control surfaces push the submersible upward or downward as it moves through the water. When the submersible is diving for example, most subs tilt down-bubble but actually descend at an angle steeper than the down-bubble angle. Some vehicles use changes in ballast and keep a level bubble while ascending or descending. Thus, the Z-axis considerations would have to extend to both pitch considerations for rotation of data elements and also vertical translation not directly related to pitch.

Vertical translation information is very easy to obtain. As the vehicle changes vertical position water pressure changes with depth. Some submersibles and most AUVs have electronic depth sensing built into their systems. There are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

also numerous sensors that are currently available for indicating pitch angle. Little modification to the existing software would be needed for these enhancements to facilitate movement in the vertical plane.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX

MICROCONTROLLER SOFTWARE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
*  PROCESSOR (A)
*
            OPT    c
* Gary R. Boucher SONAR PRE-PROCESSOR A (Closest to SPMM)
*
* Written by Gary R. Boucher
*
* Equates Section
PORTA      EQU       $1000      Port A
PORTB      EQU       $1004      Port B
PORTC      EQU       $1003      Port C
PORTD      EQU       $1008      Port D
DDRC       EQU       $1007      Data Direction for C
DDRD       EQU       $1009      Data Direction for D
PACTL      EQU       $1026      For A7 direction
SPCR       EQU       $1028      SPI Control Register
SPSR       EQU       $1029      SPI Status Register
SPDR       EQU       $102A      SPI Data Register
BAUD       EQU       $102B      SCI Baud Rate Register
SCCR1      EQU       $102C      SCI Control Register 1
SCCR2      EQU       $102D      SCI Control Register 2
SCSR       EQU       $102E      SCI Status Register
SCDR       EQU       $102F      SCI Data Register
BPROT      EQU       $1035      Block Protect for EEPROM
OPTION     EQU       $1039      Option Register
ADCTL      EQU       $1030      A/D Control Channel
ADR1       EQU       $1031      A/D Byte 1
STACK      EQU       $01FF      Stack Pointer
RAMST      EQU       $0000      Start RAM Threshold Msk
FBTE       EQU       10         Feet Spaced for Event Generation

CTBTW      EQU       4          Min Distance Between Pings
AB_MAX     EQU       8          Time > Thresh Before Auto Event

           ORG       $0000

* RAM Variables Section
SPR_BUF:   RMB       64         SPI Receive Buffer
SPR_PTR:   RMB        2         SPI Receive Buffer Pointer
SPT_BUF:   RMB       64         SPI Transmit Buffer
SPT_PTR:   RMB        2         SPI Transmit Buffer Pointer
SP_DRDY:   RMB        1         SPI Data Rdy (Input Buf Loaded)
DELCYC:    RMB        1         Size of Delay 1=10mS, 2=20mS ETC.
SDELY:     RMB        2         333=1mS Delay Setup Variable
TEMP:      RMB        2         General Use Ram Space
COUNT:     RMB        2         Argument for Commands
FUNCTN:    RMB        1         What Function is Happening
ANGLE:     RMB        1         0=Far Left, 1=8, 2=16,...
HEAD_POS:  RMB        1         (0) = Up, (1) = Down
ERR_TYPE:  RMB        1         Error Type
ECHO_H:    RMB        1         (1) If Echo Has Been > Thresh
EVEN_CT:   RMB        1         Count for Event Eligability
DN_ERR:    RMB        1         Done Error Flag
STOR_PT:   RMB        2         RAM Storage Pointer
RETV_PTR:  RMB        2         RAM Retrevial Pointer
AB_CTR:    RMB        1         Time Above Threshold Count
CX_CTR:    RMB        1         Time Since Crossing Count
LOW_HI:    RMB        1         Last Threshold Lever Indicator
TRAND:     RMB        1         Transducer Number for Events
H_POS:     RMB        1         Recorded Head Position
ANGL:      RMB        1         Recorded Angle Information
NBR_EV:    RMB        1         Number of Events Occured / Thresh
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* Below are the Display Variables.
VAR:      RMB      1          Screen Line Counter Variable
C1:       RMB      1          Count Variable (2 Bytes Actually)
C2:       RMB      1          Lower Order Byte of Count Var
LV:       RMB      1          Sonar Echo Return Level Variable
TH:       RMB      1          Threshold Variable
EVL:      RMB      1          ' '=No Event, 'E' is Event


          ORG      $B600


START:    LDS      #STACK     Set Stack Top
          LDAA     #$93       Bit to turn on ADPU Bit (Pump)
          STAA     OPTION     Store it
          LDAA     #$00       For EEPROM Block Protect
          STAA     BPROT      Make EEPROM Writable from BUFFALO
          JSR      INIT       Init the System
          JMP      MAIN       Start Looking at Commands

**************************************************************
***** M  A  I  N     L  I  N  E     P  R  O  G  R  A  M ****
**************************************************************
          ORG      $B630

* EEPROM Constants
TOT_CNT:  FDB      #636       How Many Char Before Check DN
DAT_CNT:  FDB      #600       Count of Total Ping Data Hex
SCR_TST:  FCB      0          1=Display Scrn Test Info, 0=No
EVNT_PP:  FCB      4          Events Per Ping (Maximum Number)
SH_DEL:   FDB      333        DELY Value = 1mS Used as SDELY
PING:     FCB      1          0=No Ping Test, 1=Normal Operate
FAST_BR:  FCB      $01        62.5K Baud Rate (Fast Mode)

          ORG      $B640

* [ HEAD_UP ]
* Sets PB7 High to engage relay and send 12 volt power to
*  pneumatic control valve.  This raises the sonar head.
HEAD_UP:  CLR      HEAD_POS   Current Head Position Indicator
          PSHA                Save Register A
          LDAA     PORTB      Get PORT B
          ORAA     #$80       Make PB7 = 1
          STAA     PORTB      Store Back in A
          PULA                Restore A
          RTS

* [ HEAD_DN ]
* Clears PB7 so as to disengage the relay and kill 12 volt power
* to the control valve.  This lowers the head.
HEAD_DN:  CLR      HEAD_POS   Current Head Position Indicator
          INC      HEAD_POS   Make Indicator = 1
          PSHA                Save Register A
          LDAA     PORTB      Get PORT B
          ANDA     #$7F       Mask to Make PB7 =0
          STAA     PORTB      Store Back in Port B
          PULA                Restore A
          RTS
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
************************************************************
***********  S  U  B  R  O  U  T  I  N  E  S  *************
************************************************************

          ORG       $D000      EPROM Memory

* [ INIT ]
* This routine does an initialization of the system.  It is
* called only once upon RESET or POR.
INIT:     LDAA      #$90       Initial Byte for Port A
          STAA      PORTA      Disables *WE and *OE
          LDAA      PACTL      Get Control Register
          ORAA      #$80       Make A7 Output (Initially = 1)
          STAA      PACTL      Store Back
          CLR       PORTB      Make ALL PB Outputs Zero
          CLR       PORTC      Zero Before Setting Direction
          LDAA      #$3A       Set Up Data Direct for SPI, SCI
          STAA      DDRD       Store Data Direction Register
          LDAA      #$54       SPE=1,MSTR=1,COPL=0,CPHA=1,E/2
          STAA      SPCR       Set Up SPI Channel
          CLR       ANGLE      Make Angle Zero
          CLR       ECHO_H     Start with Echo has Been < Thresh
          LDD       #SPR_BUF   Address Fst Byte of SPI Receiver
          STD       SPR_PTR    Store in Pointer Location
          LDD       #SPT_BUF   Address Fst Byte of SPI Trans Buf
          STD       SPT_PTR    Store in Pointer Location
          CLR       SP_DRDY    SPI Data Ready Cleared
          LDAA      #$30       Code for 9600 Baud Rate
          STAA      BAUD       Place in Baud Register
          CLR       SCCR1      8 Bit Data, No Bit 8 Etc.
          LDAA      #$0C       Enable Transmitter/Receiver
          STAA      SCCR2      SCI Control Register II
          CLR       ANGLE      Clear Angle Position
          JSR       HEAD_UP    Start With Head Upward
          CLR       ERR_TYPE   Type of Error Flag
          CLR       DN_ERR     Clear Done Error Flag
          LDAA      #'S'       (S)top is Default Condition
          STAA      FUNCTN     Store it in Function
          LDAA      #20        Get Ready for 0.2 Second Delay
          STAA      DELCYC     Delay Variable (DELCYC * 10mS)
          JSR       DELAY      Go Delay
          RTS


* This is the MAIN body of the program for Processor A.
* MAIN is NOT a subroutine.  Once Initialization is acheived
* this main-line program is vectored to where execution remains
* until the system is shut down.
MAIN:     LDAA      PORTA      Get Handshaking Bits
          ANDA      #$01       Look at PB0 (RTS From P_B)
          BEQ       TST_RDY    Branch if Not High
          JSR       RECEIVE    Go Get it!
TST_RDY:  TST       SP_DRDY    (1) if New Data String, (0) if Not
          BNE       NW_DATA    New Data if SP_DRDY = 1
          LDAA      FUNCTN     Get Existing Function
          CMPA      #'R'       Is it a (R)un Condition?
          BNE       MAIN       If Not 'R' Then Go MAIN
          JSR       RUN_SNR
          BRA       MAIN
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
NW_DATA:  CLR       SP_DRDY   Reset the RDA Flag
          LDX       #SPR_BUF  There was New Data. Point to it
          LDAA      0,X       Look at First Character of Data
          CMPA      #'X'      Exit to BUFFALO Character
          BNE       NOT_BUF   If Not 'X' Then Not BUFFALO
          JMP       $E000     Bit User Fast Friendly Aid :-)
NOT_BUF:  CMPA      #'<'      (<)=Proc_A Cmd, (:)=Sonar Control
          BEQ       LSTHAN
          JMP       CHK_COL   If Not P_A Cmd - Check for Colon
LSTHAN:   LDAA      1,X       Look at Strings Second Character
          CMPA      #'I'      Is it an INIT Command?
          BEQ       LEGAL     Branch if 'I'
          CMPA      #'C'      Is it a CENTER Command?
          BEQ       LEGAL     Branch if 'C'
          CMPA      #'U'      Is it an UP HEAD Command?
          BEQ       LEGAL     Branch if 'U'
          CMPA      #'D'      Is it a DOWN HEAD Command?
          BEQ       LEGAL     Branch if 'D'
          CMPA      #'B'      Is it an BOOT BUFFALO Command?
          BEQ       LEGAL     Branch if 'B'
          CMPA      #'F'      Is it Fast Baud Rate?
          BEQ       LEGAL     Branch if 'F'
          CMPA      #'R'      Is it a RUN Command?
          BEQ       LEGAL     Branch if 'R'
          CMPA      #'S'      STOP?
          BEQ       LEGAL     Branch if 'S'
          CMPA      #'P'      Point to AUX RAM?
          BEQ       LEGAL     Branch if 'P'
          CMPA      #'T'      Transfer to RAM?
          BEQ       LEGAL     Branch if 'T'
          CMPA      #'e'      Error Report
          BEQ       LEGAL     Branch if 'e'
          JSR       LCM_ERR   Legal Command Error Handler
          BRA       MAIN      Branch to MAIN
* If at legal function we have a good FUNCTN code.
LEGAL:    STAA      FUNCTN    It was "One of the Above"
          CMPA      #'R'      If 'R' then Run Just GoTo Main
          BEQ       MAIN      Main Continues Running if 'R'
          CMPA      #'S'      Could it be a (S)top Command?
          BEQ       MAIN      If Stop then Main

* At this point there is a NEW Function other than 'R'.  This
* function will be executed only once.  The below tests will
* determine which function is to be executed and how.

* Test for Initialize
TRY_I:    CMPA      #'I'      INITIALIZE?
          BNE       TRY_C     Try Center Command
* Initialize the system here.  Rotates to extreme left
* position, angle=0, head up, and stops action.
          JSR       ROT_STP   Rotate Fully Left
          JSR       HEAD_UP   Point Head to UP Position
          CLR       ANGLE     Make Angle Zero (Far Left)
          JSR       STOP      Put in Stop Mode
          JMP       MAIN      Continue

* Test for Center.
TRY_C:    CMPA      #'C'      Was it a CENTER?
          BNE       TRY_U     Try UP Command
* Center the Sonar Head.  Rotates to extreme left then
* rotates right 20 degrees to center position.  Head raised
* and system stopped
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          JSR     ROT_STP   Rotate Till Stopped at Left
          LDX     #CS2      Right 8 Degrees
          JSR     MESSAGE   Send it
          JSR     CHK_DONE  Check for Completion
          LDX     #CS2      Right 8 Degrees Again
          JSR     MESSAGE   Do it
          JSR     CHK_DONE  Check Completion
          LDX     #CS4      Right 4 degrees
          JSR     MESSAGE   Send it
          JSR     CHK_DONE  Check for Completion
          JSR     HEAD_UP   Point Sonar Head Upward
          JSR     STOP      Stop All Actions
          JMP     MAIN      Continue

* Test for raise head position to UP.
TRY_U:    CMPA    #'U'      Head UP?
          BNE     TRY_D     Try Head Down
* Raise head position to UP here. All action stopped.
          JSR     STOP      Stop All Actions
          JSR     HEAD_UP   Point Head Upward
          JMP     MAIN      Continue

* Test for lower head position to DOWN.
TRY_D:    CMPA    #'D'      Head DOWN?
          BNE     TRY_B     Try Boot System
* Lower head position to down.  All action stopped.
          JSR     STOP      Stop All Actions
          JSR     HEAD_DN   Point Head Downward
          JMP     MAIN      Continue

* Test for Boot BUFFALO.
TRY_B:    CMPA    #'B'      BOOT System?
          BNE     TRY_F     Try New Baud Rate
* Boot BUFFALO here.  CR and LF Sent after which a delay
* of 10mS is used before the G B600 message is sent.
          LDAA    #$0D      Get RETURN Character
          JSR     OUTPUT    Send it Out
          JSR     DELAY     Delay 10mS
          LDX     #CS9      'G B600' Message
          JSR     MESSAGE   Send Message
          JSR     STOP      Stop All Actions
          JMP     MAIN      Continue (No Error Check)

* Test for Fast Baud Rate.
TRY_F:    CMPA    #'F'      Fast Baud Rate?
          BNE     TRY_P     Try Point Command
* This is Fast Baud Rate entry point.  This changes the
* P_A to Sonar Pod Baud Rate to be increased to the value
* of FAST_BR located in EEPROM.
          LDAA    FAST_BR   Get the New Baud Rate
          STAA    BAUD      Store in Baud Register
          JSR     STOP      Stop All Operations
          JMP     MAIN      Continue (No Error Checking)

* Test for point to Threshold Memory (RAM)
TRY_P:    CMPA    #'P'      Threshold Point Command?
          BNE     TRY_T     Try a 'T'
* Point STOR_PT to start of RAM containing Threshold. This
* is for the purpose of loading the Threshold RAM with new
* values using the (T)ransfer command.
          LDX     #$3000    Point to RAM Area
          STX     STOR_PT   Store RAM Thresh Pointer
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          JSR       STOP      Stop Operations
          JMP       MAIN      Continue (No Error Checking)

* Test for Transfer to Threshold Memory.
TRY_T:    CMPA      #'T'      Transfer to Thresh Mem?
          BNE       TRY_e     Try an (e)rror report

* Transfer is made to Threshold Memory (RAM).
          PSHX                Store X
          LDX       #SPR_BUF+2  Point to First Data
TT1:      LDAA      0,X       Get Threshold Byte
          CMPA      #$0D      Is it a RETURN
          BEQ       TT2       If RETURN Finish
          JSR       STORE     Store Data Byte
          INX                 Bump Storage Pointer
          BRA       TT1       Get Another Byte
TT2:      PULX                Restore X
          JSR       STOP      Stop All Operations
          JMP       MAIN      Continue (No Error Checking)

* Test for (e)rror Report.
TRY_e:    CMPA      #'e'      Is it an Error Report?
          BNE       EXPAND    Room for Expansion in Comds
* Error reporting routine.  When this routine is called it
* sends to P_B the contents of ERR_TYPE.  This is done by
* sending an 'e[' followed by 8 zeros or ones and then
* ']'<CR><LF>  Example:  e[01010010]<CR><LF>.
          PSHA                Save A
          PSHB                Save B
          LDX       #SPT_BUF  Point to Transmit Buffer
          LDAA      #'>'
          STAA      0,X
          INX
          LDAA      #'e'      Identify as (e)rror Rpt
          STAA      0,X       Store 'e' in Trans Buffer
          INX                 Bump Buffer Pointer
          LDAA      #'['      Get Hard Bracket
          STAA      0,X       Send to Buffer
          INX                 Bump Buffer Pointer
          LDAB      #8        Get Ready for 8 Rotations
ROT_ET:   ROL       ERR_TYPE  Rotate Error Type Variable
          BCS       BIT_ONE   If Carry=1 Send a '1'
          LDAA      #'0'      If Carry=0 Send a '0'
          BRA       RT_AGN    Look at Rotation Count
BIT_ONE:  LDAA      #'1'      Must Have Carry=1
RT_AGN:   STAA      0,X       Place in Transmit Buffer
          INX                 Bump Buffer Pointer
          DECB                Bump Down the Rot Count
          BNE       ROT_ET    If Not Finished Rot Again
          ROL       ERR_TYPE  Rotate Back to Origional
          LDAA      #']'      Finish with Hard Bracket
          STAA      0,X       Place into Buffer
          INX                 Bump Buffer Pointer
          LDAA      #$0D      RETURN Character
          STAA      0,X       Place Last Character in Buff
          JSR       TRANSMIT  Send Buffer Contents
          PULB                Restore B
          PULA                Restore A
          JSR       STOP      Stop All Operations
* This section resets the PB3 output that represents an
* error.  Also all error flags are cleared.
          LDAA      PORTB     Get Port B
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ANDA      #$F7      Mask Off PB3
          STAA      PORTB     Store PB3=0
          CLR       DN_ERR    Clear Error
          CLR       ERR_TYPE  Clear Error Type
          JMP       MAIN      Loop At Main

* Location for future expansion.  If no more commands then
* just loop to MAIN.
EXPAND:   JMP       MAIN      Loop Again

* CHK_COL is a segment of the MAIN program that is vectored
* to if the '<' command is not found.  This area of the
* program checks to see if a Colon denoting a :XXXXXX type
* command is present.
CHK_COL:  CMPA      #':'      Maybe First Char was a Colon
          BEQ       COLON     Branch if Colon
          JSR       BFC_ERR   Bad First Character Error
          JMP       MAIN
COLON:    LDAA      0,X       Get One Char of String
          JSR       OUTPUT    Send it to Output
          INX                 Bump String Pointer
          CPX       #SPR_BUF+62   Compare to Extreme End
          BEQ       ADCR      If Extreme End - Add RETURN
          CMPA      #$0D      Was the Last Char a RETURN?
          BNE       COLON     If Not Finished - Loop
          BRA       ADLF      Just Add a Line Feed Character
ADCR:     LDAA      #$0D      Add RETURN or Another RETURN
          JSR       OUTPUT    Send RETURN Char to Output
ADLF:     LDAA      #$0A      Line Feed Character
          JSR       OUTPUT    Send LF to Output
          JSR       CHK_DONE  Check for Completion
          BEQ       CON1      If No Error - Continue
          JSR       COS_ERR   Colon Type Statement Error
CON1:     JMP       MAIN      Error!  Jump MAIN

* This subroutine loads an 'S' as New Function to Perform.
STOP:     LDAA      #'S'      Sets Stop Function
          STAA      FUNCTN    Store Function
          RTS


* [ ROT_STP ]
* Rotate to Stop Subroutine.  Rotates to Left Stop.
ROT_STP:  LDAB      #5        Set Up for 500 Left Rotations
RTS0:     LDX       #CS1      Point to L100 Command
          JSR       MESSAGE   :L100 Command
          JSR       CHK_DONE  Are We Done?
RTS1:     DECB                Lower 100's Count
          BNE       RTS0      L100 Again
          RTS

* Legal Command Error Sub Entry Point.
LCM_ERR:  PSHA                Save Register A
          LDAA      #$01      Bit 0 = Command Error
          BRA       ERR_HNDL  Handle the Error

* Bad First Character Sub Entry Point.
BFC_ERR:  PSHA                Save Register A
          LDAA      #$02      Bit 1 = Bad First Char Error
          BRA       ERR_HNDL  Handle the Error
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* Receiver Overrun Error Sub Entry Point.
ROR_ERR:  PSHA                  Save Register A
          LDAA      #$04        Bit 2 = Receiver Overrun Err
          BRA       ERR_HNDL    Handle the Error

* Done Error Handler Sub Entry Point.
DON_ERR:  PSHA                  Save Register A
          LDAA      #$80        Bit 7 = Done Error Flag
          BRA       ERR_HNDL    Handle the Error

* Colon Statement Error Sub Entry Point.
COS_ERR:  PSHA                  Save Register A
          LDAA      #$08        Bit 3 = Colon Statement Flag
          BRA       ERR_HNDL    Handle the Error
* Sets Error condition and stores type of error in ERR_TYPE
ERR_HNDL: ORAA      ERR_TYPE    Set Correct Error Bit in ET
          STAA      ERR_TYPE    Store Back in Error Type Var
          LDAA      PORTB       Load Control Lines
          ORAA      #$08        Make Error Indicator High
          STAA      PORTB       Store Back in Port B
          PULA                  Restore A
          RTS


* [ RUN_SNR ]
* This RUNS the Sonar System.  Entering here takes up where we left off
* previously.  One Cycle here Pings (3) times, records, and thresholds
* the data.  Events are sent to Processor (B).  If the head is at the
* far Right position then it is moved back to Left most position.
RUN_SNR:  LDX       #$0000      Point to First Block in RAM Memory
          STX       STOR_PT     Storage Pointer
          LDX       #CS5        Ping Command for Top Sonar Module ($01)
          JSR       PRC_PING    Process the Ping Info
          LDX       #$1000      Point to Second Block in RAM Memory
          STX       STOR_PT     Storage Pointer
          LDX       #CS6        Ping Command for Mid Sonar Module ($02)
          JSR       PRC_PING    Process the Ping Info
          LDX       #$2000      Point to Third Block in Memory
          STX       STOR_PT     Storage Pointer
          LDX       #CS7        Ping Command for Bot Sonar Module ($04)
          JSR       PRC_PING    Process the Ping Info (Move Head)
          LDAA      HEAD_POS    Get Head Position for Recording
          STAA      H_POS       Save the Current Head Position
          LDAA      ANGLE       Look at Current Angle
          STAA      ANGL        Record for Later
          INCA                  Increment Angular Position
          STAA      ANGLE       Store Back in Memory
          CMPA      #6          Far Right Position?
          BEQ       SWG_HD
          LDX       #CS2        Rotate Right 8 Degrees (71 Steps)
          JSR       PRC_ROT     Complete the Rotate
          BRA       PROCESS     Continue
* Swing the head fully left.
SWG_HD:   LDAB      #5          Needs to Rotate (5) Times Left
ROT_L:    LDX       #CS3        L100 Command (Rotates 8 Degrees Left)

          JSR       MESSAGE     Send Command to Sonar
          JSR       CHK_DONE    Check To See if Complete
ROT_AGN:  DECB                  Count Down Rotation Number
          BNE       ROT_L       Branch if Not Yet Finished
          CLR       ANGLE       Reset Angle Variable to Zero
          TST       HEAD_POS    Head Position (0=Up, 1=Dn)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          BNE        HEAD_U     Branch if Current Pos is Down
          JSR        HEAD_DN    Current Pos is Up.  Move Down
          BRA        PROCESS    Finished Now
HEAD_U:   JSR        HEAD_UP    Move Head Up
* Process takes data stored in memory for all three pings and
* compares it against a threshold template also contained in
* RAM memory.  Events are generated and transferred to P_B
* where the event information is obtained from an EPROM look-
* up table.
PROCESS:  LDX        #$0000     Point X to Ping (1's) Data
          LDY        #$3000     Point Y to RAM Stored Threshold
          LDAA       #'0'       Label Transducer
          STAA       TRAND      Transducer #0
          LDAA       EVNT_PP    Get Max Nbr of Events/Thresh
          STAA       NBR_EV     Store for Counting Down to 0
          JSR        THRESH     Go Check Against Threshold
          LDX        #$1000     Point X to Ping (2's) Data
          LDY        #$3000     Point Y to RAM Stored Threshold
          LDAA       #'1'       Label Transducer
          STAA       TRAND      Transducer #1
          LDAA       EVNT_PP    Get Max Nbr of Events/Thresh
          STAA       NBR_EV     Store for Counting Down to 0
          JSR        THRESH     Go Check Against Threshold
          LDX        #$2000     Point X to Ping (3's) Data
          LDY        #$3000     Point Y to RAM Stored Threshold
          LDAA       #'2'       Label Transducer
          STAA       TRAND      Transducer #2
          LDAA       EVNT_PP    Get Max Nbr of Events/Thresh
          STAA       NBR_EV     Store for Counting Down to 0
          JSR        THRESH     Go Check Against Threshold
          RTS


* [ THRESH ]
* This is the subroutine where the data elements and threshold
* elements are compared to see if and when the data is greater
* than the threshold.  The variable COUNT represents the Range
* count.  The AB_CTR is a variable that counts the number of
* feet of range that are continiously above the threshold.
* Multiple Events are generated if echo data values remain avove
* the threshold level.  The CX_CTR keeps repeated close crossings
* from triggering a multitude of Events.  This count spaces the
* occurance of Events where frequent crossings are seen.
THRESH:   CLR        COUNT      Range Count High Byte
          CLRA                  Make A Zero
          INCA                  Make A (1)
          STAA       COUNT+1    Range Count Low Byte (1 to Start)
          CLR        AB_CTR     Above Threshold Counter Var
          CLR        CX_CTR     Threshold Crossing Count
          CLR        LOW_HI     Last Level Indicator
          TST        SCR_TST    Display Scrn Info? 1=Yes
          BEQ        THRESH1    If 0 Then Branch Around
          LDAA       #22        Lines of Display Before Rest
          STAA       VAR        Counts Lines of Display -> 0
AGAN:     JSR        INCHAR     Console Input Routine
          CMPA       #'~'       Hard-To-Hit Char to Continue
          BNE        AGAN       Keep Looking Till Character
THRESH1:  LDAA       #' '       No Event Tag - Default
          STAA       EVL        Store Tag
          PSHX                  Save X Temporarily
          LDX        COUNT      Get Count in X
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          STX     C1        Save in Display Variable
          PULX              Restore X
          JSR     GET_MEM   Get a Byte From Memory
          STAA    LV        Level Value - For Display
          TAB               New Byte Goes to (B)
          JSR     XCHG      Swap X <--> Y
          JSR     GET_MEM   Get Threshold Byte
          STAA    TH        Threshold Byte - For Display
          JSR     XCHG      Swap Back X <--> Y
          INX               Point to Next Data Element
          INX               Two Hex Char = 1 Byte
          INY               Point to Next Thr Element
          INY               Two Hex Char + 1 Byte
          CBA               Compare Data to Threshold
          BEQ     SAME      If Equal - Go Same
          BPL     LOWER     Branch if Data < Threshold
          BRA     HIGHER    Branch Cause Data > Thresh
SAME:     DEX               If Same Look Back One Hex
          DEY               Look Back One Hex Digit
          JSR     GET_MEM   Get Data LO Byte.
          TAB               Put LO Byte in B
          JSR     XCHG      Exchange X and Y
          JSR     GET_MEM   Get Threshold LO Byte
          JSR     XCHG      Switch Back X and Y
          INX               Bump Hex Data Pointer
          INY               Bump Hex Thresh Pointer
          CBA               Compare Data to Threshold
          BEQ     LOWER     If Data = Threshold Branch
          BPL     LOWER     If Data < Threshold Branch
          BRA     HIGHER    Here Data > Threshold
LOWER:    CLR     AB_CTR    Clear Data-Above_Thr Ctr
          CLR     LOW_HI    Make Low-Hi Indicator Low
          TST     CX_CTR    Look at Thresh Crossing Ctr
          BEQ     CHG_CNT   If Crossing Ctr = 0 Branch
          DEC     CX_CTR    Ctr > 0, So Decrement
          BRA     CHG_CNT   Continue On
HIGHER:   LDAA    AB_CTR    Get Above Counter
          INCA              Increment Counter Value
          STAA    AB_CTR    Store Count Back
          CMPA    #AB_MAX   Is Above Cnt > Max Cnt?
          BNE     NOT_MAX   If Not Yet Max - Branch
          JSR     EVENT     We Have an Event Generted
          CLR     AB_CTR    After Event, Clear Above Ctr
          LDAA    #CTBTW    Load Counts Before Next Evnt
          STAA    CX_CTR    Store Into Crossing Cntr
          STAA    LOW_HI    Store Any Non-Zero Value
          BRA     CHG_CNT   Continue
NOT_MAX:  TST     LOW_HI    Not Max Count Yet Hi or Low?
          BEQ     NOTMX2    If Zero Branch
          BRA     CHG_CNT   Continue
NOTMX2:   TST     CX_CTR    Test Crossing Counter
          BNE     NOT_RDY   Not Ready for Next Event
          JSR     EVENT     CX_CTR=0, Generate Event
          LDAA    #CTBTW    Load Counts Between Events
          STAA    CX_CTR    Store Again in Cross Ctr
          INC     LOW_HI    Make LOW_HI > Zero
NOT_RDY:  DEC     CX_CTR    Lower Counter Toward Zero
CHG_CNT:  TST     SCR_TST   Are we Displaying?
          BEQ     CONT      Branch if Not Displaying
          LDAA    C1        High Order Value of Count
          JSR     HI_NIB    Display Upper Nibble
          JSR     LO_NIB    Display Lower Nibble
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          LDAA      C2         Low Order Value of Count
          JSR       HI_NIB     Display High Nibble
          JSR       LO_NIB     Display Low Order Nibble
          LDAA      #'_'       Space for Spacing Fields
          JSR       OUTPUT     Send Space Character
          LDAA      LV         Get Level of Sonar Echo
          JSR       HI_NIB      Display Upper Nibble
          JSR       LO_NIB     Display Lower Nibble
          LDAA      #'_'       Space Char for Spacing Fields
          JSR       OUTPUT     Send Space Character
          LDAA      TH         Get Threshold Value
          JSR       HI_NIB     Send High Nibble of Thresh
          JSR       LO_NIB     Display Low Nibble of Thresh
          LDAA      #'_'       Space Character
          JSR       OUTPUT     Send Space Character
          LDAA      EVL        Get ' ' or 'E' for Events
          JSR       OUTPUT     Send it Out
          LDAA      #$0D       Get a RETURN Character
          JSR       OUTPUT     Send RETURN Character
          LDAA      #$0A       Get a LF
          JSR       OUTPUT     Send LF
          DEC       VAR        Decrement Page Line Counter
          BNE       CONT       If Not Finished Branch
          LDAA      #22        Reload Page Line Counter
          STAA      VAR        Store Page Line Counter
          JSR       INCHAR     (Hit Any Character) Hangs!
CONT:     PSHX                 Save X (Level Pointer)
          LDX       COUNT      Look at Count
          INX                  Bump Count in X
          STX       COUNT      Store Count Back into COUNT
          CPX       #301       Is the COUNT Terminal?
          PULX                 Restore Level Pointer
          BEQ       TH_FIN     If COUNT=300 Then Finished
          JMP       THRESH1    Do Again
TH_FIN:   RTS
* Swaps X and Y registers using the stack.
XCHG:     PSHX      Push X
          PSHY      Push Y
          PULX      Pull X as Y
          PULY      Pull Y as X
          RTS
* This subroutine Displays Higher Order Nibble of A.
* A register preserved.
HI_NIB:   PSHA                 Save A for Later
          LSRA                 Shift Right 4 Times
          LSRA
          LSRA
          LSRA
          ORAA      #$30
          CMPA      #$39       '9' ASCII Character
          BLS       HI_N       $30 -> $39 (0-9) OK
          ADDA      #7         Correction for Hex
HI_N:     JSR       OUTPUT     Send It To SCI Output
          PULA
          RTS
* This subroutine Displays Lower Order Nibble of A.
* Register A NOT SAVED!
LO_NIB:   ANDA      #$0F
          ORAA      #$30
          CMPA      #$39       '9' ASCII Character
          BLS       LO_N       $30 -> $39 (0-9) OK
          ADDA      #7         Correction for Hex
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
LO_N:     JSR     OUTPUT    Send It To SCI Output
          RTS

* [ GET_MEM ]
* This subroutine obtains either Data or Threshold information
* from the RAM area.  It returns with the data in (A).
GET_MEM:  PSHX              Save X
          PSHB              Save B
          XGDX              Exchange X with AB (D)
          STAA    PORTC     High X is in A - Send Out On C
          JSR     PC_OUT    Set Port C as Output
          JSR     STB_HOL   Store Byte Into High Order Latch
          STAB    PORTC     Low X is in B - Send Out On C
          JSR     STB_LOL   Strobe Low Order
          JSR     PC_INP    Put Port C Into Input Mode
          JSR     OUT_ENAB  Set RAM Output Low-Z
          LDAA    PORTC     Get Data Via Port C Input
          JSR     OUT_DSBL  Disable RAM Output
          PULB              Pull B
          PULX              Pull X
          RTS

* [ EVENT ]
* This is where Events are generated for sending to Proc_B
* An Event represents a sonar target location.  Normally they
* occur at the crossing of the return echo across the thresh-
* hold value.  Event information is sent to P_B.
EVENT:    LDAA    NBR_EV
          TSTA
          BEQ     FIN_EV
          DECA
          STAA    NBR_EV
DO_EVNT:  LDAA    #'E'      Display Variable 'E'=Event
          STAA    EVL       Record that this Was Event
          PSHX              Save X
          LDX     #SPT_BUF  Point X to Transmit Buffer
          LDAA    #'>'      '>' Is First Char of Event
          STAA    0,X       Store the '>' in Buffer
          LDAA    #'E'      Load 'E' for (E)vent
          STAA    1,X       Store the 'E'
          LDAA    TRAND     Transducer Number
          STAA    2,X       Store After 'E'
          LDAA    H_POS     Head Position
          ORAA    #$30      Make Head Position into ASCII
          STAA    3,X       Store It
          LDAA    ANGL      Get Angle (0-5) Horizontal
          ORAA    #$30      Make into ASCII
          STAA    4,X       Store Angle
          LDAA    COUNT     Range Only (1-300) COUNT (0-1)
          ORAA    #$30      Make Into ASCII
          STAA    5,X       Store It Into Buffer
          LDAA    COUNT+1   Load Lower Byte of Total Range
          LSRA              Shift 4 Times to the Right
          LSRA              High Nibble --> Low Nibble
          LSRA
          LSRA
          ORAA    #$30      Make Into ASCII
          CMPA    #$39      Is Char (0-9)?
          BLS     ASC_OK    If '9' or Below - Branch
          ADDA    #7        If > 9 - Add 7 to Adjust Hex
ASC_OK:   STAA    6,X       Store Middle Char of Range
          LDAA    COUNT+1   Get Lower Byte of Range Again
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ANDA      #$0F      Mask Off Upper Nibble
          ORAA      #$30      Add $30 to Make ASCII
          CMPA      #$39      Is ASCII (0-9)?
          BLS       ASC_OK2   If (0-9) Branch
          ADDA      #7        Add 7 to Adjust for Hex
ASC_OK2:  STAA      7,X       Store in Buffer
          LDAA      #$0D      Get a RETURN Character
          STAA      8,X       Store it in Buffer
          JSR       TRANSMIT  Send the EVENT!
          PULX                Restore X
FIN_EV:   RTS


* [ MESSAGE ]
* This subroutine is called with (X) Pointing to the first
*  character of a string.  The last character of this string
*  will have to be $FF.  It sends this string to the OUTPUT
*  of the system (RS-232 to Sonar Module)
MESSAGE:  PSHA                Save A
MESS1:    LDAA      0,X       Get a Message (Command) Character
          CMPA      #$FF      Is it the EOR Character?
          BEQ       MESSRT    If EOR Char then Finished
          JSR       OUTPUT    Send the Character to Sonar Module
          INX                 Point to Next Char in String
          BRA       MESS1     Get Another Character
MESSRT:   LDAA      #$0D      RETURN Character
          JSR       OUTPUT    Send It
          LDAA      #$0A      Line Feed Character
          JSR       OUTPUT    Send It
          PULA                Restore A
          RTS


* [ CHK_DONE ]
* CHECK FOR 'DONE'.  This routine is set up to look for the message
* 'DONE' that appears after each SUCCESSFUL Sonar Module Command.
* Because this message may not be echoed back it does its checking
* by looping and looking at the serial data's register SCDR.  It
* looks for first an 'N' as in DO(N)E.  If it cannot find an 'N'
* in the time allotted it will give an error.  If it finds an 'N'
* then it looks for the RETURN that must follow in a few characters.
* Once again if it does not find this char then it returns an error.
* Errors are returned by placing a $01 in (A) on return.  $00 denotes
* no errors were found.
CHK_DONE: PSHX                Saves X
          PSHB                Saves B
          LDAB      #6        (3) Seconds of Looking for 'N'
          LDX       #$FFFF    $FFFF x 15 cycles @ 2MHz is 0.5 sec.
CDONE_1:  LDAA      SCSR      Read Status Register
          LDAA      SCDR      Go read the SCDR Over and Over
          CMPA      #'N'      Can it be an 'N'?
          BEQ       CDONE_2   If 'N' Found then go to Next Step
          DEX                 Lower the (X) Count
          BNE       CDONE_1   If not X=0 then Do Again
          DECB                Lower B From a Starting Point
          BNE       CDONE_1   If (B) is Not Zero them Do Again
          BRA       D_ERROR   Timed out and No Char Found
CDONE_2:  LDX       #2000     Look for RETURN in Next 15mS (#2000)
CDONE_3:  LDAA      SCSR      Read Status Register
          LDAA      SCDR      Read SCI Data Register for Data
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            CMPA      #$0A        Is it a LF Character?
            BEQ       CDONE_4     Branch if RETURN
            DEX                   Lower the 2000-0 Count
            BNE       CDONE_3     If Not Finished then Again
            BRA       D_ERROR     Error Occured (Time Out)
CDONE_4:    CLRA                  $00 Here Represents NO Error
            BRA       DONE_RET    Finished
D_ERROR:    CLRA
            INCA
            STAA      DN_ERR
            JSR       DON_ERR
DONE_RET:   TSTA                  Set Flags to Indicate Error
            PULB                  Restore Registers
            PULX
            JSR       DLY3        3mS Delay for Sonar Pod CRLF
            RTS

* [ TO_MEM ]
* Captures Incoming Data from Sonar Pod.
TO_MEM:     LDX       #0          Counter for Total Characters
            LDY       #0          Counter for Hex Characters
CHK_RDY:    LDAA      SCSR        Look at SCI Status Register
            ANDA      #$20        Look at RDRF Flag (Rec Dat Rdy)
            BEQ       CHK_RDY     If No Character Try Again
            INX                   Bump Total Character Counter
            CPX       TOT_CNT     Should be 638
            BEQ       IS_FIN      Finished if Terminal Count
            LDAA      SCDR        Get the Actual Data
            CMPA      #$0D        Don't Need RETURNS
            BEQ       CHK_RDY     Go Get Another Character
            CMPA      #$0A        Don't Need Line Feeds
            BEQ       CHK_RDY     Go Get Another
            INY                   Valid Character - Count It
            JSR       STORE       Store Character In Memory
            BRA       CHK_RDY     Loop Back - Get Another Char
IS_FIN:     CPY       DAT_CNT     Should be 600 Exactly
            BEQ       IS_RET      If 600 Then Branch to Return
            LDAA      PORTB       Error Has Occurred Here!
            ORAA      #$08        Get Hardware Error Flag
            STAA      PORTB       Store Error Bit
IS_RET:     RTS

* [ STORE ]
* Subroutine that stores a character in RAM memory.  This
* program Places the value of the A Register in the memory
* (RAM) pointed to by the STOR_PT pointer.  This pointer is
* auto incremented.
STORE:      PSHX                  Save X
            PSHB                  Save B
            PSHA                  Save A
            BSR       OUT_DSBL    Disable the RAM Output Lines
            LDX       STOR_PT     Get Storage Pointer
            INX                   Bump Pointer Just for Storage
            STX       STOR_PT     Store Incremented Pointer
            DEX                   Decrement Pointer Back
            XGDX                  Exchange (X) <--> (AB)
            STAA      PORTC       Store Hi-H for Output to RAM
            BSR       PC_OUT      Put on Output Lines
            BSR       STB_HOL     Strobe Hi Order Latch
            STAB      PORTC       Store Lo-H for Output to RAM
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
           BSR        STB_LOL    Strobe Lo Order Latch
           PULA                  Get the Data Byte
           STAA       PORTC      Store Port C
           BSR        WRTE_STB   Write to RAM Strobe
           BSR        PC_INP     Change Port C to Input
           PULB                  Restore B
           PULX                  Restore X
           RTS

* Set PORTC to Output (Usually an Input Port)
PC_OUT:    PSHA                  Save A
           LDAA       #$FF       All Bits High
           STAA       DDRC       Store in Data Direction Reg
           PULA                  Restore A
           RTS

* Change PORTC to Input (Normal State)
PC_INP:    CLR        DDRC       All Bits (0) in Data Direction
           RTS

* Strobe Low Order Latch
STB_LOL:   LDAA       PORTA      Get Port A (Has Strobe Lines)
           ORAA       #$40       Lower Order Strobe Bit High
           STAA       PORTA      Store High Bit in Port A
           ANDA       #$BF       Mask Strobe Bit to Zero
           STAA       PORTA      Store Back in A
           RTS

* Strobe High Order Latch
STB_HOL:   LDAA       PORTA      Get Port A (Has Strobe Lines)
           ORAA       #$20       High Order Strobe Bit Low
           STAA       PORTA      Store High Bit in Port A
           ANDA       #$DF       Mask Strobe Bit to Zero
           STAA       PORTA      Store Back in A
           RTS

* Lowers write strobe and then raises it for writing to RAM.
WRTE_STB:  PSHA                  Save A
           LDAA       PORTA      Get Port A (Control Lines)
           ANDA       #$EF       Mask Off Write Stb (PD4)
           STAA       PORTA      Store Active Strobe Port A
           ORAA       #$10       Make Strobe Inactive
           STAA       PORTA      Store Inactive Strobe
           PULA                  Restore A
           RTS

* Output Enable.  Takes RAM output lines out of Tri-State
OUT_ENAB:  LDAA       PORTA      Get Strobe Port (A)
           ANDA       #$7F       Make PA7 Low
           STAA       PORTA      Put Back in Port A
           RTS

* Output Disable.  Disables RAM output lines.
OUT_DSBL:  PSHA
           LDAA       PORTA      Get Strobe Port (A)
           ORAA       #$80       Make PA7 High
           STAA       PORTA      Store in Port A
           PULA
           RTS

* Reads from Memory starting at location RETV_PTR.
FROM_MEM:  PSHX                  Save X
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          PSHB                  Save B
          BSR       PC_INP      Make Port C Input
          LDX       RETV_PTR    Get Memory Retrieve Pointer
          INX                   Increment This Pointer
          STX       RETV_PTR    Store Incremented Pointer
          DEX                   Decrement Ptr to Org Value
          XGDX                  Exchange (H) <--> (AB)
          STAA      PORTC       Store Hi-H to Output Lines
          BSR       PC_OUT      Make Port C Output
          BSR       STB_HOL     Strobe High Order Latch
          STAB      PORTC       Store Lo-H to Output Lines
          BSR       STB_LOL     Strobe Low Order Latch
          BSR       PC_INP      Make Port C Input
          BSR       OUT_ENAB    Output Enable RAM
          LDAA      PORTC       Read RAM Output Via Port C
          BSR       OUT_DSBL    Disable RAM Output Lines
          PULB                  Restore B
          PULX                  Restore X
          RTS


* [ PRC_PING ]
* PROCESS PING.  Ping will be generated.  This routine take incomming
* data from the Sonar Module and places it into the memory of P_A.
* There it thresholds it with a stored template.  Values rising above
* the template create "EVENTS" that are fed to PROC_B where they are
* in-turn sent to the Host Computer.
PRC_PING: TST       PING        0=Test Mode (No Ping), 1=Normal Operate
          BEQ       PRC_P1      Return if Ping Not Going to Happen
          JSR       MESSAGE     Sends Ping Selected by CS Command Pointed
          JSR       TO_MEM      Loads Incoming Data
          JSR       CHK_DONE
PRC_P1:   RTS


* [ PRC_ROT ]
* PROCESS ROTATE.  (X) points to the R071 CS String that rotates the
* head some 8 degrees to the right.
PRC_ROT:  JSR       MESSAGE     Sends Rotate Message for 8 Degrees
          JSR       CHK_DONE    Checks for 'DONE' Message
PRC_R1:   RTS


* [ TRANSMIT ]
* This subroutine sends 5 zeros, followed by up to 64 characters
* of data, followed by a RETURN character, followed by another
* 5 zeros to the SPI.
* Proc_A output line PB1 is an indicator to Proc_B that the first
* actual char of data is being sent.  This is used to reset the
* receiving buffer pointer so as not to have picked up stray in-
* -the-pipeline characters.  It is only held active (H) for one
* character duration.  Proc_B senses this line only when it is
* servicing an SPI interrupt.
TRANSMIT: PSHA                  Save All Registers
          PSHB
          PSHX
          LDD       #333        Set Up Delay for 1mS
          STD       SDELY       Place This in Delay Variable
          LDX       #SPT_BUF    Load Value of Start of Trans Buffer
          BSR       FLUSH       Flush SPI With 5 $00's
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          LDAB      PORTB      Look at Port B (Control Outputs)
          ORAB      #$02       Make PB1 High
          STAB      PORTB      Set First Char Identifier Active
          BSR       CLRFLAG    Get Ready to Send First Real Char
          LDAA      0,X        Get First Actual Character
          INX                  Increment X as Usual
          STAA      SPDR       Store it to SPI
          JSR       DELY       Delay Before Dropping Active Line
          LDAB      PORTB      Get Port B for Control
          ANDB      #$FD       Make PB2 Low By Masking
          STAB      PORTB      Store it at Port B
          BRA       INPROG     Jump into the Regular Loop
TRAN2:    BSR       CLRFLAG    Do Housekeeping Functions
          LDAA      0,X        Get An Actual Character
          INX                  Bump Buffer Pointer
          STAA      SPDR       Send Character to SPI System
INPROG:   CMPA      #$0D       Was the Char a RETURN?
          BEQ       FINTRAN    If it Was a RETURN then Finish
          CPX       #SPT_PTR   Are We Past End of Buffer?
          BEQ       ADDCR      If Past EOB then Branch
          BRA       TRAN2      Not Finished, Do Again
ADDCR:    BSR       CLRFLAG    Housekeeping...
          LDAA      #$0D       Get a RETURN Character
          STAA      SPDR       Send to SPI
FINTRAN:  BSR       FLUSH      Flush Buffer With 5 Zeros
          PULX                 Restore All Registers
          PULB
          PULA
          RTS

* Clears the SPDR Flag
CLRFLAG:  JSR       DELY       Delay for Standard Delay
          LDAA      SPSR       Load Status Register
          LDAA      SPDR       Get Data (Dummy Get)
          RTS


* [ FLUSH ]
* Flush is used to send 5 Zeros ($00) to the SPI device.
* This is to Flush out any old data prior to actual transfers.
* All Registers Preserved
FLUSH:    PSHA                 Save A and B.  Only Registers Used
          PSHB
          LDAB      #5         Prepare to Write 5 Zeros
FL1:      BSR       CLRFLAG    Read SPSR, Read SPDR to Reset Syst
          CLR       SPDR       Write $00 to SPI
          DECB                 Count the Number of Times
          BNE       FL1        Do Again if Not Finished Flushing
          PULB                 Restore A and B
          PULA
          RTS

* [ RECEIVE ]
* This routine is responsible for reading information sent from
* P_A into the SPR_BUF input buffer.  Handshaking is used to
* coordinate the transfer between P_A and P_B.  When this routine
* is called, there is already a Request from Processor B to
* Service.
RECEIVE:  LDAA      PORTB      Output Bit (Control) Port
          ORAA      #$01       Acknowledge Bit
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          STAA    PORTB     Sent...  Committed Now!
          LDD     SH_DEL    Short Delay
          STD     SDELY     Set Up Delay
          CLR     TEMP      0=Discard Leading Zeros, 1=Don't
          LDAA    SPSR      Clear SPIF Flag
          LDAA    SPDR      Dummy Read
          LDX     #SPR_BUF  Point to Receive Buffer First Char
          CLR     SPDR      Flush out SPI Byte (IMPORTANT!)
          JSR     DELY      Delay for 1.5mS
          LDAA    SPSR      Read Status Register for RESET
          LDAA    SPDR      Read SPI Data for Clearing
ROT1:     CLR     SPDR      Send a Zero Byte to Rotate SR's
          JSR     DELY      Give it Time to Complete
          LDAA    SPSR      Clear SPIF Flag
          LDAA    SPDR      Getting Real Data
          TST     TEMP      0=Discard Leading Zeros, 1=Don't
          BNE     ROT2      Through Looking for Leading Zeros
          TSTA              Was Data Byte a Zero?
          BEQ     ROT3      Just Loop Again If Zero
          INC     TEMP      Don't Discard Zeros Anymore
ROT2:     STAA    0,X       Store in Buffer
          INX               Increment Buffer Pointer
          CPX     #SPR_BUF+64  Check for End Of Buffer EOB
          BNE     ROT3      Branch if Not End of Buffer
          DEX               Set Pointer Back at End - EOB
          JSR     ROR_ERR   Receiver Overrun Error
ROT3:     LDAA    PORTA     Get Request To Send Flag Byte
          ANDA    #$01      Look at Request To Send (RTS)
          BNE     ROT1      Go Do Again if Not Finished
          LDAA    PORTB     Get Ready to Clear Error Bit
          ANDA    #$FE      Mask for Error Bit Clearing
          STAA    PORTB     Store in Port B
          INC     SP_DRDY   Indicates Data Input Buf Full
          RTS

* [ INCHAR ]
* Subroutine to input one character from the SCI and echos
* it back.
INCHAR:   LDAA    SCSR      Look at SCI Status Register
          ANDA    #$20      Look at RDRF Flag
          BEQ     INCHAR    If RDRF Flag = 0 then No Data
          LDAA    SCDR      Load Character - It is Ready
          JSR     OUTPUT    Send it Back
          RTS


* [ OUTPUT ]
* Called with Byte to output in Register A.
* All registers preserved.
OUTPUT:   PSHA              Save Byte to Output in A
O_WAIT:   LDAA    SCSR      Load SCI Status Register
          ANDA    #$80      Look at Bit 7
          BEQ     O_WAIT    If TDRE=0 Loop, TrDatRegEmpty
          PULA              Get Byte to Output
          STAA    SCDR      Send Byte Out
          RTS


* [ DELAY ]
* Variable delay loop.  Called with a preset value of 10mS delays
* in DELCYC.  All registers perserved.
* Ex:  If DELCYC=30 then delay is 30x10mS or 0.3 Seconds.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
DELAY:     PSHA                   Save A
           LDAA      DELCYC       Load Preset Num of Cycles
DEL1:      JSR       DLY10        Call BUFFALO's 10mS Delay
           DECA                   Bump down count
           BNE       DEL1         Jump if not finshed
           PULA                   Restore A
           RTS

* [ DLY3 ]
* This subroutine when called delays for 3mS and returns
* No registers are effected.
DLY3:      PSHX                   Save X
           LDX       #1000        Count for 3mS
DLY3LP:    DEX                    Decrement Count
           BNE       DLYLP        If not Finished Loop
           PULX                   Restore X
           RTS


* [ DLY10 ]
* This subroutine when called delays for 10mS and returns
* No registers are effected.
DLY10:     PSHX                   Save X
           LDX       #$0D06       Count for 10mS
DLYLP:     DEX                    Decrement Count
           BNE       DLYLP        If not Finished Loop
           PULX                   Restore X
           RTS

* [ DELY ]
* Variable Short Delay.  SDELY must be set prior to calling
* Every Count in SDELY is 1/333 mS.  333=1mS
* All registers perserved.
DELY:      PSHX                   Save X
           LDX       SDELY        Variable Delay 333=1mS
DLYLP:     DEX                    Bump Down
           BNE       DLYLP        If not finished do again
           PULX                   Restore X
           RTS


* Command Strings for Sonar Module
CS1:       FCC       ':L100'      Rotate Left 100 Steps
           FCB       $FF
CS2:       FCC       ':R071'      Step of 8 Degrees (Right)
           FCB       $FF
CS3:       FCC       ':L071'      Step of 8 Degrees (Left)
           FCB       $FF
CS4:       FCC       ':R035'      Half Center String
           FCB       $FF
CS5:       FCC       ':P001'      Ping Head #1
           FCB       $FF
CS6:       FCC       ':P002'      Ping Head #2
           FCB       $FF
CS7:       FCC       ':P004'      Ping Head #3
CS8:       FCB       $FF          Just Sends CRLF
CS9:       FCC       'G B600'     Start User Program
           FCB       $FF
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          OPT     c
* Gary R. Boucher SONAR PRE-PROCESSOR B (Closest to Host Comp)
*
* Written by Gary R. Boucher
*
* Equates Section
PORTA     EQU     $1000     Port A
PORTB     EQU     $1004     Port B
PORTC     EQU     $1003     Port C
PORTD     EQU     $1008     Port D
DDRC      EQU     $1007     Data Direction for C
DDRD      EQU     $1009     Data Direction for D
PACTL     EQU     $1026     For A7 direction
SPCR      EQU     $1028     SPI Control Register
SPSR      EQU     $1029     SPI Status Register
SPDR      EQU     $102A     SPI Data Register
BAUD      EQU     $102B     Baud Rate Register
SCCR1     EQU     $102C     SCI Control Register 1
SCCR2     EQU     $102D     SCI Control Register 2
SCSR      EQU     $102E     SCI Status Register
SCDR      EQU     $102F     SCI Data Register
OPTION    EQU     $1039     Option Register
ADCTL     EQU     $1030     A/D Control Channel
VSCI      EQU     $00C4     SCI Vector Location
VSPI      EQU     $00C7     SPI Vector Location
ADR1      EQU     $1031     A/D Byte 1
STACK     EQU     $01FF     Stack Pointer

          ORG     $0000     Start of RAM Memory

* RAM Variables Section
SCI_BUF:  RMB     32        SCI Buffer Area for Commands
SCI_PTR:  RMB     2         SCI Buffer Pointer
SPR_BUF:  RMB     64        SPI Receive Buffer (Events)
SPR_PTR:  RMB     2         SPI Receive Buffer Pointer
SPT_BUF:  RMB     64        SPI Transmit Buffer
SPT_PTR:  RMB     2         SPI Transmit Buffer Pointer
SCI_RDY:  RMB     1         0=No CR, 1=CR
ERR_TYPE: RMB     1         Error Flag Byte
PADING:   RMB     1         Flag for $00 Padding Characters
SP_SENT:  RMB     1         Transmit Complete Flag
SPI_RDY:  RMB     1         $01=Command Received, $00=No Comnd
DELCYC:   RMB     1         Size of Delay 1=10mS, 2=20mS ETC.
SDELY:    RMB     2         333=1mS
TEMP:     RMB     2         General Use Ram Space
HOST_IT:  RMB     1         1=Send to Host, 0=Don't Send
HI_BYTE:  RMB     1         High Byte of Address for EPROM
MD_BYTE:  RMB     1         Medium Byte of Address for EPROM
LO_BYTE:  RMB     1         Low Byte of Address for EPROM
PS_NG1:   RMB     1         Sign Bit for (X) Value
PS_NG2:   RMB     1         Sign Bit for (Y) Value
PS_NG3:   RMB     1         Sign Bit for (Z) Value
Q_WRT:    RMB     2         Front End Pointer for Queue
Q_RD:     RMB     2         Rear End Pointer for Queue
SPOL_CT:  RMB     1         Byte Counter for Spooler
H_RNG:    RMB     1         High Binary Value of Range
L_RNG:    RMB     1         Low Binary Value of Range
H_R:      RMB     1         High Order ASCII Char of Range
M_R:      RMB     1         Middle Ord ASCII Char of Range
L_R:      RMB     1         Low Order ASCII Char of Range
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ORG       $00D0     Location for FIFO_Q

* FIFO_Q is the queue used for spooling the output data.
FIFO_Q:   RMB       256       Queue for Serial Output

          ORG       $B600     EEPROM Starting Location

* Initial Start-Up location for EEPROM location.

START:    LDS       #STACK    Set Stack Top
          LDAA      OPTION    Option Register $1039
          ORAA      #$90      Bit to turn on ADPU Bit (Pump)
          STAA      OPTION    Store it
          JSR       INIT      Init the System
          BRA       MAIN      Start Looking at Commands

* EEPROM Constants
STRTD:    FCB       30        Delay Timer (0.3 Seconds)
DSP_ES:   FCB       1         1=Display SPI Event, 0=Do Not
HANG:     FCB       0         1=Hang on Transmission, 0=Don't
ECHO:     FCB       0         1=Echo Host Commands, 0=Don't

************************************************************
***** M  A  I  N     L  I  N  E     P  R  O  G  R  A  M ****
************************************************************

          ORG       $B630     Secondary EEPROM Locat for MAIN

* MAIN is the Main Line Program for Processor B.  This program
* tests the interrupt driven SCI sytem for a complete command.
* If a command is present from the Host then it is serviced.
* Then the SPI buffer is checked for Data/Command Ready and
* serviced if needed.
MAIN:     TST       SCI_RDY   Check SCI Command Ready
          BNE       SCI_IDR   SCI Command Rdy Causes a Branch
          TST       SPI_RDY   Check SPI Command Ready
          BNE       GET_DAT   If Command Ready Branch
          BRA       MAIN      Loop Again

* SCI Interrupt System Data Service Segment
SCI_IDR:  LDY       #SPT_BUF  Y=Start of SPI Transmit Buffer
          LDX       #SCI_BUF  X=Start of SCI Receive Buffer
          STX       SCI_PTR   Store SCI Pointer to SCI_PTR
          CLR       SCI_RDY   Make SCI_RDY False
SCI0:     LDAA      0,X       Get a Byte from SCI Receive Buf
          STAA      0,Y       Store in SPI Transmit Buffer
          TST       ECHO      1=Echo Com to Host, 0=Don't Echo
          BEQ       SCI1
          JSR       OUTPUT    Send Command Back to Host
SCI1:     INX                 Point to Next SCI Byte
          INY                 Point to Next SPI Buffer Location
          CMPA      #$0D      Was the Byte a RETURN?
          BEQ       SCI_FIN   If RETURN Then Finished
          BRA       SCI0      Loop to Move Another Byte
SCI_FIN:  TST       ECHO      Test for Echo Back to Host
          BEQ       SCF0      If ECHO=0 then Branch (No Echo)
          LDAA      #$0D      Load RETURN
          JSR       OUTPUT    Send it
          LDAA      #$0A      Load Line Feed
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
         JSR      OUTPUT    Send it
SCF0:    JSR      TRANSMIT  Call SPI Transmit To PA Routine
         TST      ECHO      Test for Echo Back to Host
         BEQ      SCF1      If ECHO=0 then Branch (No Echo)
         LDAA     #$0D      Load RETURN Character
         JSR      OUTPUT    Send it
         LDAA     #$0A      Load Line Feed Character
         JSR      OUTPUT    Send it
SCF1:    JMP      MAIN      Complete the Loop

* GET_DAT is a program segment that takes received information
* from the SPI and acts upon it.  If the information is an error
* indication requested from the Host, the information will always
* be delivered back to the host.
GET_DAT: CLRA               Make a Zero
         INCA               Make a (1) From (0) (Set up to Send)
         STAA     HOST_IT   0=Do Not Send to Host, 1=Send
         LDX      #SPR_BUF  Point X to SPI Receive Buffer
         LDAA     0,X       Get First Character
         CMPA     #'>'      Is it a Legal Command?
         BNE      CLR_FG
         LDAA     1,X
         CMPA     #'E'      Is it an Event?
         BNE      REC_LP    If Not 'Event' Go Display
         TST      DSP_ES    Test Display Event String
         BNE      REC_LP
         CLR      HOST_IT
REC_LP:  LDAA     0,X       Get SPI Byte From PA
         INX                Point to Next Buffer Position
         TST      HOST_IT   Does Host Need to See it?
         BEQ      RECLP0    If HOST_IT=0 Do Not Output
         JSR      OUTPUT    Send it To Host
RECLP0:  CMPA     #$0D      Is it the Last (RETURN) Char?
         BNE      REC_LP    If Not Last Character Branch
         TST      HOST_IT   Send to Host?
         BEQ      RECLP1    If Zero Host Does Not See it
         LDAA     #$0D      Get a RETURN Character
         JSR      OUTPUT    Send RETURN to Host
         LDAA     #$0A      Get a Line Feed Character
         JSR      OUTPUT    Send LF to Host
RECLP1:  LDX      #SPR_BUF  Point to Start of SPI Rec Buffer
         LDAA     1,X       Get Identifier Byte (E,e, Etc)
         CMPA     #'E'      Is it an Event?
         BNE      NOEVENT   If Not an Event,then Loop MAIN
         JSR      MK_EVENT  Service the Event
NOEVENT: LDAA     1,X       Get Type Character
         CMPA     #'e'      Was it an Error Report?
         BNE      CLR_FG    If Not Err Rpt then Finish
         LDAA     #'B'      ID as B Type Origion
         JSR      OUTPUT    Send the B for Origion
         LDAA     #'e'      Get (e)rror Reporting Byte
         JSR      OUTPUT    Send it
         LDAA     #'{'      Get Squiggly Bracket
         JSR      OUTPUT    Send it
         LDAB     #8        Set Up to Rotate 8 Times
ROT_ET:  ROL      ERR_TYPE  Rotate Error Type into CY
         BCS      BIT_ONE   If CY=1 then (1) Goes to Host
         LDAA     #'0'      Get the Default...  (0)
         BRA      RT_AGN    Send it and Rotate Again
BIT_ONE: LDAA     #'1'      Carry Was a (1)
RT_AGN:  JSR      OUTPUT    Send it
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          DECB                 Lower B Count (Started at 8)
          BNE       ROT_ET     If Not Finished then Rotate
          ROL       ERR_TYPE   Restore to Origional
          LDAA      #'}'       Get a Closing Bracket
          JSR       OUTPUT     Send the Bracket
          LDAA      #$0D       Get a RETURN Character
          JSR       OUTPUT     Send it
          LDAA      #$0A       Get a Line Feed Character
          JSR       OUTPUT     Send it
          CLR       ERR_TYPE   Auto Clear Error Flags on Display
CLR_FG:   CLR       SPI_RDY    Reset the SPI Ready Flag
          JMP       MAIN

**************************************************************
***********  S  U  B  R  O  U  T  I  N  E  S  ***************
**************************************************************

          ORG       $D000      EPROM Area


* [ INIT ]
* This routine does an initialization of the system.  It is
*   called only once upon RESET or POR.
INIT:     LDAA      #$80       Initial Setting for PORTA
          STAA      PORTA      A7=(1), All Other Strobes Low
          LDAA      PACTL      Byte has A7 Data Direction
          ORAA      #$80       Bit to Set for A7 as Output
          STAA      PACTL      A7 Set as Output
          CLR       PORTB      Zero (B) Port
          CLR       PORTC      Zero Before Setting Direction
          CLR       DDRC       Make All C Pins Input
          LDAA      #$06       Set Up for Slave SPI
          STAA      DDRD       Get Data Direction for Port D
          LDAA      #$7E       The $7E is the JMP Opcode
          STAA      VSPI       Set Up Vector for SPI (JMP)
          STAA      VSCI       Set Up Vector for SCI (JMP)
          LDD       #SPI_ISR   D = Address of SPI Service Rout
          STD       VSPI+1     Store at VSPI Jump Vector
          LDD       #SCI_INT   D = Address of SCI Service Rout
          STD       VSCI+1     Store at VSCI Jump Vector
          LDAA      #$C4       Intr,Enabl,Slave,CPOL=0,CPHA=1
          STAA      SPCR       SPI Control Register
          LDAA      #$30       9600 Baud Rate (Host Baud)
          STAA      BAUD       Store Baud Register
          CLR       SCCR1      8 Bit Data, No Bit 8 Etc.
          LDAA      #$2C       Enables Receive Int ONLY
          STAA      SCCR2      SCI Control Register 2
          LDD       #SCI_BUF   Location of First Buf Position
          STD       SCI_PTR    Store First Position in Pointer
          LDD       #SPR_BUF   Start Location of Rec Buffer
          STD       SPR_PTR    Store in Buffer Pointer
          LDD       #SPT_BUF   First Position in Trans Buffer
          STD       SPT_PTR    Store as Pointer
          CLR       SCI_RDY    SCI ($0D) Indicator
          CLR       SPI_RDY    Received Command Indicator
          CLR       PADING     Clears Padding Variable
          LDD       #FIFO_Q    Spool Buffer
          STD       Q_WRT      Write Buffer Pointer
          STD       Q_RD       Read Buffer Pointer
          CLR       SPOL_CT    No Elements in Spooler
          CLR       ERR_TYPE   Error Reporting Flag
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          LDAA      STRTD     Starting Delay out of Reset
          STAA      DELCYC    Store in Delay Variable
          JSR       DELAY     Go Delay for 10mS x STRTD
          CLI                 Clear Interrupt Mask
          RTS

* [ SCI_INT ]
* The SCI Receiver and Transmitter use the same SPI
* Interrupt Vector.  For this reason only one vector
* location is used for both operations.  This routine
* is called each time either Interrupt is generated.
* This sub first looks at the receiver RDRF Flag and
* then the TDRE Flag.
SCI_INT:  LDAA      SCSR      Look at SCI Status
          ANDA      #$20      Look at RDRF Flag
          BEQ       SCINT2    Branch if No Rec Data Avail
          JSR       SCI_ISR   Service the SCI Input Rout
SCINT2:   TST       SPOL_CT   In Process of Sending??
          BEQ       FIN_SCI   Finished if Nothing to Send
          LDAA      SCSR      Look at SCI Status Again
          ANDA      #$80      Look at TDRE Flag Status
          BEQ       FIN_SCI   (0)=Not Ready to Xmit Yet
          JSR       SPOOL     Go Transmit a Char from Buf
FIN_SCI:  RTI

* [ SPOOL ]
* When called, this routine sends a single byte found in
* the FIFO_Q buffer to the output.  This routine should
* not be called if there is no data in queue. The SPOL_CT
* byte counter is decremented automatically.
SPOOL:    LDX       Q_RD      Get Queue Pointer in X
          LDAA      0,X       Get Read Byte
          STAA      SCDR      Send it to Output Device
          INX                 Bump Pointer to Next Byte
          CPX       #FIFO_Q+256 Check for End of Queue
          BNE       GSP2      Branch if Not End of Que
          LDX       #FIFO_Q   Point Y to Start of Queue
GSP2:     STX       Q_RD      Store Pointer Till Next Tm
          DEC       SPOL_CT   Lower Buffer Count
          TST       SPOL_CT   Is it Time to Stop Sending?
          BNE       GSP3      If Not Time - Continue
          LDAA      SCCR2     SCI Control Register II
          ANDA      #$7F      Make TIE Low (OFF)
          STAA      SCCR2     SCI Control Register
GSP3:     RTS

* [ OUTPUT ]
* Called with Byte to output in Register A.  Sends to Spooler.
OUTPUT:   BRA       PUT_SPOL  PUT_SPOL Handles All Output

* [ PUT_SPOL ]
* This subroutine is called anytime there is a need to send
* data to the Host Computer.  The data is passed to this
* subroutine via the (A) register.
PUT_SPOL: PSHX                Save X Register
          PSHB                Save B Register
          PSHA                Save A Register
          SEI                 Stop ALL Maskable Interrupts
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          LDAB      SPOL_CT   Load Byte Counter for Spooler
          INCB                Bump Up Byte Count
          STAB      SPOL_CT   Store Spool Byte Counter
          LDX       Q_WRT     Load X With Queue Write Ptr
          STAA      0,X       Place Byte in Queue
          INX                 Bump Queue Write Pointer
          CPX       #FIFO_Q+256 Compare to EOQ
          BNE       PTSP1     Branch if Not End of Queue
          LDX       #FIFO_Q   Point to Beginning of Queue
PTSP1:    STX       Q_WRT     Store Write Pointer Back
          LDAA      SCCR2     Get Control Register II
          ORAA      #$80      Make TIE=1
          STAA      SCCR2     Put Back in Register
          CLI                 Allow Interrupts Again
          PULA                Restore A
          PULB                Restore B
          PULX                Restore X
          RTS

* [ SCI_ISR ]
* SCI Interrupt Service Routine.  Called every time a new SCI
* character comes in.
SCI_ISR:  LDAA      SCSR      Load SCI Status Register
          ANDA      #$0F      Look Only at Flags OR,NF,FE
          BNE       ONF_ERR   If Not Zero then Error Occured
          LDX       SCI_PTR   Get Rec Buffer Pointer in X
          CPX       #SCI_PTR  Is Pointer Past Buffer?
          BEQ       OVR_ERR   EOR Error!  Go Figure!
          LDAA      SCDR      Load Data Character
          STAA      0,X       Place Data Character in A
          INX                 Point to Next Buffer Location
          STX       SCI_PTR   Store Bumped Pointer
          CMPA      #$0D      Is Byte Stored a RETURN Char?
          BNE       SCI_RET   If Not RETURN then Finished
          LDAA      #1        At This Point it WAS a RETURN
          STAA      SCI_RDY   Store a $01 In RETURN Location
          BRA       SCI_RET   Finished Return
OVR_ERR:  LDAA      #$01      OverRun Err Posit in ERR_TYPE
          ORAA      ERR_TYPE  Combine with Errors
          STAA      ERR_TYPE  Store Back
          BRA       SCI_RET   Finish
ONF_ERR:  LDAA      #$02      OR,NF,FE Err Position
          ORAA      ERR_TYPE  Combine with Errors
          STAA      ERR_TYPE  Store Back in Type
SCI_RET:  RTS

* [ SPI_ISR ]
* SPI Service Routine. This services Interrupts for SPIs coming
* into the system from Proc_A.  This routine will do one of
* two things.  If PB0 from Proc_A is High (Request Acknowledge)
* then Proc_B has the attention of Proc_A for the purpose of
* data transfer to P_A.  This will continue until P_B lowers
* its PB0 line to release P_A from its obligation.
* If No Acknowledge is present, then the SPI Interrupt can only
* mean that P_A is to sending a command or data to P_B.
* If P_A pulls its line B1 high during a character transfer this
* routine resets the SPR_PTR to the first position in the
* SPR_BUF.
* These are the connections between the two processors:
*
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
*    Proc_A   I/O     Proc_B         Definitions
*    ------  ------   ------    ----------------------------------
*     PB0    O<-->I    PA0       Acknowledgement that P_B can send
*     PB1    O<-->I    PA1       Tells P_B to reset rec buffer ptr
*     PB2    O<-->I    PA2       N/A
*     PA0    I<-->O    PB0       Requests to Send Data to P_A
*     PA1    I<-->O    PB1       N/A
*     PA2    I<-->O    PB2       N/A
*
SPI_ISR:  LDAA     SPSR       Read SPIF Flag to Reset
          LDAA     PORTA      Bit A0 is from Proc_A B0
          ANDA     #$01       Look at Acknowledge Bit
          BNE      TR_TO_A    If Ack then it is Transfer to P_A
          LDAA     PORTA      Not Transfer, (Data Moving A->B )
          ANDA     #$02       Is it a Buffer Reset
          BEQ      SPI_1      If Not a Reset Buffer Go to Next
          LDD      #SPR_BUF   Was a Reset Buffer Command
          STD      SPR_PTR    Store Initial Location in Pointer
SPI_1:    LDX      SPR_PTR    Look at Buffer Pointer Address
          LDAA     SPSR       Read Status Register for Clearing
          LDAA     SPDR       Get Data from Proc_A
          CLR      SPDR       Clear Data Going Back to P_A
          STAA     0,X        Store Data in SBUFFER Location
          INX                 Point to Next Buffer Location
          CPX      #SPR_BUF+64   End of Buffer?
          BNE      SPI_2      Not at End of Buffer
          LDAA     #$04       Error Posit for SPI Overrun
          ORAA     ERR_TYPE   Combine Error
          STAA     ERR_TYPE   Store Error Back
          BRA      RET_SPI    Finished if Error
SPI_2:    CMPA     #$0D       Is it a RETURN?
          BNE      SPI_3      Are We Finished?
          LDAA     #1         If Finished Load a ($01)
          STAA     SPI_RDY    Store as Flag
SPI_3:    STX      SPR_PTR    Store Pointer Back in Pointer Loc
          BRA      RET_SPI    Finished
* Note:  This section is only accessed when P_B has issued a request
*         for transfer and the acknowledge has been granted.  Thus,
*         this section moves a command or data to Proc_A.
TR_TO_A:  LDAA     SPSR       Read Status for Flag Clearing
          LDAA     SPDR       Get Dummy Byte and Throw Away
          TST      PADING     Are We in the Padding Phase?
          BNE      HAS_PAD    Branch if PADING > Zero
          LDX      SPT_PTR    Point to Current Buff Location
          LDAA     0,X        Get a Byte from SPT_BUF
          LDAB     SPSR
          STAA     SPDR       Put Byte in SPDR for Shipment
          INX                 Bump SPT_PTR Pointer
          STX      SPT_PTR    Store New Pointer Information
          CMPA     #$0D       Was SBUFFER Character a RETURN?
          BNE      RET_SPI    Branch if NOT a Return
          INC      PADING     Make PADING > Zero
          BRA      RET_SPI    Finished
HAS_PAD:  LDAA     PADING     Find Out How Many Padding Char Sent
          CMPA     #5         Is it Up to (5)?
          BEQ      FIN_PAD    If Up to (5) Do Nothing
          INCA                Not Up to (5).  Increment Padding
          STAA     PADING     Store New Padding Value
          LDAA     SPSR       Clear By Reading SPSR
          CLR      SPDR       Store Padding Character ($00)
          BRA      RET_SPI    Finished
FIN_PAD:  LDAA     #$0D       RETURN Character (Something to Wrt)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
           LDAB     SPSR       Read Status Register
           STAA     SPDR       Make Sure We Always Write to SPDR
           LDAA     PORTB      Padding Finished Here
           ANDA     #$FE       Mask to Make PB0 Low
           STAA     PORTB      PB0 is Now Low (Request)
           INC      SP_SENT    Signal that Transfer Complete
RET_SPI:   RTI                 All Done!

* [ MK_EVENT ]
* This subroutine sends a string to the Host computer containing
* (X,Y,Z) information followed by a CR and LF.  The format for
* this string is shown inside the hard brackets:
*
*          [ 43.67-182.3 .1294<CR><LF>]
*           <-X--><-Y--><-Z-->
*
* This will represent values of:
*
*     X = +43.67
*     Y = -182.3
*     Z = +0.1294
*
* It is the responsibility of the Host Computer to Parse this
* serial input string into these three values.  Each value is a
* total of 6 characters including sign.  Positive values will
* always have a leading space and not a '+'.
MK_EVENT:  JSR      MAKE_ADR   Makes Hi and Med Address for EPROM
           JSR      SET_ADR    Loads Hi and Med Addr into Latches
           CLRA                Clear A to Point to First Byte
           JSR      SET_LADR   Set the Zero into Latch (Low Addr)
           LDAA     #' '       A Space is Same as Positive
           STAA     PS_NG1     Store in (X) Pos/Neg Sign Locations
           STAA     PS_NG2     Same for (Y)
           STAA     PS_NG3     Same for (Z)
           LDAB     #'-'       Get the Negative Sign in B
           JSR      GET_MEM    Get Byte (0) from EPROM
           STAA     TEMP       Store in Temp Location to Parse
           ANDA     #$04       Look at Bit #2
           BEQ      MKE1       If Zero Leave ' ' In Place
           STAB     PS_NG1     If the Bit was '1' X is Negative
MKE1:      LDAA     TEMP       Load Byte (0) Again
           ANDA     #$02       Look at Bit #1
           BEQ      MKE2       If Zero it is Positive
           STAB     PS_NG2     Wasn't Zero Place a '-' Sign
MKE2:      LDAA     TEMP       Once Again Get Byte (0)
           ANDA     #$01       Look at Bit #0
           BEQ      MKE3       If Positive then Branch
           STAB     PS_NG3     Negative - Store a Minus Sign
MKE3:      LDAA     PS_NG1     Get X's Sign
           JSR      OUTPUT     Send it to Host
           LDAA     #$01       Point to First Byte of X Value
           BSR      OUT_XYZ    Send to Output 5 Bytes of ASCII
           LDAA     PS_NG2     Look at Y's Sign
           JSR      OUTPUT     Send Y's Sign
           LDAA     #$06       Point to Y Value in EPROM
           BSR      OUT_XYZ    Send Y's 5 Bytes to Host
           LDAA     PS_NG3     Look at Z's Sign
           JSR      OUTPUT     Send Z's Sign to Host
           LDAA     #$0B       Point to Z's Value
           BSR      OUT_XYZ    Send Z's 5 Bytes to Host
           LDAA     #'R'       Identify Range with 'R'
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          JSR       OUTPUT    Send the 'R' Out
          LDAA      H_R       Get High ASCII Range Byte
          CMPA      #'0'      Is it Zero?
          BEQ       MKE4      If Zero then Do not Display
          JSR       OUTPUT    Send it to Output
MKE4:     LDAA      M_R       Get Second ASCII Byte
          CMPA      #'0'      Is it a Zero?
          BEQ       MKE5      If Zero Do Not Display it
          JSR       OUTPUT    Send Out
MKE5:     LDAA      L_R       Get Last LSD of Range (ASCII)
          JSR       OUTPUT    Send to Output
          LDAA      #$0D      Finish with a RETURN Character
          JSR       OUTPUT    Send Return
          LDAA      #$0A      Load Also A Line Feed Character
          JSR       OUTPUT    Send Line Feed to Host
          RTS

* Called with (A) loaded with starting LO Address
* Sends five characters to SCI Port.
OUT_XYZ:  LDAB      #5        Load B For Five Outputs
          STAB      TEMP      Preserve Count in Temp
OXYZ1:    PSHA                (A) Has Current EPROM L Ptr
          JSR       SET_LADR  Put In EPROM Latch (Low)
          JSR       GET_MEM   Get a Byte from EPROM
          JSR       OUTPUT    Send it to Host
          PULA                A is Low Addr Pointer
          INCA                Bump Low Addr Pointer
          DEC       TEMP      Lower Count
          BNE       OXYZ1     If Not Finished Do Again
          RTS

* [ MAKE_ADR ]
* This subroutine acts on new strings entering the SPI
* system receive buffer (SPR_BUF).  Once this buffer is
* loaded with a new Event Sting this subroutine is called
* which forms a two-byte address used to look up the data
* stored in EPROM.  The remainder of the full address has
* to be loaded in the LO_BYTE location.  This portion of the
* address is used for accessing the separate elements of
* each unit of information.  Q0 of u6 (pin 2) is not
* connected to the memory address lines.
MAKE_ADR: LDX       #SPR_BUF  Point X to SPI Rec Buffer
          LDAA      2,X       Transd Head ASCII "0"->"2"
          LSLA                Bits 0 & 1 -> Bits 6 & 7
          LSLA
          LSLA
          LSLA
          LSLA
          LSLA
          STAA      HI_BYTE   Start New Adr Byte in Mem
          LDAA      3,X       Get Head Position "0"-"1"
          LSLA                Shift 0 or 1 into Bit B5
          LSLA
          LSLA
          LSLA
          LSLA
          ORAA      HI_BYTE   Combine with HB from Memory
          STAA      HI_BYTE   Store Back in Memory
          LDAA      4,X       Get Angle ASCII 0-5
          ANDA      #$0F      Strip off 3 Hex
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
           LSLA
           LSLA
           ORAA      HI_BYTE   Combine with HB in Memory
           STAA      HI_BYTE   Store Back in Memory
           LDAA      5,X       Get Hi Range Byte
           ANDA      #$0F      Strip Off the 3 Hex
           LSLA                Move to Bit B1
           ORAA      HI_BYTE   Combine with Memory
           STAA      HI_BYTE   Put Back in Memory
           LDAA      6,X       Get Middle Range Byte
           BSR       ADJUST    Adjust for ASCII - Hex
           ANDA      #$08      Look Only at Bit 3
           LSRA                Shift B3 into B0
           LSRA
           LSRA
           ORAA      HI_BYTE   Set Bit in HB
           STAA      HI_BYTE   High Adr Byte is Packed!
           LDAA      6,X       Get Range Byte Again
           BSR       ADJUST    Adjust for ASCII - Hex
           ANDA      #$0F      Strip Off Lower Nibble
           LSLA                Low 3 bits to B7,B6,B5
           LSLA
           LSLA
           LSLA
           LSLA
           STAA      MD_BYTE   Start Medium Byte of Adr
           LDAA      7,X       Get Last Range Byte
           JSR       ADJUST    Adjust for ASCII - Hex
           ANDA      #$0F      Strip Off 3 Hex
           LSLA                Move to B0 -> B5
           ORAA      MD_BYTE   Combine with Memory Byte
           STAA      MD_BYTE   Place Back in Memory
           BSR       FIG_RNG   Figure Range in Decimal
           RTS
* This subroutine looks at register A and if the value is a
* legal ASCII 0-9 then it returns.  If a hex value (A-F)
* then it adjusts for the gap to make a true binary lower
* nibble.
ADJUST:    CMPA      #$39      ($39) Is '9'
           BLS       BYTE_OK   If 0-9 Then OK
           SUBA      #7        Otherwise Adjust For Hex
BYTE_OK:   RTS


* [ FIG_RNG ]
* This subroutine takes the HEX Range information round in
* the Event and converts it to a three-byte ASCII decimal
* value.  This value is then stored in H_R, M_R, and L_R.
* These are used later when sending the Event to Host.
FIG_RNG:   LDAA      5,X       Get High Range Hex Character
           ANDA      #$0F      Strip Off ASCII $30
           STAA      H_RNG     High Hex Can Only Be 0 or 1
           LDAA      6,X       Get Middle Hex Character
           BSR       ADJUST    Hex to Binary
           ANDA      #$0F      Strip Off ASCII $3
           LSLA                Rotate to Higher Nibble
           LSLA
           LSLA
           LSLA
           STAA      L_RNG     Store As Lower Range Binary
           LDAA      7,X       Get Low Order Range Hex Char
           BSR       ADJUST    Adjust for Binary from ASCII
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ANDA      #$0F       Strip Off ASCII $3
          ORAA      L_RNG      Combine with Existing L_RNG
          STAA      L_RNG      Store Back in L_RNG
          LDD       H_RNG      H_RNG + L_RNG --> (AB)
          LDX       #100       How Many 100's?
          IDIV                 Divide (AB)/(X)
          XGDX                 Remainder Now in X
          ORAB      #$30       How Many Times Did it Go?
          STAB      H_R        Store As ASCII
          XGDX                 Remainder Now in (AB)
          LDX       #10        How Many 10's?
          IDIV                 Divide (AB)/(X)
          XGDX                 Remainder Now in X
          ORAB      #$30       ASCII Nbr of 10's
          STAB      M_R        Store Middle Character
          XGDX                 Remainder now in (AB)
          ORAB      #$30       Make Remainder ASCII (0-9)
          STAB      L_R        Store in Low Position
          RTS

* [ SET_ADR ]
* This subroutine takes the HI_BYTE and MD_BYTE locations
* and loads them into the address latches (74HCT273's)
* that supply address line information to the TMS27C040.
SET_ADR:  BSR       OUT_DSBL   Make Sure PROM Outp is Hi-Z
          BSR       C_OUTPUT   Make PORTC Output
          LDAA      MD_BYTE    Get Medium Address Byte
          STAA      PORTC      Send MAB to PORTC
          BSR       STB_MD     Strobe Medium Byte to Latch
          LDAA      HI_BYTE    Get High Byte from Memory
          STAA      PORTC      Send Out on PORTC
          BSR       STB_HI     Strob High Byte to Latch
          BSR       C_INPUT    Make PORTC Input Mode
          RTS


* [ SET_LADR ]
* This subroutine loads (A) into the low byte of address
* information and sets the latch (u7).
SET_LADR: PSHA                 Save Low Addr Byte for Later
          BSR       OUT_DSBL   Make Sure EPROM is Not Out Mode
          BSR       C_OUTPUT   Make PORTC an Output
          PULA                 Get Low Byte of Address
          STAA      PORTC      Send Low Byte to PORTC
          BSR       STB_LO     Strobe Low Byte
          BSR       C_INPUT    Set PORTC Into Input Mode
          RTS


* [ GET_MEM ]
* This subroutine loads a byte from memory pointed to by the
* full address after calling SET_ADR and SET_LADR.
GET_MEM:  BSR       C_INPUT    Make PORTC an Input
          BSR       OUT_ENAB   Set EPROM to Output Mode
          LDAA      PORTC      Capture Data From EPROM
          PSHA                 Push Captured Data onto Stack
          BSR       OUT_DSBL   Disable the Output of EPROM
          PULA                 Get Data Byte Back
          RTS
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* Puts PORTC into Output Mode.
C_OUTPUT: LDAA      #$FF       All PC Pins Output
          STAA      DDRC       Data Direction Register for C
          RTS
* Puts PORTC into Input Mode
C_INPUT:  CLR       DDRC       Clear DDRC. All PC Pins Input
          RTS
* Strobe address into Hi Address Latch.
STB_HI:   LDAA      PORTA      Get Strobe Lines
          ORAA      #$10       Set PA4 High
          STAA      PORTA      Make it Go High
          ANDA      #$EF       Mask Out PA4
          STAA      PORTA      Store PA4=0
          RTS
* Strobe address into Medium Address Latch.
STB_MD:   LDAA      PORTA      Get Strobe Lines
          ORAA      #$20       Set PA5 High
          STAA      PORTA      Make it Go High
          ANDA      #$DF       Mask Out PA5
          STAA      PORTA      Store PA5=0
          RTS
* Strobe address into Low Address Latch.
STB_LO:   LDAA      PORTA      Get Strobe Lines
          ORAA      #$40       Set PA6 High
          STAA      PORTA      Make it Go High
          ANDA      #$BF       Mask Out PA6
          STAA      PORTA      Store PA6=0
          RTS
* Disable the EPROM output lines.
OUT_DSBL: LDAA      PORTA      Get Strobe/Enable Lines
          ORAA      #$80       Make PA7 High
          STAA      PORTA      Store PA7 High
          RTS
* Enable the EPROM output lines.
OUT_ENAB: LDAA      PORTA      Get Strobe/Enable Lines
          ANDA      #$7F       Mask Out PA7
          STAA      PORTA      PA7 = 0
          RTS


* [ TRANSMIT ]
* Called by the main line program after loading the SPT_BUF
* SPI Transmit buffer.  This routine raises Bit PB0 from P_B
* to P_A's PA0 input pin.  This requests data be transfered
* from P_B to P_A.  Execution hangs in this subroutine until
* P_A sends an acknowledge on its like PB0 and is received on
* P_B's PA0 along with SPI rotations until P_B releases the
* request for transfer.  This will occur at the end of the
* data movement as sensed by P_B.  Make sense???
TRANSMIT: LDAA      PORTB      Get Control Byte Port B
          ORAA      #$01       Make PB0=1
          STAA      PORTB      Set Bit High
          CLR       PADING
          LDX       #SPT_BUF   Get Starting Address of Buffer
          STX       SPT_PTR    Store Starting Addr in Pointer
          CLR       SP_SENT    Clear Done Flag
          TST       HANG       Should we Hang on Transmit?
          BEQ       TRN_RET    Finished if No Hanging
TRANS1:   TST       SP_SENT    Check Done Flag
          BEQ       TRANS1     If Not Finished Loop
TRN_RET:  RTS
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* [ DELAY ]
* Variable delay loop.  Called with a preset value of 10mS delays
* in DELCYC.  All registers perserved.
* Ex:  If DELCYC=30 then delay is 30x10mS or 0.3 Seconds.
DELAY:     PSHA                    Save A
           LDAA       DELCYC       Load Preset Num of Cycles
DEL1:      JSR        DLY10        Call BUFFALO's 10mS Delay
           DECA                    Bump down count
           BNE        DEL1         Jump if not finshed
           PULA                    Restore A
           RTS

* [ DLY10 ]
* This subroutine when called delays for 10mS and returns
* No registers are effected.
DLY10:     PSHX                    Save X
           LDX        #$0D06       Count for 10mS
DLYLP:     DEX                     Decrement Count
           BNE        DLYLP        If not Finished Loop
           PULX                    Restore X
           RTS

* [ DELY ]
* Variable Short Delay.  SDELY must be set prior to calling
*   Every Count in SDELY is 1/333 mS.  333=1mS
*   All registers perserved.
DELY:      PSHX                    Save X
           LDX        SDELY        Variable Delay 333=1mS
DLYLP:     DEX                     Bump Down
           BNE        DLYLP        If not finished do again
           PULX                    Restore X
           RTS
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
*  COMPASS AND SPEED PROCESSOR
*
          OPT    c
* Gary R. Boucher (c)
*
* Written by Gary R. Boucher
*
* Equates Section
PORTA     EQU       $1000     Port A
PORTB     EQU       $1004     Port B
PORTC     EQU       $1003     Port C
PORTD     EQU       $1008     Port D
DDRC      EQU       $1007     Data Direction for C
DDRD      EQU       $1009     Data Direction for D
TMSK2     EQU       $1024     Timing Mask Register
TFLG2     EQU       $1025     Timing Flag Register
PACTL     EQU       $1026     For A7 direction
SPCR      EQU       $1028     SPI Control Register
SPSR      EQU       $1029     SPI Status Register
SPDR      EQU       $102A     SPI Data Register
BAUD      EQU       $102B     SCI Baud Rate Register
SCCR1     EQU       $102C     SCI Control Register 1
SCCR2     EQU       $102D     SCI Control Register 2
SCSR      EQU       $102E     SCI Status Register
SCDR      EQU       $102F     SCI Data Register
OPTION    EQU       $1039     Option Register
ADCTL     EQU       $1030     A/D Control Channel
ADR1      EQU       $1031     A/D Byte 1
STACK     EQU       $00FF     Stack Pointer
RAM       EQU       $0000     Ram Memory $0000-$00FF
EEPROM    EQU       $F800     EEPROM Memory $F800-$FFFF

          ORG       RAM       RAM Memory Start

* RAM Variables Section
HO_BYTE:  RMB       1         Most Significant Bits from SPI
LO_BYTE:  RMB       1         Lst Significant Bits from SPI
DT_100:   RMB       1         BCD 100's Location
DT_10:    RMB       1         BCD 10's Location
DT_1:     RMB       1         BCD 1's Location
SDELY:    RMB       2         Value for DELY (333 = 1mS)
DELCYC:   RMB       1         Nbr of 10mS Delays for DELAY
RTI_CNT:  RMB       2         Number of RTI's
WHL_CNT:  RMB       1         Water Wheel Lobe Count
VEL_RDY:  RMB       1         0=Velocity Not Ready, 1=Ready
VELOC1:   RMB       1         Holds ASCII First Velocity Char
VELOC2:   RMB       1         Holds ASCII Second Velocity Chr


          ORG       EEPROM    EEPROM Area for 68HC811F2

* Start of program area.  Option Register must be set early.
START:    LDS       #STACK    Set Stack Top
          LDAA      #$30      IRQ Negative Edge Triggered
          STAA      OPTION    Store it
          JSR       INIT      Init the System
          BRA       MAIN      Start Looking at Commands

* EEPROM Constants Section
RTI_CTS:  FDB       732       About 3 seconds
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
************************************************************
***** M  A  I  N    L  I  N  E    P  R  O  G  R  A  M ****
************************************************************

          ORG       $F840     Leaves Gap for Constants

* This is the Main Line Program for reporting both the compass
* heading and submarine velocity information to the host
* computer.  The compass heading is reported with a 'C' before
* the value. The velocity is reported with a 'V' as char one.
* The velocity value is reported in "Clicks".  A click is a
* count representing 1/6 of a revolution of the water wheel.
* The time interval that is used for collecting these clicks is
* predetermined by a certin number of RTI cycles.  This number
* is reloaded from RTI_CTS represented as an EEPROM constant.
MAIN:     TST       VEL_RDY   Is There a New Velocity Ready?
          BEQ       COMPASS   If No New Velocity then Branch
          CLR       VEL_RDY   Clear the Velocity Ready Flag
          LDAA      #'V'      Get a 'V' for Velocity
          JSR       OUTPUT    Send Out the 'V'
          LDAA      VELOC1    Load Velocity ASCII Char #1
          JSR       OUTPUT    Send First Velocity Character
          LDAA      VELOC2    Get Second Velocity Character
          JSR       OUTPUT    Send Second Velocity Char
          JSR       CRLF      Send RETURN and Line Feed Chars
COMPASS:  JSR       MEASURE   Get a New Compass Heading
          LDAA      #'C'      Get the (C)ompass Character
          JSR       OUTPUT    Send a 'C' to Output
          LDAA      DT_100    Get 100's ASCII Character
          JSR       OUTPUT    Send the 100's Character
          LDAA      DT_10     Get 10's ASCII Character
          JSR       OUTPUT    Send the 10's Character
          LDAA      DT_1      Get 1's Character
          JSR       OUTPUT    Senu the 1's Character
          JSR       CRLF      RETURN and Line Feed Char
          BRA       MAIN

************************************************************
***********  S  U  B  R  O  U  T  I  N  E  S  **************
************************************************************


* [ INIT ]
* This routine does an initialization of the system.  It is
* called only once upon RESET or POR.
INIT:     LDAA      #$3A      Set Up Data Direct for SPI, SCI
          STAA      DDRD      Store Data Direction Register
          LDAA      #$5D      SPE=1,MSTR=1,COPL=1,CPHA=1,E/4
          STAA      SPCR      Set Up SPI Channel
          LDAA      #$30      Code for 9600 Baud Rate
          STAA      BAUD      Place in Baud Register
          CLR       SCCR1     8 Bit Data, No Bit 8 Etc.
          LDAA      #$0C      Enable Transmitter/Receiver
          STAA      SCCR2     SCI Control Register II
          LDAA      #$0F      Set All Control Lines High
          STAA      PORTC     Store in Port C
          STAA      DDRC      Make C0->C3 Output Lines
          LDAA      TMSK2     Get RTII Bit
          ORAA      #$40      Make RTII High - Enable RTI
          STAA      TMSK2     Store it Back in Register
          LDAA      TFLG2     Load Flag Register
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ORAA      #$40      Make RTIF Bit High
          STAA      TFLG2     Reset RTIF Flag
          LDAA      PACTL     RTI Rate Register
          ANDA      #$FC      Mask off RTR1 and RTR0
          STAA      PACTL     Store to make 4.096mS Irpt
          LDX       #$0001    $0001 Minus First Call=$0000
          STX       RTI_CNT   Clear RTI Counter
          DEX                 Make X=$0000
          STX       WHL_CNT   Clear Water Wheel Counter
          LDAA      #20       Set Up for 20 x 10mS
          STAA      DELCYC    Ready for 0.2 Second Delay
          JSR       DELAY     Go Do the Delay
          JSR       RESET     Reset the Compass Module
          CLI                 Clear Interrupt Mask
          RTS

* [ RESET ]
* This subroutine simply resets the Vector Compas Module.
* Normally this is done only once at the time of
* initialization.
RESET:    LDAA      PORTC     Load Control Bits
          ORAA      #$0F      Set All Lines High
          STAA      PORTC     Store to Make All Lines High
          JSR       DLY10     Delay for 10mS
          ANDA      #$0E      Make C0 Low
          STAA      PORTC     Place the C0 (Reset) on Line
          LDAB      #10       Set Up for 10 x 10mS
          STAB      DELCYC    Store in Delay Counter Variable
          JSR       DELAY     Go Delay for 100mS
          ORAA      #$0F      Make Reset Line High With Others
          STAA      PORTC     Store Back in Port C
          JSR       DELAY     Delay for Another 0.1 Seconds
          RTS

* [ MEASURE ]
* This subroutine is used to sample the Vector Compass Module
* to determine the compass heading.  This routine follows the
* Vector slave timing diagram on page 13 of the application
* notes (Version 1.08).  The final product of this subroutine is
* the three ASCII heading bytes (DT_100, DT_10, and DT_1).  For
* example these three values might be "3", "5", and "1".  This
* would represent the heading 351 degrees.
MEASURE:  PSHA                Save A Register
          PSHB                Save B Register
          PSHX                Save X Register
          LDAA      PORTC     Get Port C (Control Bits)
          ORAA      #$0F      Make Sure All Line Are High
          STAA      PORTC     All Lines High
          LDAB      #2        20mS Delay Interval
          STAB      DELCYC    Store it to Delay Variable
          JSR       DELAY     Go and Delay 20mS
          ANDA      #$F7      P/C Goes Low (0)
          STAA      PORTC     Take P/C Low
          JSR       DELAY     Delay for 20mS
          ORAA      #$0F      Raise All Lines High
          STAA      PORTC     Bring Lines High Now
MLOOP:    LDAB      PORTA     Check End of Conversion (EOC)
          ANDB      #$01      Look at PA0
          BEQ       MLOOP     Branch Till PA0=1
          JSR       DELAY     Delay for 20mS After EOC=1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ANDA     #$FD      Mask for SS=0
          STAA     PORTC     Store to Make SS=0
          JSR      DELAY     Wait Before Shifting Data
          LDX      #33       0.1mS Delay Value
          STX      SDELY     Delay Variable = 0.1mS Delay
          LDX      #HO_BYTE  Point to Most Sig Byte
          LDAB     #2        Set Up to Do this 2 Times
          LDAA     SPSR      Clear Flags Etc.
          LDAA     SPDR      Dummy Data Fetch
GET_DAT:  CLR      SPDR      Rotate Data
          JSR      DELY      Delay for 0.1mS
          LDAA     SPSR      Reset SPIF Flag
          LDAA     SPDR      Get Data in (A)
          STAA     0,X       Store Data in Memory
          INX                Point to Next Memory Location
          DECB               Lower Number of Reads Left
          BNE      GET_DAT   If Not Finished - Read Again
          JSR      DELAY     Delay for 20mS
          LDAA     #$0F      Get Ready for All Lines High
          STAA     PORTC     Make All Control Lines High
          LDAA     HO_BYTE   Get High Order of 16 SPI Bits
          ANDA     #$0F      Look at 100's BCD Only
          ORAA     #$30      Make 100's BCD Code into ASCII
          STAA     DT_100    Store into 100's Location
          LDAA     LO_BYTE   Get Lower 8 of 16 SPI Bits
          LSRA               Shift High Nibble into Low Nib
          LSRA
          LSRA
          LSRA
          ORAA     #$30      Make High Nibble into ASCII
          STAA     DT_10     Store in 10's Location
          LDAA     LO_BYTE   Get Low Byte Again
          ANDA     #$0F      Strip Off High Nibble
          ORAA     #$30      Make into ASCII
          STAA     DT_1      Store in 1's Location
          PULX               Restore X
          PULB               Restore B
          PULA               Restore A
          RTS


* [ RTI_SER ]
* This is the RTI service routine.  This routine is called
* every 4.096mS where it decrements the RTI_CNT.  If this
* decrementation does not result in a Zero the routine just
* returns.  If the RTI_CNT goes to zero it is reloaded with
* a constant value (RTI_CTS) located in EEPROM.  This RTI_CTS
* count represents a certain number of RTI cycles at 4.096mS.
* This time interval is used to count Water Wheel Lobes in
* WHL_CNT.  Once the interval is complete velocity is
* determined.
RTI_SER:  LDAA     TFLG2     Get RTI Flag (RTIF)
          ORAA     #$40      Make RTIF Location (1)
          STAA     TFLG2     Clear RTIF Flag by Writing (1)
          LDX      RTI_CNT   Load the RTI Counter Variable
          DEX                Decrement the RTI Count
          STX      RTI_CNT   Store it Back into Count Vars
          BNE      RTI_RET   If Not Zero then Return
          LDX      RTI_CTS   Count=0 - Reload with RTI_CTS
          STX      RTI_CNT   Store Back in Count
          LDAA     WHL_CNT   # of Lobes in Last RTI Series
          LSRA               Shift High Nibble into Low Nib
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          LSRA
          LSRA
          LSRA
          ADDA      #$30       Make ASCII
          CMPA      #$39       Is the ASCII '0' to '9'?
          BLS       BYTE2      If 0-9 then Branch (No Add)
          ADDA      #7         Add 7 to Adjust
BYTE2:    STAA      VELOC1     Store First Velocity Byte
          LDAA      WHL_CNT    Load Wheel Velocity Byte Again
          ANDA      #$0F       Strip Off Upper Nibble
          ADDA      #$30       Make Into ASCII
          CMPA      #$39       Is the ASCII '0' to '9'?
          BLS       NOADD      If 0-9 then Branch (No Add)
          ADDA      #7         Add 7 to Adjust
NOADD:    STAA      VELOC2     Store Second Velocity Byte
          CLR       WHL_CNT    Clear Wheel Lobe Rotations
          LDAA      #1         Get $01 for Ready Indication
          STAA      VEL_RDY    Store - Make it Ready
RTI_RET:  RTI

* [ IRQ_SER ]
* This is the IRQ Service routine.  Every time a falling edge
* results from sensing the water wheel this routine is called.
* This service routine increments the wheel count and returns.
* The wheel count values continue to increment until the RTI
* service routine clears the count and calculates the velocity.
IRQ_SER:  INC       WHL_CNT    Load the Wheel Count
          RTI

* [ INCHAR ]
* Subroutine to input one character from the SCI and echos
* it back.
INCHAR:   LDAA      SCSR       Look at SCI Status Register
          ANDA      #$20       Look at RDRF Flag
          BEQ       INCHAR     If RDRF Flag = 0 then No Data
          LDAA      SCDR       Load Character - It is Ready
          JSR       OUTPUT     Send it Back
          RTS

* [ OUTPUT ]
* Called with Byte to output in Register A.
* All registers preserved.
OUTPUT:   PSHA                 Save Byte to Output in A
O_WAIT:   LDAA      SCSR       Load SCI Status Register
          ANDA      #$80       Look at Bit 7
          BEQ       O_WAIT     If TDRE=0 Loop, TrDatRegEmpty
          PULA                 Get Byte to Output
          STAA      SCDR       Send Byte Out
          RTS

* [ CRLF ]
* Carriage Return and Line Feed.
* No Registers Preserved.
CRLF:     LDAA      #$0D       Load a RETURN Character
          JSR       OUTPUT     Send the RETURN Character
          LDAA      #$0A       Load a Line Feed Character
          JSR       OUTPUT     Send it to Output
          RTS
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* [ DELAY ]
* Variable delay loop.  Called with a preset value of 10mS delays
* in DELCYC.  All registers perserved.
* Ex:  If DELCYC=30 then delay is 30x10mS or 0.3 Seconds.
DELAY:     PSHA                      Save A
           LDAA       DELCYC         Load Preset Num of Cycles
DEL1:      JSR        DLY10          Call BUFFALO's 10mS Delay
           DECA                      Bump down count
           BNE        DEL1           Jump if not finshed
           PULA                      Restore A
           RTS


* [ DLY10 ]
* This subroutine when called delays for 10mS and returns
* No registers are effected.
DLY10:     PSHX                      Save X
           LDX        #$0D06         Count for 10mS
DLYLP:     DEX                       Decrement Count
           BNE        DLYLP          If not Finished Loop
           PULX                      Restore X
           RTS


* [ DELY ]
* Variable Short Delay.  SDELY must be set prior to calling
* Every Count in SDELY is 1/333 mS.  333=1mS
* All registers perserved.
DELY:      PSHX                      Save X
           LDX        SDELY          Variable Delay 333=1mS
DLYLP:     DEX                       Bump Down
           BNE        DLYLP          If not finished do again
           PULX                      Restore X
           RTS


* Interrupt Vector Table.  Unused locations are represented
* by 'VECTOR'.  This VECTOR location points to an RTI in
* case they are accidentally called.

VECTOR:    RTI                       Just Return from Interrupt

           ORG        $FFD6          Interrupt Vector Table

VSCI:      FDB        VECTOR         SCI Interrupt Vector
VSPI:      FDB        VECTOR         SPI Vector
VPAIE:     FDB        VECTOR
VPAO:      FDB        VECTOR
VTOF:      FDB        VECTOR
VTOC5:     FDB        VECTOR
VTOC4:     FDB        VECTOR
VTOC3:     FDB        VECTOR
VTOC2:     FDB        VECTOR
VTOC1:     FDB        VECTOR
VTIC3:     FDB        VECTOR
VTIC2:     FDB        VECTOR
VTIC1:     FDB        VECTOR
VRTI:      FDB        RTI_SER        Real Time Interrupt Vector
VIRQ:      FDB        IRQ_SER        IRQ Interrupt Vector
VXIRQ:     FDB        VECTOR
VSWI:      FDB        VECTOR
VILLOP:    FDB        VECTOR
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
VCOP:       FDB         VECTOR
VCLM:       FDB         VECTOR
VRST:       FDB         $F800       RESET (Power On) Vector
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX B

PROCESSING HOST SOFTWARE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/*
SONAR - HOST COMPUTER PROGRAM

       Written by:    Gary R. Boucher
                            430 N. Dresden Cir.
                            Shreveport, LA  71115

       This program is used to evaluate the operations of the scanning sonar system.
It takes data collected in real time from the sonar pod mounted below the floating
instrument platform.  This previously recorded data is in a special file format
that this host software reads and evaluates.

*/


// Headers

#include <stdio.h>
#include <ctype.h>
#include <iostream.h>          // Placed here only to use cout for debugging
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>


// Constant Decloration

const int LmX = 151;          // Total number of array elements in X-axix
const int LmY = 141;          // Total number of array elements in Y-axis
const int LmZ = 141;          // Total number of array elements in Z-axis
const int CnY = 70;           // Y position in center of array elements
const int CnZ = 70;           // Z position in center of array elements
const int EvSze = 200;        // Number of Events that the system can handle.
const int EvLife = 36;        // Longivity of an Event, aged in seconds


// File Variable Declorations

FILE *EvntFl;                 // Event file created by Processor B
FILE *HdVel;                  // Heading and Velocity file (Created by Probe)
FILE *Sphr;                   // Sphere points used to create spheres
FILE *Pnts;                   // Entry points vs range for ending of Sphere data
FILE *Printer;                // Printer file for outputting slices
FILE *XYfile;                 // XY Plotting file


// Global Variable Declarations

float X_Evnt, Y_Evnt, Z_Evnt; // Event X, Y, Z
int Range_Evnt;               // Actual range / 2 to scale to WS array
char Time_Evnt[10];           // Time Event occurred
char Tstr[10];                // Time String Variable for function input
int LinTime;                  // Seconds, Minitues, Hours --> int LinTime
float X_Last, Y_Last,Z_Last;  // Holding variables for times ahead of clock
int Range_Last;               // Holding variable for Range ahead of clock
char Time_Last[10];           // Time variable for reading ahead of systems clock
bool Last_Full = false;       // If true data has been read ahead of system time
bool Time_Match;              // Indicates whether or not time same as last
char String[40];              // General purpose global string
float Head_Sum = 0;           // Sum of Heading values for a single time
char Head_Last[10];           // Last Heading value for reads ahead of system time
char Vel_Last[10];            // Last Velocity value for reads ahead of system time
int NrHd_Lst = 0;             // Last count value for averaging heading (> system time)
float Velocity = 0;           // Returned vehicle raw velocity value
float Heading;                // Returned vehicle heading value
float Old_Heading;            // Heading one second ago
bool LastCV_Full = false;     // Is there values beyond system time variable
float XP, YP, ZP;             // Global X, Y, Z location variables for rotations
char Next_Tstr[10];           // Returns next sequential file time from Events() and
                              //   HdVel()
bool Flip_Flop = false;       // Flip_Flops (Translate First <---> Rotate First)
float Boat_Vel = {0};         // Boat_Velocity - If not entered then obtained from Turbine
time_t Time;                  // Time variable for making one-second ticks
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//      Global ARRAY Variables Declarations

char Water_Space[LmX][LmY][LmZ];      // 3-D Array for Occupancy Grid
char Spheres[41852][3];               // X, Y, Z Delta Values for forming Spheres
int RngPts[300];                      // Element# = Range, Value is pointer into WS Array

// Event Table Arrays
int Active[EvSze] = {0};         // '0' If location empty, '1' If location full
float X_Curr[EvSze];             // Current Values for Events in X, Y, and Z
float Y_Curr[EvSze];
float Z_Curr[EvSze];
int SphRng[EvSze];               // Range of initial target
int Tm_Exit[EvSze] = {0};        // Time Event is to be removed from system (Seconds)


//      Function Decloration Area (See functions for explainations)

void Place_Sphere(int X_Pos, int Y_Pos, int Z_Pos, int SpRg, char Incr);
void Init_Array(void);
bool Rotate_LR(float Radians);
bool Translate(float Feet);
void Slice(int Depth);
void Pslice(int Depth);
void Table_Event(float X, float Y, float Z, int Rnge, char TimeStr[10]);
bool Event(void);
bool HeadVel(void);
void Open_Files(void);
bool Fill_Tables(void);
void Clock_Tick(void);
void StrTmInc(void);
int  MkTimeInt(char T_String[10]);
void Maint_Table(void);
void OpenXYfile(void);
void CloseXYfile(void);
void XY_Log(void);
float Conv_Vl_Str(char Str[10]);


// Main Line Program
void main()
{
        bool Data;              // General Data Available flag
        bool DaEvnt;            // Data available flag variable for Events
        bool DaHdVl;            // Data available flag variable for Heading and Velocity
        bool MoreData = {true};// More data to be read indicator
        char FrTmEvnt[10];      // First Event time after starting synchronization
        char FrTmHdVl[10];      // First Heading and Velocity time after sychronization
        char LstMnTm[10] = {NULL};     // Last Maint Time.  Keeps from over rotating and
                                       //   translating
        int I = 0;
        int Seconds = 0;               // Seconds counter for main line looping

        // Housekeeping Operations
        Open_Files();           // Open all necessary files.  If error exit program.
        Data = Fill_Tables();   // Place data file table information into arrays.
        Init_Array();           // Make Water_Space Array elements equal zero.


        // Events file and Heading and Velocity file synchronization
        // This program section assures that both of these data input files begin at a
        // common time.  Often this data does not.  Some data is wasted but usually only
        // a few elements.
        strcpy(Tstr, "");       // Make Tstr 0 length to find starting time in events.
        do
        {
                DaEvnt = Event();       // Get first Event with time.
        }
        while (Time_Match);             // When finished Next_Tstr has the NEXT time.
        strcpy(FrTmEvnt, Next_Tstr);  // Store next event time in FrTmEvnt.
        strcpy(Tstr, "");       // Make Tstr zero length to find starting time for HdVls
        DaHdVl = HeadVel();                           // Get the first HdVel with time.
        strcpy(FrTmHdVl, Next_Tstr);  // Store NEXT HdVel time in FrTmVl.
        while (strcmp(FrTmEvnt, FrTmHdVl) < 0)        // Loops only if Event time < HdVel
        {
                strcpy(Tstr, FrTmEvnt);         // Store Event time in Tstr
                do                              // Loop till Event caught up with HdVel
                {
                        DaEvnt = Event();       // Go get an event
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        }
        while (Time_Match);             // There are more events
        strcpy(FrTmEvnt, Next_Tstr);  // Load Next_Event time into FrTmEvnt
}
// Finishes loop when two times are equal
while (strcmp(FrTmEvnt, FrTmHdVl) > 0)       // Loops only if Event time > HdVel
{
        strcpy(Tstr, FrTmHdVl);              // Store HdVel time in FrTmHdVl
        DaHdVl = HeadVel();                  // Get next time SET
        strcpy(FrTmHdVl, Next_Tstr);         // New_Tstr is the new time
}
strcpy(Tstr, Next_Tstr);
// End of file synchronization section.
// The variable Tstr contains the next time to read.

cout << "Current Event time: " << FrTmEvnt << endl;
cout << "Current HDVEL time: " << FrTmHdVl << endl;
cout << "Enter Boat Velocity > ";
cin >> Boat_Vel;

Old_Heading = Heading;                  // Set heading initially for Angle variable

Data = true;
do
{
        Clock_Tick();                   // Hang till the current second finishes
        char Cmd;
        cout << Seconds << endl;        // Show elapsed seconds
        if (MoreData)                   // Get Head - Vel only if still reading data
        {
                DaHdVl = HeadVel();     // Heading and velocity fetch
        }
        do                              // Do until finished with processing
        {
                if (MoreData)           // Get Event only if still reading data
                {
                        DaEvnt = Event();       // Event read
                }
                if (!DaHdVl || !DaEvnt)         // If either false - finished reading
                {
                        MoreData = false;       // MoreData false - never true again
                }

///////////////// I T E R A T I V E   C O R E   O F   P R O G R A M ////////////////////
                if (MoreData)  // Continue entering events if more data available
                {
                        if (strcmp(LstMnTm, Tstr) != 0)
                        {
                                Maint_Table();          // Update table information
                                strcpy(LstMnTm, Tstr);
                        }
                        if (Time_Match)
                        {
                                Place_Sphere(floor(X_Evnt+.5),floor(Y_Evnt+.5),
                                floor(Z_Evnt+.5),Range_Evnt,1);
                                Table_Event(X_Evnt, Y_Evnt, Z_Evnt, Range_Evnt,
                                 Tstr);         // Put event into table
                        }
                        Slice(-2);              // Rough graphic to console
                        printf("L = Log X,Y Points > ");      // Print Log prompt.
                        Cmd = getchar();        // Get one character with <CR>
                        if ((Cmd == 'L') || (Cmd == 'l'))     // Was it a Log?
                        {
                                XY_Log();       // If Log then logit
                        }
                }
                else
                {
                        if (strcmp(LstMnTm, Tstr) != 0)
                        {
                                Maint_Table(); // No more data - just maint table
                                strcpy(LstMnTm, Tstr);
                        }
                        Time_Match = false;     // Continue loop
                        Slice(-2);              // Rough graphic to console
                        printf("L = Log X,Y Points > ");      // Print Log prompt
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                                        Cmd = getchar();        // Get one character with <CR>
                                        if ((Cmd == 'L') || (Cmd == 'l'))     // Was it a Log?
                                        {
                                                XY_Log();       // If Log then logit
                                        }
                                }
////////////////////////////////////////////////////////////////////////////////////
                }
                while (Time_Match);             // If matching time for Events continue
                                                //   reading
                StrTmInc();                     // Increment Ex:  09:03:38 --> 09:03:39
                LinTime = MkTimeInt(Tstr);      // Linear Time from string time (aging)
                Seconds++;                      // Elapsed time incrementation
        }
        while (Data);                           // When false program is finished
}
// End of Main Line


// Maint_Table() is used to update every entry in the Event Table one at a time.
// If an entry is out of bounds such as having a negative X value, or over 30 units
// from the Y-axis, the entry is removed.
void Maint_Table(void)
{
        int I;                                  // Incrementation variable
        bool OutBnd1, OutBnd2;                  // Out of Bounds boolean variables
        double Angle;                           // Angle of turn over the last second (-+)
        float Degrees;                          // Holds delta heading.

        cout << "Maint_Table" << endl;
        Degrees = Heading - Old_Heading;        // Find Delta angle
        if (Degrees > 180)                      // 359 <-- 0 crossing Case I
        {
                Degrees = -(Degrees - 360);     // Obstruction turning opposite vehicle
        }
        else
        {
                if (Degrees < -180)             // 359 --> crossing Case II
                {
                        Degrees = -(Degrees + 360);   // Adjust by adding 360
                }
                else
                {
                        Degrees = -Degrees;     // No adjustment necessary
                }                               //   just make negative
        }
        Old_Heading = Heading;          // Record Heading one second ago.
        Angle = Degrees / 57.296;       // Convert degrees to radians

        if (Flip_Flop)                  // Flip_Flop changes each time this function is
        {                               //   called to translate then rotate, then rotate
                Flip_Flop = false;      //   then translate.
        }
        else
        {
                Flip_Flop = true;       // Change order to true
        }
        for (I = 0; I < EvSze; I++)     // Iterate through each entry in the event table
        {
                if (Active[I] == 1)     // If there is an entry at this position
                {
                        XP = X_Curr[I];         // Get current X, Y, and Z values
                        YP = Y_Curr[I];
                        ZP = Z_Curr[I];
                        // Remove the previously placed sphere with a (-1).
                        Place_Sphere(floor(XP + .5), floor(YP + .5),
                        floor(ZP + .5), SphRng[I], -1);
                        if (Boat_Vel != 0)
                        {
                                Velocity = Boat_Vel / 2;      // Scales Velocity to WS Array
                        }
                        if (Tm_Exit[I] > (LinTime + 1))
                        {
                                if (Flip_Flop)          // If true - Rotate then Translate
                                {
                                        OutBnd1 = Rotate_LR(Angle);   // R then T
                                        OutBnd2 = Translate(Velocity);
                                }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                                        else
                                        // If false - Translate then Rotate
                                        {
                                                OutBnd2 = Translate(Velocity);       // T then R
                                                OutBnd1 = Rotate_LR(Angle);
                                        }
                                        if ((OutBnd1) || (OutBnd2))   // Was T or R out of bounds?
                                        {
                                                Active[I] = 0;         // O of B - Remove the entry
                                        }
                                        else
                                        // Else NOT O_of_B - Place at new location
                                        {
                                                Place_Sphere(floor(XP + .5), floor(YP + .5),
                                                floor(ZP + .5), SphRng[I], 1);
                                        }
                                        // Update values in event table
                                        X_Curr[I] = XP;
                                        Y_Curr[I] = YP;
                                        Z_Curr[I] = ZP;
                                }
                                else
                                {
                                        Active[I] = 0;         // Time limit exceeded - remove
                                }
                        }
                }
        }


// Functions Definitions Area

// EVENT() returns event information from the events file.  It is called with
// Tstr="" returns the first time recorded with all global variables filled.
// The current value of Tstr must be used until there is a false returned by
// the function itself at which time the Tstr can be incremented to the next
// second and so forth. Global variables returned with data are:
//    X_Evnt, Y_Evnt, Z_Evnt, and Range_Evnt
//
// Time is sent to this function via the Tstr global string.
// Function itself returns a boolean variable reflecting file data available.
bool Event(void)
{
        bool DataAvail = (true);       // Data Available flag
        char Buffer[40];               // String storage buffer
        char R, I;                     // Iteration and working char variables
        char Temp_X[10] = "", Temp_Y[10] = "", Temp_Z[10] = "";     // Working string vars
        char Temp_Range[10] = "", Temp_Time[10] = "";

        Time_Match = false;            // Assume no time match to start with
        if (Last_Full)                 // One event value pre-read so use it
        {
                if (strcmp(Tstr, Time_Last) == 0)     // Compare current time to last time
                {
                        X_Evnt = X_Last;       // If same time then use pre-stored vars
                        Y_Evnt = Y_Last;
                        Z_Evnt = Z_Last;
                        Range_Evnt = Range_Last;
                        strcpy(Time_Evnt, Time_Last);
                        Time_Match= true;      // This time = previous time flag
                        Last_Full = false;     // No last read vars waiting
                }
        }
        else
        {                                      // Was no previously read variables to load
                do
                {
                        DataAvail = (fgets(Buffer, 40, EvntFl) != NULL);     // Get Event
                }
                while ((DataAvail) && (Buffer[0] != char(32)));                 //
                if (DataAvail)
                {
                        for(I = 0; I < 18; I++)
                        // Scan Buffer to parse
                        {
                                if (I < 6)
                                // First value (X) string
                                {
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                        Temp_X[I] = Buffer[I];
                        // Temp_X is string X value
                  }
                  else
                  {
                        // Was not first(Second? Third?)
                        if (I < 12)
                        {
                              Temp_Y[I-6] = Buffer[I];
                              // Second number (Y) in string
                        }
                        else
                        {
                              // Must be third number (Z)
                              if (I < 18)
                              {
                                    Temp_Z[I-12] = Buffer[I];
                                    // Transfer to Temp_Z
                              }
                        }
                  }
            }
            for (I = 19; I < 22; I++)     // Skip "R" for range go to Rnge Val
            {
                  R = Buffer[I];          // Get each character of range
                  if (isdigit(R))         // Is character in R a digit?
                  {
                        Temp_Range[I-19] = R;  // Form Temporary Range string
                  }
            }
            DataAvail = (fgets(Buffer, 40, EvntFl) != NULL);
            // Get date part of event
            if (DataAvail)
            {
                  for (I = 1; I < 9; I++)       // Scan date string just read
                  {
                        Temp_Time[I-1] = Buffer[I];
                        // Create Temp_Time - Time string
                  }
                  X_Evnt = atof(Temp_X) / 2;    // Divided by 2 to Scale to WS
                  Y_Evnt = atof(Temp_Y) / 2;    // 300 feet is 150th X element
                  Z_Evnt = atof(Temp_Z) / 2;
                  Range_Evnt = atoi(Temp_Range) / 2;    // Do same with range
                  strcpy(Time_Evnt, Temp_Time);         // Last time string
                  if (strlen(Tstr) == 0)                // First time called?
                  {
                        strcpy(Tstr, Temp_Time);        // Tstr = current time
                  }
                  if (strcmp(Tstr, Temp_Time) != 0)     // Tstr<>current time
                  {
                        X_Last = X_Evnt;                // Load 'Last'
                        Y_Last = Y_Evnt;
                        Z_Last = Z_Evnt;
                        Range_Last = Range_Evnt;
                        strcpy(Time_Last, Time_Evnt);
                        Last_Full = true;       // Indicate 'Last' variables
                        Time_Match = false;     // Indicate time did NOT match
                        strcpy(Next_Tstr, Temp_Time);
                        // The next Tstr is indicated by Next_Tstr
                  }
                  else
                  {
                        // Still same time - No last variables
                        Last_Full = false;
                        Time_Match = true;      // Time match with old time
                  }
            }
      }
}
      return DataAvail;                                                   //
DataAvail = true means not EOF yet
};

// HEADVEL() called with Tstr="" returns the first time recorded with all
//     global variables filled.  After the first call to this function the
//     variable Tstr must be loaded with the next sequential time in the form
//     09:03:38 and following 09:03:39 etc.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//      Global variables returned with data are:
//              (Velocity) and (Heading)
//      Heading and Velocity are floating point data type.
//
//      Time is sent to this function via the Tstr global string.
//  Function itself returns a boolean variable reflecting file data available.
bool HeadVel(void)
{
        bool HV_Avail;                  // Data available flag
        bool STm;
        char Buffer[40];                // Working buffer for reads
        char Temp_Head[10] = "";        // Temp Heading string
        char Temp_Vel[10] = "";         // Temp Velocity string
        char HV_Time[10] = "";          // Time string for heading and velocity
        int NrHd = 0, T_NrHd = 0;       // Heading received counters for average


        if (LastCV_Full)                // Was there a last valid reading?
        {
                Head_Sum = atof(Head_Last);   // Start new sum from Last-Head value
                if (strlen(Vel_Last) > 0)     // If there was a velocity reading...
                {
                        Velocity = Conv_Vl_Str(Temp_Vel);
                }
                NrHd = NrHd_Lst;              // Heading counter
        }

        do
        {
                do
                {
                        HV_Avail = (fgets(Buffer, 40, HdVel) != NULL);
                        // Read Heading and Velocity
                }       //
                while ((HV_Avail) && (Buffer[0] != char(67)) && (Buffer[0] != char(86)));
                // Avail,C,V
                if (HV_Avail)
                {
                        strcpy(String, Buffer);       // Buffer into String - Check later
                        HV_Avail = (fgets(Buffer, 40, HdVel) != NULL);
                        // Get second read from file
                        strcpy(Temp_Head, "");        // Clear Temp Heading string to ""
                        strcpy(Temp_Vel, "");         // Clear Temp Velocity string to ""
                        if (HV_Avail)                 // Was there data read?
                        {
                                for(int I = 1; I < 9; I++)    // Parse the time portion
                                {
                                        HV_Time[I - 1] = Buffer[I];
                                        // Reconstruct time for HV
                                }
                                if (String[0] == char(67))    // Is String[0] a 'C'?
                                {
                                        String[0] = ' ';      // Clear out the 'C'
                                        strcpy(Temp_Head, String);    // Copy to Temp String
                                        T_NrHd = 1;
                                }
                                else
                                {       // Must be a 'V'
                                        String[0] = ' ';              // Clear out the 'V'
                                        strcpy(Temp_Vel, String);     // Copy to Temp String
                                }
                                if (strlen(Tstr) == 0)
                                // If Tsrt="" then Tsrt=First Record Time
                                {
                                        // Normal usage (First call to function)
                                        strcpy(Tstr, HV_Time);
                                }
                                if (strcmp(Tstr, HV_Time) == 0)
                                // Compare current time with read time
                                {
                                        if (strlen(Temp_Head) > 0)
                                        // Don't consider if not new
                                        {
                                                Head_Sum = Head_Sum + atof(Temp_Head);
                                                NrHd = NrHd + T_NrHd;
                                                // Number of entries for average
                                        }
                                        if (strlen(Temp_Vel) > 0)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                                        // Don't consider if not new
                                        {
                                                Velocity = Conv_Vl_Str(Temp_Vel);
                                                // Take Velocity from Turbine
                                        }
                                        STm = true;     // Same time flag
                                }
                                else
                                {
                                        // Finished with a Seconds Time Interval
                                        if (NrHd)
                                        {
                                                Heading = Head_Sum / NrHd;
                                                // Calculate Average Heading
                                        }
                                        Head_Sum = 0;  // Zero Head Sum
                                        NrHd = 0;       // Set element count for average = 0
                                        NrHd_Lst = T_NrHd;
                                        strcpy(Head_Last, Temp_Head);
                                        // Currently good Temp_Heading
                                        strcpy(Vel_Last, Temp_Vel);
                                        // Load possibly new values for later
                                        LastCV_Full = true;
                                        // LastCV_Full will never be false again.
                                        STm = false;            // Exit we are finished
                                        strcpy(Next_Tstr, HV_Time);
                                }

                        }

                }

        }
        while ((HV_Avail) && (STm));            // While data is available and Same Time
        return HV_Avail;                        // To determine if more data is to be read
};

// Init_Array() clears out all Water_Space array elements to zero.
void Init_Array(void)
{
        for (int X = 0; X < LmX; X++)                   // Iterate X values through range
        {
                for (int Y = 0; Y < LmY; Y++)           // Iterate Y values through range
                {
                        for (int Z = 0; Z < LmZ; Z++)   // Iterate Z values through range
                        {
                                Water_Space[X][Y][Z] = 0;
                                // Put them all in Water_Space array
                        }
                }
        }
}



// Place_Sphere() both places a sphere in Water_Space and also removes a sphere
// from Water_Space.  When placing a sphere set value passed to Incr to (+1).
// This adds one to all sphere elements in the array.  When moving toward removal
// of a sphere place (-1) in increment.  This subtracts the same amount added but
// does not clear out the element if another sphere is co-located using the same
// elements.  Y_Pos and Z_Pos are referenced to the Y and X axis, not 0 element of
// the WS array.
void Place_Sphere(int X_Pos, int Y_Pos, int Z_Pos, int SpRg, char Incr)
{
        int EndPoint;                   // End point of look-up into Sphere array
        int I;                          // Iteration variable
        int X, Y, Z;                    // Temporary X, Y, and Z variables

        Y_Pos = Y_Pos + CnY;            // Far left Y is zero, so correct Y
        Z_Pos = Z_Pos + CnZ;            // Correct Z as well. (X needs no correction)
        EndPoint = RngPts[SpRg];        // Get EndPoint from Range Points array
        for (I = 1; I <= EndPoint; I++)         // Color sphere elements till Endpoint.
        {
                X = X_Pos + Spheres[I][0];      // X_Pos has Delta X added from Sphere array.
                Y = Y_Pos + Spheres[I][1];      // Y_Pos has Delta Y added from Sphere array.
                Z = Z_Pos + Spheres[I][2];      // Z_Pos has Delta Z added from Sphere array.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                if ((X >= 0) && (X <= LmX))           // Check X for limits of WS array.
                {
                        if ((Y >= 0) && (Y <= LmY))   // Check Y for limits of WS array.
                        {
                                if ((Z >= 0) && (Z <= LmZ))          // Do same for Z
                                {
                                        // If limit checks passed - Place in WS.
                                        Water_Space[X][Y][Z] = Water_Space[X][Y][Z] + Incr;
                                }
                        }
                }
        }
}


// Rotate_LR() rotates X, Y, and Z variables around the (Z) axix.  These variables
//  reference the Water_Space array.  Returned global variables are XP, YP, and ZP
bool Rotate_LR(float Radians)
{
        float X_Old, Y_Old;

        X_Old = XP;
        Y_Old = YP;
        XP = X_Old * cos(Radians) - Y_Old * sin(Radians);    // Rotate X
        YP = X_Old * sin(Radians) + Y_Old * cos(Radians);    // Rotate Y
        // ZP Is unchanged
        if ((XP < 0) || (YP < -100) || (YP > 100))          // Check for Out of Bounds
        {
                return true;            // Yes, Out of Bounds
        }
        else
        {
                return false;           // No, Not Out of Bounds
        }
}


// Translate() translates vehicle forward by floating point variavle Feet.
bool Translate(float Feet)
{
        XP = XP - Feet;                 // Global XP variable changed by Feet.
        return (XP < 0);                // If XP still positive then no O_of_B condition
}



// Table_Event() is used to load a newly generated event into the event table.
// It must be called with X, Y, and Z coordinates, range, and a time string.
void Table_Event(float X, float Y, float Z, int Rnge, char TimeStr[10])
{
        int  I;
        char T_Hr[4] = {NULL};          // Temp Hours string
        char T_Mn[4] = {NULL};          // Temp Minutes string
        char T_Sc[4] = {NULL};          // Temp Seconds string

        I = 0;
        while((Active[I] != 0) && (I < EvSze))
        // Keep going till vacancy or end of table
        {
                I++;
        }
        if (I < EvSze)
        {
                Active[I] = 1;          // Make table locations active
                X_Curr[I] = X;          // Place X, Y, and Z into of Event Table
                Y_Curr[I] = Y;
                Z_Curr[I] = Z;
                SphRng[I] = Rnge;       // Store range
        }
        Tm_Exit[I] = MkTimeInt(TimeStr) + EvLife;    // Make linear ending time integer
}


// Open_Files() opens all necessary files at once including the printer file for output.
// If any file does not exist the program is terminated by exit().
void Open_Files(void)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
        if ((Sphr = fopen("Sphere.Dat","r")) == NULL)       // Data to form spheres
        {
                printf("UNABLE TO OPEN FILE <Sphere.DAT>\n");
                exit(2);
        }
        if ((Pnts = fopen("Entry.Dat","r")) == NULL)        // Table for entry into 'Sphr'
        {
                printf("UNABLE TO OPEN FILE <Entry.DAT>\n");
                exit(2);
        }
        if ((EvntFl = fopen("Events.txt","r")) == NULL)     // Events file
        {
                printf("UNABLE TO OPEN FILE <Events.txt>");
                exit(2);
        }
        if ((HdVel = fopen("DirSpd.txt","r")) == NULL)      // Direction and Speed data
        {
                printf("UNABLE TO OPEN FILE <DirSpd.txt>");
                exit(2);
        }
}


// Fill_Tables() fills the Sphere Table and also the Range Point Table.
// This is done so as to be able to read directly from array and not disk file.
bool Fill_Tables(void)
{
        int I;                          // Incrementation variable
        int Rp = 1;                     // Array pointer
        bool SDAval, RDAval;            // Data Available variables
        char Buffer[40];                // String storage variables
        char TempChr[10];
        do
        {
                SDAval = (fgets(Buffer, 40, Sphr) != NULL);  // Read a Sphere value
                if (SDAval)
                // Do IF only if data is available
                {
                        TempChr[0] = NULL;                  // Init string to ""
                        for (I = 5; I < 10; I++)            // Parse out X value
                        {
                                TempChr[I - 5] = Buffer[I]; // X goes into TempChr[]
                        }
                        Spheres[Rp][0] = char(atoi(TempChr));
                        // Place X value into Sphere array
                        TempChr[0] = NULL;                  // Init string
                        for (I = 10; I < 15; I++)           // Parse out Y value
                        {
                                TempChr[I - 10] = Buffer[I]; // Y goes into TempChr[]
                        }
                        Spheres[Rp][1] = char(atoi(TempChr));
                        // Place Y value into Sphere array
                        TempChr[0] = NULL;                  // Init string
                        for (I = 15; I < 20; I++)           // Parse out Z value
                        {
                                TempChr[I - 15] = Buffer[I]; // Z goes into TempChr[]
                        }
                        Spheres[Rp][2] = char(atoi(TempChr));
                        // Place Z value into Sphere array
                        TempChr[0] = NULL;                  // Init string
                        Rp++;                               // Bump array pointer variable
                }
        }
        while (SDAval);         // As long as Data is available
        // Fill Range Points array
        Rp = 1;
        // Start with array pointer = 1
        do                      // Do while data available
        {
                RDAval = (fgets(Buffer, 40, Pnts) != NULL);  // Get Range Points from file
                if (RDAval)     // If data is available still
                {
                        for (I = 0; I < 6; I++)      // ASCII Range Points are 6 chars
                        {
                                TempChr[I] = Buffer[I];
                                // Build all six charactors for RPs
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                        }
                        RngPts[Rp] = atoi(TempChr);    // Load RPs into array as conv int's
                        Rp++;                          // Point to next array element
                }
        }
        while (RDAval);                                // Keep going till no more data.
        return RDAval;
}

// Clock_Tick() is used to end a one second interval. When called it reads
//  time during a second and loops until the second is completed.
void Clock_Tick(void)
{
        Time = time(NULL);                     // Read current seconds time
        while (Time == time(NULL))             // Dummy loop till second ends
        {
        }
}


// StrTmInc() is a function that takes Tstr and increments it to the next second.
// The function returns the global variable Tstr incremented.
void StrTmInc(void)
{
        int Sec, Min, Hr;              // Temporary variables for time

        int I;                         // Incrementing variable

        char S1[4] = {NULL};           // Temporary storage strings
        char S2[4] = {NULL};
        char S3[4] = {NULL};

        for(I = 0; I < 9; I++)         // Look through string
        {
                if (I<2)
                {
                        S1[I] = Tstr[I];                // Make Hrs string
                }
                else
                {
                        if ((I > 2) && (I < 5))         // Make Minutes string
                        {
                                S2[I - 3] = Tstr[I];
                        }
                        else
                        {
                                if ((I > 5) && (I < 8))         // Make Seconds string
                                {
                                        S3[I - 6] = Tstr[I];
                                }
                        }
                }
        }
        Hr = atoi(S1);                         // Convert Hours string to numeric
        Min = atoi(S2);                        // Convert Minutes string to numeric
        Sec = atoi(S3) + 1;                    // Convert Seconds string to numeric
        if (Sec > 59)                          // Correct Seconds for rollover
        {
                Sec = 0;                       // Reset Seconds to zero
                Min++;                         // Increment Minutes
                if (Min > 59)                  // Correct Minutes for rollover
                {
                        Min = 0;               // Reset Minutes to zero
                        Hr++;                  // Increase Hours because of rollover
                        if (Hr > 23)           // Correct for 12 midnight
                        {
                                Hr = 0;        // Early in the morning
                        }
                }
        }
        Hr = Hr + 100;                         // Make sure Hours are at least 2 digits
        Min = Min + 100;                       // Do same for Minutes
        Sec = Sec + 100;                       // Do same for Seconds
        itoa(Hr, S1, 10);                      // Convert to string (Ex: 2 hours --> "102")
        itoa(Min, S2, 10);                     // Same for Minutes
        itoa(Sec, S3, 10);                     // Same for Seconds
        for (I = 0; I < 8; I++)                // Scan the result string as we construct it
        {
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                if (I < 2)                      // Hours portion of Tstr
                {
                        Tstr[I] = S1[I + 1];  // Take only right 2 digits for Hours
                }
                else
                {                               // Not first 2 digits
                        if ((I > 2) && (I < 5))       // Minutes portion of Tstr string
                        {
                                Tstr[I] = S2[I - 2];  // Use only right 2 digits of S2
                        }
                        else
                        {                             // Not Hours or Minutes - Seconds now
                                if (I > 5)
                                {
                                        Tstr[I] = S3[I - 5];  // Right 2 digits of Seconds
                                }
                        }
                }
        }
}


// MkTimeInt() receives a string in the form of '09:03:38' and returns the value
// of the string based on hours equal 3600 seconds, minutes equal 60 seconds, and
// seconds being equal to seconds.
int MkTimeInt(char T_Stg[10])
{
        int I;                          // Incrementing variable
        char T_Hr[4] = {NULL};          // Temporary time string storage
        char T_Mn[4] = {NULL};
        char T_Sc[4] = {NULL};

        for (I = 0; I < 8; I++)         // Scan time string
        {
                if (I < 2)              // If Hours portion
                {
                        T_Hr[I] = T_Stg[I];   // Load Hours string into temp storage
                }
                else
                {                                       // Was not Hours location
                        if ((I > 2) && (I < 5))         // If Minutes portion of string
                        {
                                T_Mn[I - 3] = T_Stg[I];       // Load Minutes string
                        }
                        else
                        {
                                // Was not Hours or Minutes
                                if ((I > 5) && (I < 8))       // Seconds portion of string
                                {
                                        T_Sc[I - 6] = T_Stg[I];
                                        // Load Seconds string into temp storage
                                }
                        }
                }
        }
        return atoi(T_Hr) * 3600 + atoi(T_Mn) * 60 + atoi(T_Sc);     // Make time integer
}


// Slice() takes a cross section of the Water_Space array at a given depth in the
// Z axix.  This Z plane is then printed to the screen for observation.  This is an
// extremely low resolution depiction of what is in the WS array.  It is only for
// identifying the presents and general configuration of data as the program runs.
void Slice(int Depth)
{
        int A;                  // Sample of WS array
        char J, K;              // Temporary variables

        cout << Tstr << endl;
        for (int X = 75; X >= 5; X = X - 5)             // Scan X values from high to low
        {
                printf("\n");
                for (int Y = 0; Y <= LmY - 1; Y = Y + 2)     // Scan Y values left to right
                {
                        A = 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                        for(J = 0; J > -5; J = J - 1)
                        {
                                for(K = 0; K > -21; K--)
                                {
                                        A = A + Water_Space[X + J][Y][Depth + CnZ + K];
                                        // Look into the array
                                }
                        }
                        if (A == 0)     // If nothing there print '-'
                        {
                                printf("-");   // Nothing at that location
                        }
                        else
                        {
                                printf("O");   // Something at that location (A => 1)
                        }
                }
        }
        printf("\n");                          // New line
        printf("End of Slice\n");              // Identify the end of a slice.
}


// OpenXYfile() opens the slice file for samples of the WS array.  'Fname' is the
// file name string used to name the opened file.  The first two characters of this
// name is always 'XY'.  The second two characters are the Hrs from the Tstr string.
// The third two characters are Mins from Tstr and the last two characters are Secs.
// This automatically labels the file for later reference as to the time it was
// recorded.  The file extension is '.TXT' for reasons of ease of display for debugging
// purposes.
void OpenXYfile(void)
{
        int I;                                 // Incrementing variable
        int J = {0};                           // String pointer
        char Fstrg[20] = {"XY"};               // Working string in making Fname
        char Fname[20] = {NULL};               // Final string for Name of file

        strcat(Fstrg, Tstr);                   // String now is 'XYhr:mn:sc'
        strcat(Fstrg, ".txt");                 // String now is 'XYhr:mn:sc.txt'
        for(I = 0; I < 14; I++)                // Scan the whole string
        {
                if ((I != 4) && (I != 7))      // Locations for ':' are left out
                {
                        Fname[J] = Fstrg[I];
                        // Reconstruct Fname from Fstrg with no colons
                        J++;                   // Pointer variable to Fname
                }
        }
        if ((XYfile = fopen(Fname,"w")) == NULL)       // Open file Fname
        {
                printf("UNABLE TO OPEN FILE <");       // Error message if not opened
                fputs(Fname, stdout);                  // Labels file not opened
                printf(">\n");                         // Finish with error reporting
                exit(3);                               // Quit everything
        }
}


// CloseXYfile() closes the file that the WS array slice was stored into.
void CloseXYfile(void)
{
        fclose(XYfile);                // XYfile is current slice file.
}


// XY_Log() is a function that stores a set of X,Y points into a file opened for this
// purpose.  This function inputs from the console a value for Depth with defines the
// Z plane level for a cross-section slice of the Water_Space Array.  This array is
// scanned and non-zero elements are defined as to their X and Y coordinates.  These
// coordinates are then loaded into a file with TAB delimiters and CRLF end of line
// characters.  This file is read by plottig software and graphed for observation of
// sonar targets.
void XY_Log(void)
{
        int A;                         // Element variable for testing
        int Z;                         // Variable to take "Dig" into Z direction
        int Depth;                     // Depth of slice through WS array
        char Temp[10] = {NULL};        // Temporary string for manipulation of input data
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        printf("\n");                   // New Line
        printf("Enter Depth > ");       // Prompt for depth of slice
        gets(Temp);                     // Dummy gets to purge new line character
        gets(Temp);                     // Real gets to load temp with Depth string
        Depth = atoi(Temp);             // Depth of slice taken in WS array
        OpenXYfile();                   // Open new file with 'Tstr' embedded in name
        for (int X = LmX - 1; X >= 0; X = X - 1)          // Scan through all X's
        {
                for (int Y = 0; Y <= LmY - 1; Y = Y + 1)     // Scan through all Y's
                {
                        A = 0;          // Init A to Zero each X, Y, Position
                        for(Z = 0; Z > -30; Z--)
                        {
                                A = A + Water_Space[X][Y][Depth + CnZ + Z];
                                // Read element of WS array
                        }
                        if (A > 0)      // If A=0 then NOT an XY point set
                        {
                                itoa((Y - CnY) * 2, Temp, 10);       // Y is displayed as X
                                fputs(Temp, XYfile);                 // Put into file
                                putc(9, XYfile);                     // Delimited by a TAB
                                itoa(X * 2, Temp, 10);               // X is displayed as Y
                                fputs(Temp, XYfile);                 // Put into file
                                putc(13, XYfile);                    // Return into file
                                putc(10, XYfile);                    // Line Feed into file
                        }
                }
        }
        printf("\n");           // New line
        CloseXYfile();          // Finished so close XYfile
}


// Conv_Vl_Str() takes a string[10] comprised of hexidecimal numbers from the velocity
// sensor converts these to clicks and then converts the clicks to actual velocity in
// feet per second.  This value is then returned to the program.
float Conv_Vl_Str(char Str[10])
{
        int I;                  // Iteration variable
        int J;                  // String array pointer
        int Digit;              // Holds value of Hex digit in decimal
        int Weight = {1};       // Progressive weighting of Hex digits
        char L;                 // Used for string length determination
        float Value = {0};      // Number of clicks
        char *Hexdig = "0123456789ABCDEF";   // Hex string for conversion purposes

        L = strlen(Str);                // Length of velocity string
        for(I = L; I >= 0; I--)         // Iterate through string back to front      {
                if (isxdigit(Str[I]))  // Returns > 1 if character is a Hex
                {
                        for(J = 0; J < 16; J++) // Look at each Hex character in order
                        {
                                if (Hexdig[J] == Str[I])      // Is there a match?
                                {
                                        Digit = J;     // Digit is value in decimal of H
                                        Value = Value + Digit * Weight;
                                        // Accumulate the value in (Value)
                                        Weight = Weight * 16; // Weights are 1, 16, 256...
                                        continue;  // Break execution of loop if finished
                                }
                        }
                }
        }
        // Currently (Value) contains the number of clicks per 3 second interval.  This
        // figure needs to be used as a look-up into a table of velocities.  However, at
        // this time no such table exists.  Therefore this variable returns the number of
        // rotations per 3 second time interval times 6.
        return Value;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX C

BASIC LANGUAGE SUPPORT SOFTWARE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program EPROM.BAS
     REM Program To Make 512k x 8 Eprom Memory For Processor A

     REM Make File Area and Fill with $FF's
     OPEN "R", 1, "BINARY", 1
     FIELD 1, 1 AS F$
     LSET F$ = CHR$(255)
     FOR I = 1 TO 524288
     PUT 1, I
     NEXT I

1050 REM Nexted Loops for EEPROM Address Calculation

     REM Transponder Number
     FOR TXP# = 0 TO 2

     REM Head Position
     FOR HP# = 0 TO 1

     REM Horizontal Angle
     FOR ANG# = 0 TO 5

     REM Range Value
     FOR RANGE# = 1 TO 300

     REM Actual EPROM Address Calculation
     ADDRESS# = TXP#*131072# + HP#*65536# + ANG#*8192# + RANGE#*16# + 1
     PRINT "AD: "; ADDRESS#

     IF ANG# < 2.5 THEN YS = (-1) ELSE YS = (1) : REM Sign of X value
     IF ANG# = 0 OR ANG# = 5 THEN XANGLE = .3490659# : REM Abs Val Angls
     IF ANG# = 1 OR ANG# = 4 THEN XANGLE = .2094395#
     IF ANG# = 2 OR ANG# = 3 THEN XANGLE = 6.981320000000001D-02

     ANBR = TXP# * 2 + HP# : REM Vertical Angle

     IF ANBR < 2.5 THEN ZS = (1) ELSE ZS = (-1) : REM Sign of Y value
     IF ANBR = 0 OR ANBR = 5 THEN YANGLE = .3490659# : REM Abs Val Angls
     IF ANBR = 1 OR ANBR = 4 THEN YANGLE = .2094395#
     IF ANBR = 2 OR ANBR = 3 THEN YANGLE = 6.981320000000001D-02

     Z# = RANGE# * SIN(YANGLE) * ZS : REM Z Axis Value

     SPAN# = RANGE# * COS(YANGLE) : REM Projection on Z=0 Plane

     Y# = SPAN# * SIN(XANGLE) * YS : REM Y Axis Value
     X# = SPAN# * COS(XANGLE) : REM X Axis Value

     BYTE0 = 0 : REM First Byte has Code Representing Sign of YS/ZS
     IF YS < 0 THEN BYTE0 = BYTE0 + 2
     IF ZS < 0 THEN BYTE0 = BYTE0 + 1
     S1$ = MID$(STR$(X#) + SPACE$(20), 2, 5) : REM Make Stgs for X,Y,Z
     S2$ = MID$(STR$(Y#) + SPACE$(20), 2, 5)
     S3$ = MID$(STR$(Z#) + SPACE$(20), 2, 5)

     W$ = S1$ + S2$ + S3$ : REM Combine into Larger String

     LSET F$ = CHR$(BYTE0) : REM Place Code in Memory
     PUT 1, ADDRESS#
     ADDRESS# = ADDRESS# + 1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
REM Place String into memory
FOR I = 1 TO 15
G$ = MID$(W$, I, 1)
LSET F$ = G$
PUT 1, ADDRESS#
ADDRESS# = ADDRESS# + 1
NEXT I
NEXT RANGE#
NEXT ANG#
NEXT HP#
NEXT TXP#
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program VECTOR.BAS
     REM Program to Create Sphere Prototype

     REM Open Temporary File
     OPEN "R", 1, "SPHERE.INT", 27
     FIELD 1, 5 AS DIST$, 5 AS FX$, 5 AS FY$, 5 AS FZ$, 5 AS RG$, 2 AS
       CRLF$
     LSET CRLF$ = CHR$(13) + CHR$(10) : REM Will Not SORT Without CR/LF
     RP = 1

     REM Iterate through all Possible Values of X, Y, and Z.
     FOR X = -22 TO 22
     FOR Y = -22 TO 22
     FOR Z = -22 TO 22

     REM Calculate Range Section
     BSE = SQR(X * X + Y * Y) : REM Projection on Z=0 Plane (Base)
     DIST = SQR(BSE * BSE + Z * Z) : REM Actual Distance to Endpoint
     DIST = FIX(DIST + .5) : REM Round DIST to nearest Integer

     REM Load into Record
     RSET DIST$ = STR$(DIST) : REM Place Values into File Field
     RSET FX$ = STR$(X)
     RSET FY$ = STR$(Y)
     RSET FZ$ = STR$(Z)
     RG = DIST / (.0699268) : REM Not Used for Later Calculations
     RG = FIX(RG): REM Not Used for Later Range Calculations
     RSET RG$ = STR$(RG)

     REM Put Record into File
     PUT 1, RP : REM Place into File
     RP = RP + 1 : REM Increment Storage Pointer
     CNT = CNT + 1 : REM Just for Observation
     TRY = TRY + 1
     IF TRY = 500 THEN PRINT CNT: TRY = 0

     NEXT Z
     NEXT Y
     NEXT X


     REM Note: After creation of file SPHERE.INT the MS-DOS SORT
     REM       utility is used to create SPHERE.DAT file.  This
     REM       file is used by the Host Computer.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program SCAN_SPH.BAS

     REM Scans the Sphere.Dat file for the first field containing
     REM range information then stores it in an Index file named
     REM INDEX.TSH which is used later in creation of Endpoints for
     REM the sphere build operation.

     OPEN "I", 1, "SPHERE.DAT"
     OPEN "R", 2, "INDEX.TSH"
     FIELD 2, 4 AS F$
     LR = 1

     REM Loop Till Finished
1100 IF EOF(1) THEN PRINT LR - 1: END
     INPUT #1, L$
     S$ = LEFT$(L$, 4)
     LSET F$ = S$
     PUT 2, LR
     LR = LR + 1
     PRINT S$
     GOTO 1100
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program FIND_SZ.BAS
     REM Determines the Max Radius for a Sphere at a given range.
     REM Then it searches the Sphere.Dat file to find the EndPoints
     REM for the Sphere Build Process.
     REM
     REM This program is SLOW due to its use of a sequential search
     REM rather than more elegant means.  However, it should only have
     REM to run once.

     REM  Let N = Max Radius of a Sphere
     REM  Let D = Actual Diameter of affected Water_Space Sphere
     REM  D = 2 * N + 1
     REM  Let RANGE = Actual (OR) Scaled Range to Target
     REM  Tan (4 Degrees) = D / Range
     REM  N = Range * Tan (4deg) +1

     OPEN "R", 1, "INDEX.TSH": REM Radius Information from Sphere.Dat
     FIELD 1, 4 AS F$

     OPEN "R", 2, "ENTRY.DAT", 8: REM File to be used by Host.cpp
     FIELD 2, 6 AS RPT$, 2 AS CRLF$: REM To be read by Host Program
     LSET CRLF$ = CHR$(13) + CHR$(10)

     REM Loop through ranges
     FOR RANGE = 1 TO 300
     N = .07 * RANGE - .5 : REM Calculate N values
     N = FIX(N + .999999) : REM Round N values
     IF N = 0 THEN N = 1 : REM No N=0's

     REM Loop and search for Endpoints
     I = 1
1500 GET 1, I
     IF (VAL(F$) > N) OR (I = 41851) THEN 2000
     I = I + 1
     GOTO 1500

     REM Create Endpoint file
2000 V = I - 1
     LSET RPT$ = STR$(V)
     PUT 2, RANGE
     PRINT RANGE
     NEXT RANGE
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program RECORDER.BAS (RECORDER.EXE)
     REM Data Recorder Program

     CLS
     REM Open files
     OPEN "R", 5, "FILEINFO.DAT", 10
     FIELD 5, 10 AS F$
     IF LOF(5) > 0 THEN 1050
     LSET F$ = "000" : REM If first time then create
     FOR I = 1 TO 9
     PUT 5, I
     NEXT I
     LSET F$ = "1"
     PUT 5, 6
1050 DISP = 0: REM DO NOT SHOW CALLS TO 9800
     REM Get Gain, Attenuation, Width from stored values
     GET 5, 1
     GAIN$ = LEFT$(F$, 3)
     GET 5, 2
     ATTN$ = LEFT$(F$, 3)
     GET 5, 3
     WIDTH$ = LEFT$(F$, 3)
     GET 5, 6
     FILE = VAL(F$) : REM Next file starting number
     GOSUB 9500: REM STOP SYSTEM

     REM Set initial conditions.
     S$ = ":G" + GAIN$
     GOSUB 9800
     S$ = ":A" + ATTN$
     GOSUB 9800
     S$ = ":W" + WIDTH$
     GOSUB 9800
     S$ = ":B001" : REM Baud Rate
     GOSUB 9800
     S$ = "<F" : REM Fast baud rate
     GOSUB 9800

     REM Screen drawing section
     LOCATE 1, 1
     PRINT CHR$(201);
     PRINT STRING$(78, 205);
     PRINT CHR$(187);
     FOR Y = 2 TO 23
     LOCATE Y, 1, 1, 0, 7
     PRINT CHR$(186);
     NEXT Y
     FOR Y = 2 TO 22
     LOCATE Y, 80
     PRINT CHR$(186);
     NEXT Y
     LOCATE 23, 1
     PRINT CHR$(200);
     PRINT STRING$(78, 205);
     PRINT CHR$(188);

     REM Print current settings on screen and options
     X = 10: Y = 3
     LOCATE Y, X
     PRINT "Gain:   "; GAIN$;
     Y = Y + 2
     LOCATE Y, X
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      PRINT "Atten:  "; ATTN$;
      Y = Y + 2
      LOCATE Y, X
      PRINT "Width:  "; WIDTH$;
      Y = Y + 3
      LOCATE Y, X
      PRINT "Thres:  "; TN$;
      X = 1: Y = 21
      GOSUB 9700: REM DRAW HORIZONTAL LINE
      X = 40: Y = 6
      LOCATE Y, X
      PRINT "_____ COMMANDS _____"
      Y = Y + 1
      LOCATE Y, X
      PRINT "S = STOP SYSTEM"
      Y = Y + 1
      LOCATE Y, X
      PRINT "R = RUN AND RECORD"
      Y = Y + 1
      LOCATE Y, X
      PRINT "C = CENTER HEAD"
      Y = Y + 1
      LOCATE Y, X
      PRINT "I = INITIALIZE HEAD"
      Y = Y + 1
      LOCATE Y, X
      PRINT "T = THRESHOLD INSTALL"
      Y = Y + 1
      LOCATE Y, X
      PRINT "M = MAKE THRESH TABLE"
      LOCATE 22, 40
      PRINT "STATUS: ";

      REM Enter commands section
2000  REM ENTER COMMAND
      DISP = 1: REM DISPLAY FROM NOW ON
      GOSUB 9650: REM CLEAR STATUS
      X = 10: Y = 22
      LOCATE Y, X, 1
      PRINT "COMMAND (E=END) >           ";
      LOCATE Y, X + 18
      C$ = INPUT$(1)
2100  IF C$ = "E" OR C$ = "e" THEN END
      IF C$ = "G" OR C$ = "g" THEN 3000
      IF C$ = "A" OR C$ = "a" THEN 3100
      IF C$ = "W" OR C$ = "w" THEN 3200
      IF C$ = "I" OR C$ = "i" THEN 3300
      IF C$ = "C" OR C$ = "c" THEN 3400
      IF C$ = "F" OR C$ = "f" THEN 3500
      IF C$ = "R" OR C$ = "r" THEN 3600
      IF C$ = "S" OR C$ = "s" THEN 3700
      IF C$ = "T" OR C$ = "t" THEN 3800
      IF C$ = "M" OR C$ = "m" THEN CLOSE : CHAIN "TABLE"
      CM$ = ""
      GOTO 2000

3000  REM GAIN
      LOCATE 3, 18
      PRINT SPACE$(5);
      LOCATE 3, 18
      GOSUB 9400: G$ = II$
      IF G$ = CHR$(27) THEN 3020
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
     IF LEN(G$) <> 3 THEN 3000
     BAD = 0
     FOR I = 1 TO 3
     IF INSTR("0123456789", MID$(G$, I, 1)) = 0 THEN BAD = 1
     NEXT I
     IF BAD = 1 THEN PRINT CHR$(7): GOTO 3000
3010 ST$ = "SETTING GAIN..."
     GOSUB 9600: REM DISPLAY STATUS
     GOSUB 9500: REM STOP
     S$ = "<G" + G$
     DISP = 0: GOSUB 9800
     GAIN$ = G$
3020 LOCATE 3, 18
     PRINT GAIN$
     LSET F$ = GAIN$
     PUT 5, 1
     GOTO 2000

3100 REM ATTENUATION
     LOCATE 5, 18
     PRINT SPACE$(5);
     LOCATE 5, 18
     GOSUB 9400: A$ = II$
     IF A$ = CHR$(27) THEN 3120
     IF LEN(A$) <> 3 THEN 3100
     BAD = 0
     FOR I = 1 TO 3
     IF INSTR("0123456789", MID$(A$, I, 1)) = 0 THEN BAD = 1
     NEXT I
     IF BAD = 1 THEN PRINT CHR$(7): GOTO 3100
3110 ST$ = "SETTING ATTENUATION..."
     GOSUB 9600
     GOSUB 9500: REM STOP
     S$ = "<A" + A$
     DISP = 0: GOSUB 9800
     ATTN$ = A$
3120 LOCATE 5, 18
     PRINT ATTN$
     LSET F$ = ATTN$
     PUT 5, 2
     GOTO 2000

3200 REM WIDTH
     LOCATE 7, 18
     PRINT SPACE$(5);
     LOCATE 7, 18
     GOSUB 9400: W$ = II$
     IF W$ = CHR$(27) THEN 3220
     IF LEN(W$) <> 3 THEN 3200
     BAD = 0
     FOR I = 1 TO 3
     IF INSTR("0123456789", MID$(W$, I, 1)) = 0 THEN BAD = 1
     NEXT I
     IF BAD = 1 THEN PRINT CHR$(7): GOTO 3200
3210 ST$ = "SET PULSE WIDTH..."
     GOSUB 9600: REM DISPLAY STATUS
     GOSUB 9500: REM STOP
     S$ = "<W" + W$
     DISP = 0: GOSUB 9800
     WIDTH$ = W$
3220 LOCATE 7, 18
     PRINT WIDTH$
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
     LSET F$ = WIDTH$
     PUT 5, 3
     GOTO 2000

3300 REM INITIALIZE
     ST$ = "INITIALIZING..."
     GOSUB 9600
     GOSUB 9500
     S$ = "<I"
     DISP = 0: GOSUB 9800
     TM = 10: GOSUB 9900
     GOTO 2000

3400 REM CENTER SONAR HEAD
     ST$ = "CENTERING HEAD..."
     GOSUB 9600
     GOSUB 9500
     S$ = "<C"
     DISP = 0: GOSUB 9800
     TM = 10: GOSUB 9900
     GOTO 2000

3500 REM FAST BAUD RATE
     ST$ = "SET FAST BAUD RATE..."
     GOSUB 9600
     GOSUB 9500
     S$ = ":B001"
     DISP = 0: GOSUB 9800
     TM = 4: GOSUB 9900
     S$ = "<F"
     DISP = 0: GOSUB 9800
     TM = 2: GOSUB 9900
     GOTO 2000

3600 REM RUN SONAR RECORDER
     ST$ = "SETTING UP TO RUN..."
     GOSUB 9600: REM DISPLAY STATUS
     GOSUB 9500: REM STOP
     S$ = "<I"
     DISP = 0: GOSUB 9800: REM SEND COMMAND
     S$ = "<R"
     DISP = 0: GOSUB 9800: REM SEND COMMAND
     TM = 4: GOSUB 9900
     GOSUB 9000: REM DO RECORDING
     C$ = CM$
     GOTO 2100

3700 REM STOP
     ST$ = "STOPPING SYSTEM..."
     GOSUB 9600: REM DISPLAY STATUS
     GOSUB 9500: REM STOP SYSTEM
     GOTO 2000

3800 REM THRESHOLD
     LOCATE 10, 18
     PRINT SPACE$(5);
     LOCATE 10, 18
     LINE INPUT TFN$
     IF TFN$ = "" THEN 3820
3810 ST$ = "INSTALL THRESHOLD..."
     GOSUB 9600: REM DISPLAY STATUS
     GOSUB 9500: REM STOP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
3820 LOCATE 10, 18
     PRINT TFN$
     CLOSE 6
     OPEN "R", 6, TFN$, 1
     FIELD 6, 1 AS FTH$
     GOSUB 9500: REM STOP SYSTEM
     S$ = "<P": REM POINT TO START OF THRESHOLD AREA
     DISP = 0: GOSUB 9800: REM SENDIT
     TM = 4: GOSUB 9900: REM TIMER
     CNT = 0: LNE = 0
     S$ = "<T"
     LOCATE 3, 40
     BINS = 0
     PRINT "Bytes Installed:                  ";
     LOCATE 3, 57, 0
     PRINT USING "###"; 0
     LF = LOF(6)
     FOR RP = 1 TO LF
     GET 6, RP
     IF INSTR("0123456789ABCDEF", FTH$) = 0 THEN 3840
     S$ = S$ + FTH$
     CNT = CNT + 1
     IF CNT < 20 THEN 3840
     CNT = 0
     BINS = BINS + 10
     LOCATE 3, 57, 0
     PRINT USING "###"; BINS;
     DISP = 0: GOSUB 9800: REM SENDIT
     TM = 2: GOSUB 9900
     S$ = "<T"
3840 NEXT RP
     GOSUB 9650
     LOCATE 3, 40
     PRINT SPACE$(38);
     LOCATE 3, 40
     PRINT "Threshold = "; TFN$;
     CLOSE 6
     GOTO 2000

9000 REM DO RECORDING
     ST$ = "OPENING RECORDER FILES..."
     GOSUB 9600: REM DISPLAY STATUS
     CM$ = ""
     CLOSE 1: CLOSE 2: CLOSE 3: CLOSE 4
     OPEN "COM1:9600,N,8,1" FOR RANDOM AS #1
     OPEN "COM2:9600,N,8,1" FOR RANDOM AS #2
     FILE = FILE + 1
     FILE$ = STR$(FILE)
     FILE$ = MID$(FILE$, 2, 10)
     LSET F$ = STR$(FILE)
     PUT 5, 6
     OPEN "O", 3, "SN1_" + FILE$ + ".SON"
     OPEN "O", 4, "SN2_" + FILE$ + ".SON"
     ST$ = "RECORDING DATA...     "
     GOSUB 9600: REM DISPLAY STATUS
     LF$ = CHR$(10)
9010 CM$ = INKEY$
     IF CM$ <> "" THEN 9030
     IF EOF(1) THEN 9020
     N1$ = INPUT$(1, #1)
     PRINT #3, N1$;
     IF N1$ <> LF$ THEN 9020
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
     PRINT #3, "[" + TIME$ + "]"
9020 CM$ = INKEY$
     IF CM$ <> "" THEN 9030
     IF EOF(2) THEN 9010
     N2$ = INPUT$(1, #2)
     PRINT #4, N2$;
     IF N2$ <> LF$ THEN 9010
     PRINT #4, "[" + TIME$ + "]"
     GOTO 9010
9030 CLOSE 1
     CLOSE 2
     CLOSE 3
     CLOSE 4
     GOSUB 9650: REM REMOVE DISPLAY FROM STATUS
     RETURN

9400 REM LINE INPUT ROUTINE
     CCNT = 0
     II$ = ""
9410 I$ = INPUT$(1)
     IF ASC(I$) = 27 THEN RETURN
     IF ASC(I$) = 13 THEN RETURN
     IF ASC(I$) <> 8 THEN 9420
     IF CCNT = 0 THEN 9420
     LOCATE CSRLIN, POS(0) - 1
     PRINT " ";
     LOCATE CSRLIN, POS(0) - 1
     CCNT = CCNT - 1
     J = LEN(II$)
     II$ = LEFT$(II$, J - 1)
     GOTO 9410
9420 IF INSTR(" 0123456789", I$) = 0 THEN 9410
     II$ = II$ + I$
     CCNT = CCNT + 1
     PRINT I$;
     GOTO 9410


9500 REM STOP
     S$ = "<S"
     DISP = 0: GOSUB 9800
     TM = 6: GOSUB 9900
     RETURN


9600 REM DISPLAY STATUS
     ST$ = ST$ + SPACE$(40)
     ST$ = LEFT$(ST$, 31)
     LOCATE 22, 48, 0
     PRINT ST$;
     RETURN

9650 REM REMOVE DISPLAY FROM STATUS
     LOCATE 22, 48
     PRINT SPACE$(31);
     LOCATE 22, 48
     RETURN

9700 REM DRAW HORIZONTAL LINE
     LOCATE Y, X
     PRINT CHR$(199);
     PRINT STRING$(78, 196);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
     PRINT CHR$(182);
     RETURN

9800 REM SEND IT
     CLOSE 1: CLOSE 2: CLOSE 3: CLOSE 4
     IF DISP = 0 THEN 9810
     ST$ = S$
     DISP = 1: GOSUB 9600
9810 OPEN "COM1:9600,N,8,1" FOR RANDOM AS #1
     L = LEN(S$)
     FOR I = 1 TO L
     PRINT #1, MID$(S$, I, 1);
     NEXT I
     PRINT #1, CHR$(13);
9820 TM = 1: GOSUB 9900
     IF DISP = 1 THEN GOSUB 9650
     CLOSE 1
9830 RETURN

9900 REM TIMER (TM=1 FOR 1/2 SECOND DELAY)
     FOR T = 1 TO TM
     FOR ZZ = 1 TO 1237
     NEXT ZZ
     NEXT T
     RETURN
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program LOGIT.BAS (LOGIT.EXE)
     REM Does logging of sonar data and displays to screen.

     CLS
     PRINT "OPENING RECORDER FILES...  "
     OPEN "COM1:9600,N,8,1" FOR RANDOM AS #1
     OPEN "COM2:9600,N,8,1" FOR RANDOM AS #2
     HD$ = "1": REM HEAD DOWN POSITION

     OPEN "R", 5, "FILENO.DAT", 10
     FIELD 5, 10 AS F$
     IF LOF(5) = 0 THEN LSET F$ = STR$(1): PUT 5, 1
     GET 5, 1
     FILE = VAL(F$)
     FILE$ = STR$(FILE)
     LG = LEN(FILE$)
     FILE$ = MID$(FILE$, 2, LG - 1)
     LSET F$ = STR$(FILE + 1)
     PUT 5, 1
     CLOSE 5
     OPEN "O", 3, "SN1_" + FILE$ + ".TXT"
     OPEN "O", 4, "SN2_" + FILE$ + ".TXT"
     LOCATE 22, 48
     PRINT "RECORDING...";

     CLS
     GOSUB 9280: REM PRINT LINE

     REM Recording loop
     CR$ = CHR$(13)
9010 CM$ = INKEY$
     IF CM$ = "" THEN 9015
     IF CM$ = "C" OR CM$ = "c" THEN GOSUB 9280
     IF CM$ = "S" OR CM$ = "s" THEN CLOSE : CHAIN "RECORD"
9015 IF EOF(1) THEN 9020
     N1$ = INPUT$(1, #1)
     CMS$ = CMS$ + N1$
     PRINT #3, N1$;
     IF N1$ <> CR$ THEN 9020
     GOSUB 9100: REM DISPLAY
     CMS$ = ""
     PRINT #3, "[TIME]" + TIME$ : REM Time Stamp
9020 CM$ = INKEY$
     IF CM$ = "" THEN 9025
     IF CM$ = "C" OR CM$ = "c" THEN GOSUB 9280
     IF CM$ = "S" OR CM$ = "s" THEN CLOSE : CHAIN "RECORD"
9025 IF EOF(2) THEN 9010
     N2$ = INPUT$(1, #2)
     PRINT #4, N2$;
     IF N2$ <> CR$ THEN 9010
     PRINT #4, "[TIME]" + TIME$ : REM Time Stamp
     GOTO 9010
9030 LOCATE 22, 48
     PRINT SPACE$(51);
     LOCATE 22, 48
     RETURN

9100 REM DISPLAY
     PSN = INSTR(CMS$, ">E") : REM Find starting location
     IF PSN = 0 THEN RETURN : REM If no starting location exit
     IF MID$(CMS$, PSN + 2, 1) <> "1" THEN RETURN: REM Check For TXP #
     HD$ = MID$(CMS$, PSN + 3, 1)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          IF HD$ = "1" THEN RETURN
          PRINT CHR$(7); : REM Bell Character
          REM Locate event angle
          ANGL = VAL(MID$(CMS$, PSN + 4, 1)) + 1

          REM Range information
          LO$ = MID$(CMS$, PSN + 7, 1)
          MD$ = MID$(CMS$, PSN + 6, 1)
          HI$ = MID$(CMS$, PSN + 5, 1)
          LO = INSTR("0123456789ABCDEF", LO$) - 1
          MD = INSTR("0123456789ABCDEF", MD$) - 1
          HI = INSTR("0123456789ABCDEF", HI$) - 1
          RANGE = HI * 256 + MD * 16 + LO
          RNG = FIX(RANGE / 6) + 1 : REM Scale for screen display
          IF RNG > 24 THEN RNG = 24
          Y = 25 - RNG
          ON ANGL GOTO 9150, 9175, 9200, 9225, 9250, 9275 : REM Ray location

     9150 REM 20 DEGREES LEFT
          X = 26 + FIX((Y - 1) / 2)
          LOCATE Y, X, 0
          PRINT "0";
          RETURN

     9175 REM 12 DEGREES LEFT
          X = 33 + FIX((Y - 1) / 4)
          LOCATE Y, X, 0
          PRINT "1";
          RETURN

     9200 REM 4 DEGREES LEFT
          X = 38
          IF Y > 12 THEN X = 39
          LOCATE Y, X, 0
          PRINT "2";
          RETURN

     9225 REM 4 DEGREES RIGHT
          X = 42
          IF Y > 12 THEN X = 41
          LOCATE Y, X, 0
          PRINT "3";
          RETURN

     9250 REM 12 DEGREES RIGHT
          X = 47 - FIX((Y - 1) / 4)
          LOCATE Y, X, 0
          PRINT "4";
          RETURN

     9275 REM 20 DEGREES RIGHT
          X = 54 - FIX((Y - 1) / 2)
          LOCATE Y, X, 0
          PRINT "5";
          RETURN

     9280 REM Set up screen
          CLS
          LOCATE 1, 1
          PRINT "C=Clear Screen"
          PRINT "S=Stop-GOTO RECORD MENU"
          REM PRINT '|' LINE
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
FOR Y = 1 TO 24
LOCATE Y, 40
PRINT "|";
NEXT Y
LOCATE 25, 20
PRINT STRING$(41, "=");
VALU = 0
FOR Y = 24 TO 1 STEP -1
LOCATE Y, 60
PRINT USING "###"; VALU;
PRINT "-";
PRINT USING "###"; VALU + 5;
VALU = VALU + 6
NEXT Y
RETURN
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1000 REM Program TABLE.BAS (TABLE.EXE)
     REM Program to create threshold tables under a file name

     CLS
     CLOSE
     DIM VLU(300) : REM One value for each of 300 feet

     REM Screen header
     PRINT "P=Print Report > ";
     P$ = INPUT$(1)
     IF P$ = "P" OR P$ = "p" THEN CLS : PRINT "Print Option: ON": GOTO
1003
     PRINT "Print Option: OFF"
1003 IF P$ = "p" THEN P$ = "P"
     PRINT "E=Expo, L=Linear, R=Recorder, Q=Quit > ";
     CM$ = INPUT$(1)
     IF CM$ = "E" OR CM$ = "e" THEN 1005
     IF CM$ = "L" OR CM$ = "l" THEN 2000
     IF CM$ = "Q" OR CM$ = "q" THEN END
     IF CM$ = "R" OR CM$ = "r" THEN CLOSE : CHAIN "RECORD"
     GOTO 1000
1005 REM Exponential method of threshold generation
     CLS
     PRINT "******* EXPONENTIAL ********"
     PRINT
     E = 2.7183
     INPUT "ENTER INITIAL > "; IT : REM Initial value of curve
     INPUT "ENTER FINAL   > "; FL : REM Final value of curve
     INPUT "ENTER TIME CT > "; T : REM Time constant
     INPUT "ENTER FT DELAY> "; DL : REM Ft of $FF (255) values at start
     INPUT "ENTER FILE NM > "; FILE$ : REM Name of file
     PRINT
1010 OPEN "O", 1, FILE$

     REM Calculate constants
     CNTS = 300 - DL
     FVAL = FL
     FOR I = 1 TO 10
     RANGE = IT - FL
     FV = RANGE * E ^ (-CNTS / T) + FL
     REM PRINT FV
     FL = FL - (FV - FVAL)
     NEXT I

     REM Print Threshold values to file and screen/printer
     FOR I = 1 TO DL
     PRINT "FF ";
     IF P$ = "P" THEN LPRINT "FF ";
     PRINT #1, "FF ";
     CNT = CNT + 1
     NEXT I
     FOR I = 1 TO (300 - DL)
     V = RANGE * E ^ (-I / T) + FL
     V = FIX(V + .5)
     H = FIX(V / 16)
     L = V - H * 16
     H$ = MID$("0123456789ABCDEF", H + 1, 1)
     L$ = MID$("0123456789ABCDEF", L + 1, 1)
     PRINT H$; L$; " ";
     IF P$ = "P" THEN LPRINT H$; L$; " ";
     PRINT #1, H$; L$; " ";
     CNT = CNT + 1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

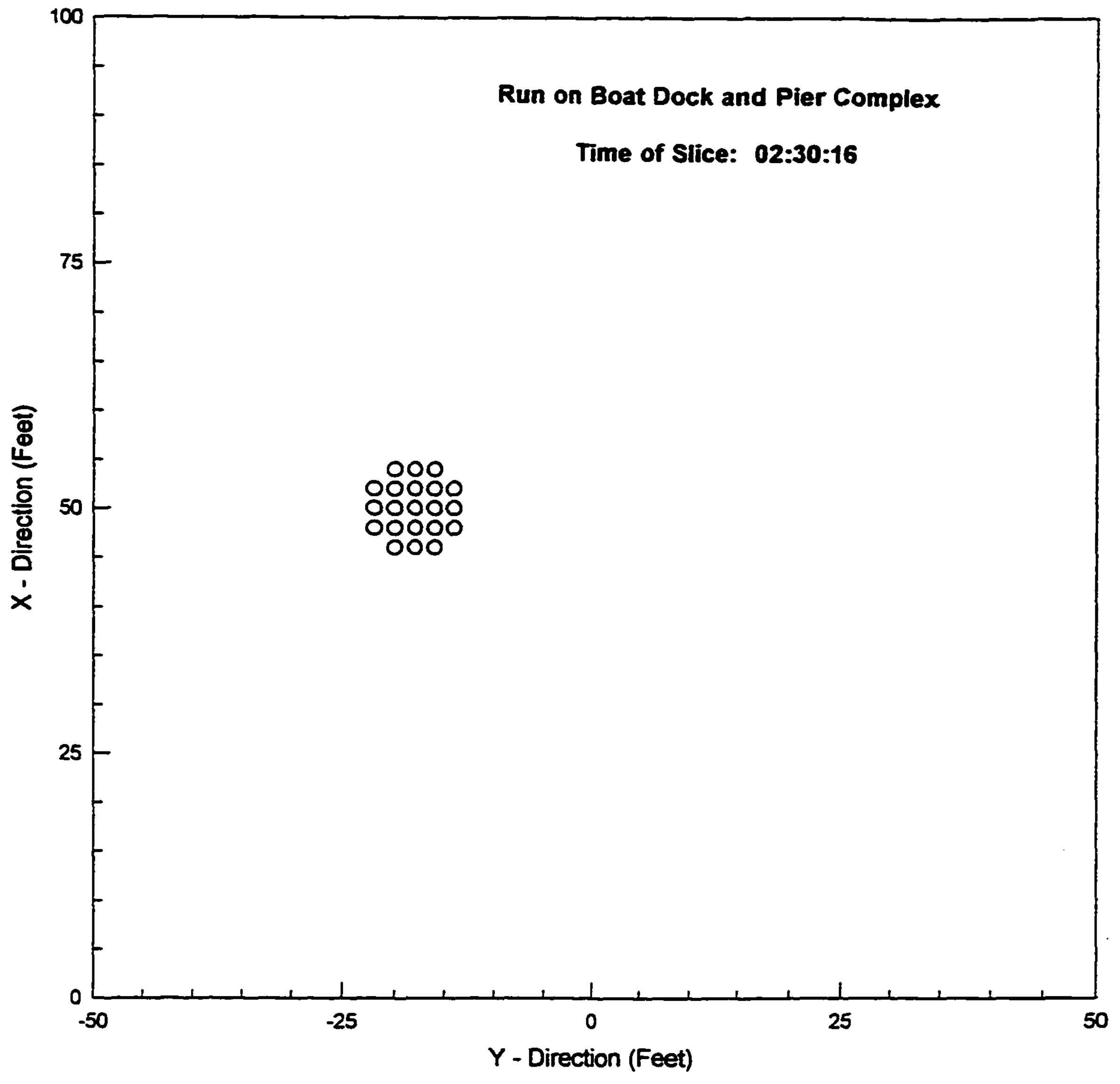
```
     IF CNT = 20 THEN PRINT : PRINT #1, "": GOSUB 1500: CNT = 0
     NEXT I
     PRINT
     PRINT #1, ""
     IF P$ = "P" THEN LPRINT
     PRINT "Hit Any Key > ";
     CM$ = INPUT$(1)
     GOTO 1000
1500 IF P$ = "P" THEN LPRINT
     RETURN

2000 REM Linear Table Generation
     CLS
     PRINT "******* LINEAR TABLE ********"
     PRINT
     INPUT "ENTER FILE NAME > "; FILE$
     OPEN "O", 1, FILE$
     PRINT
     PT = 1: CV = 255
2010 INPUT "ENTER (FEET),(ENDING VALUE) > "; FT, VL
     IF FT > 300 THEN FT = 300
     DELTA = (VL - CV) / (FT - PT)
     FOR I = PT TO FT - 1
     VLU(I) = FIX(CV + .5)
     CV = CV + DELTA
     NEXT I
     VLU(FT) = FIX(CV + .5)
     PT = FT
     IF FT >= 300 THEN 2100
     GOTO 2010

2100 REM Print values and log
     PRINT
     CNT = 0
     FOR I = 1 TO 300
     H = FIX(VLU(I) / 16)
     L = VLU(I) - H * 16
     H$ = MID$("0123456789ABCDEF", H + 1, 1)
     L$ = MID$("0123456789ABCDEF", L + 1, 1)
     PRINT H$; L$; " ";
     PRINT #1, H$; L$; " ";
     CNT = CNT + 1
     IF CNT = 20 THEN PRINT : PRINT #1, "": CNT = 0
     NEXT I
     PRINT
     PRINT "Hit Any Key > ";
     CM$ = INPUT$(1)
     GOTO 1000
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

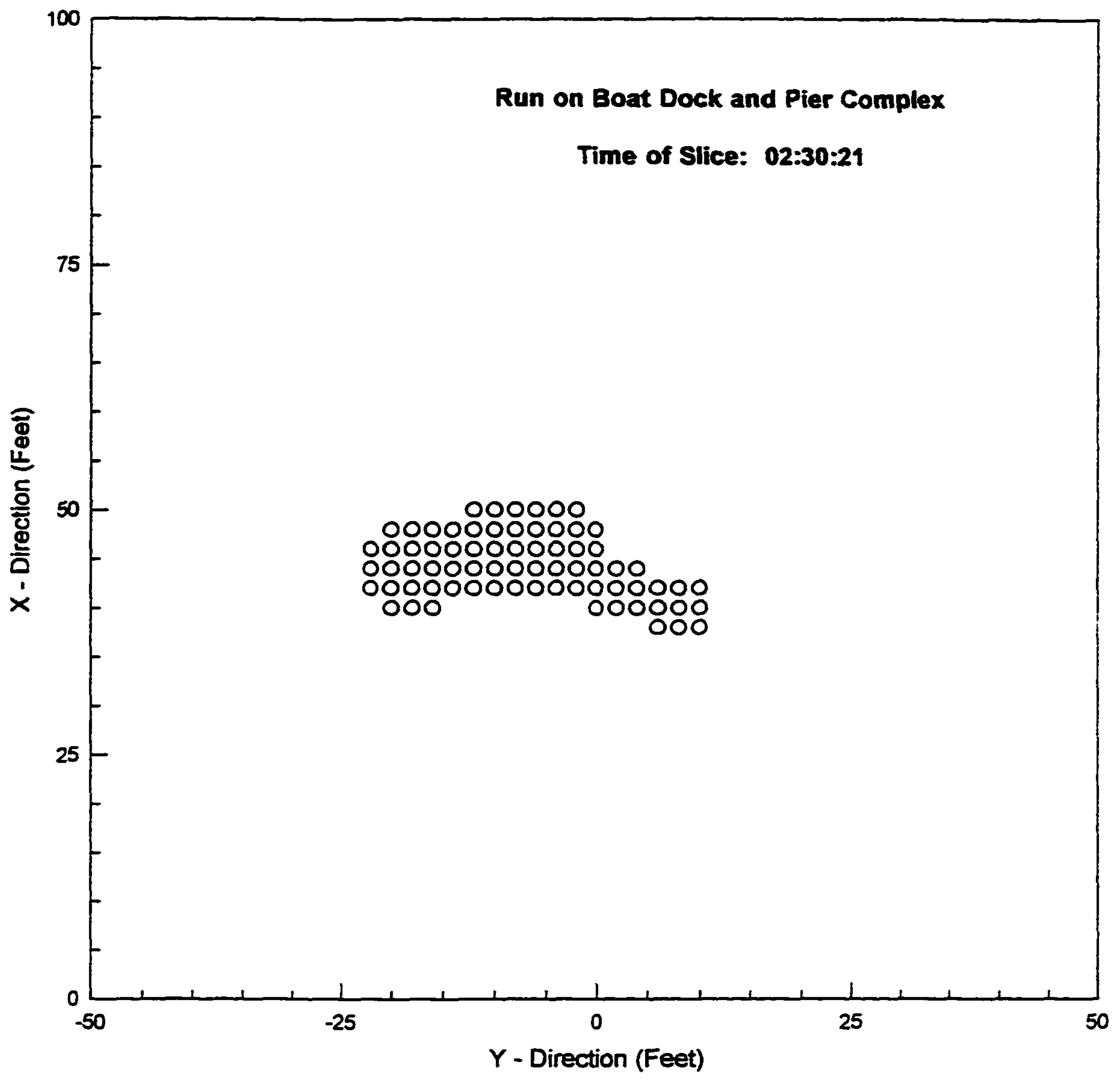
APPENDIX D

DATA PLOTS

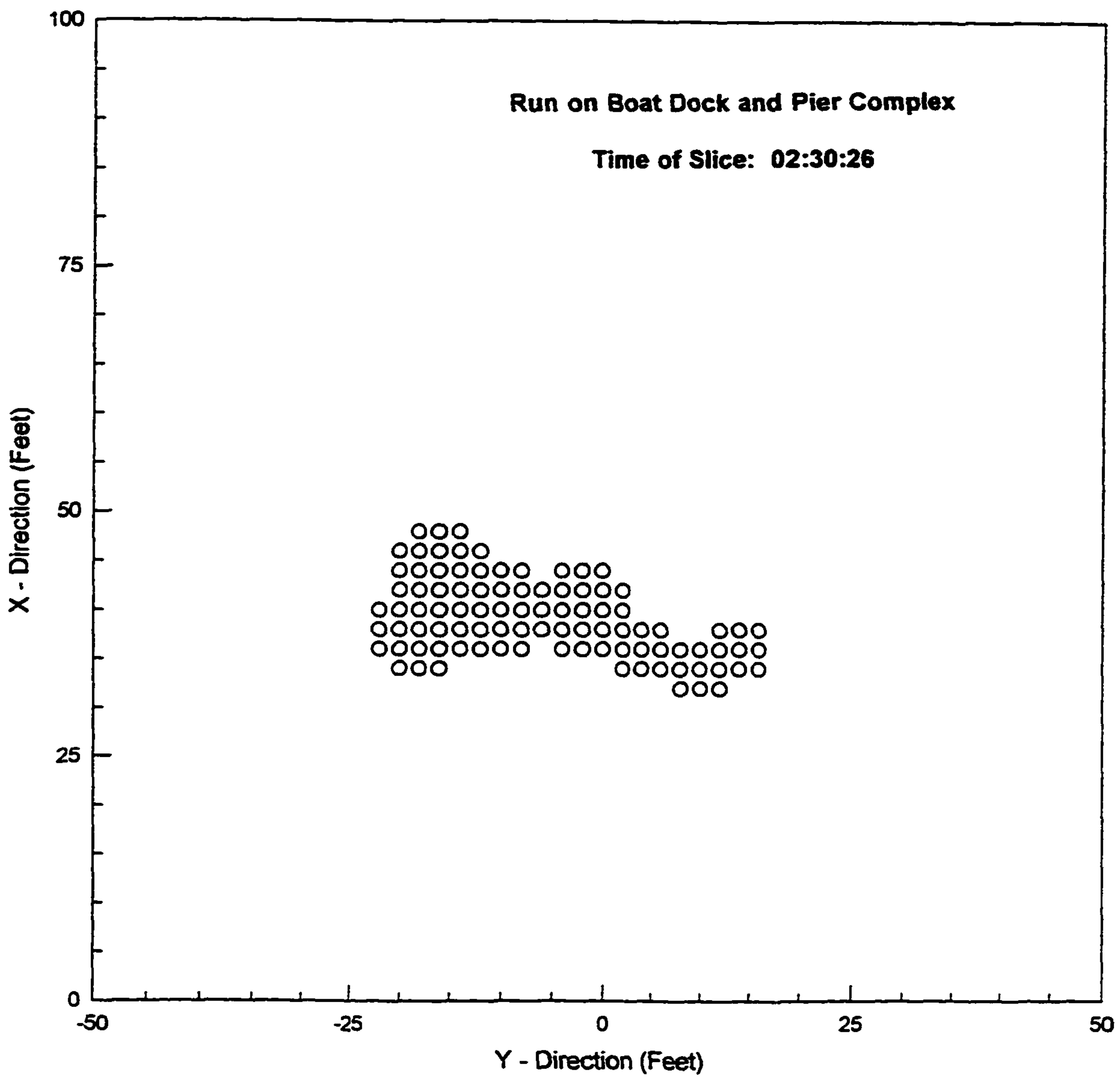
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



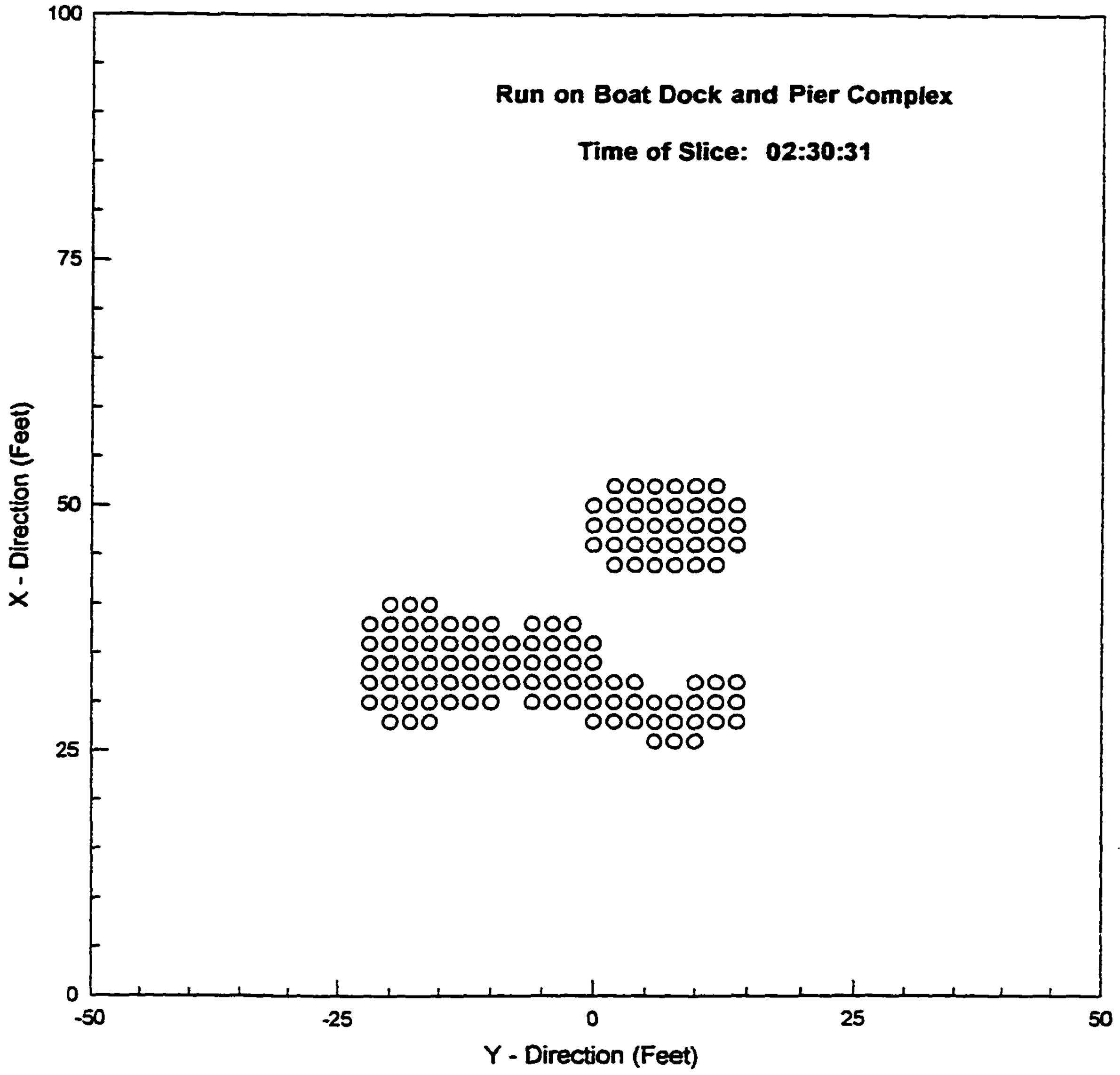
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



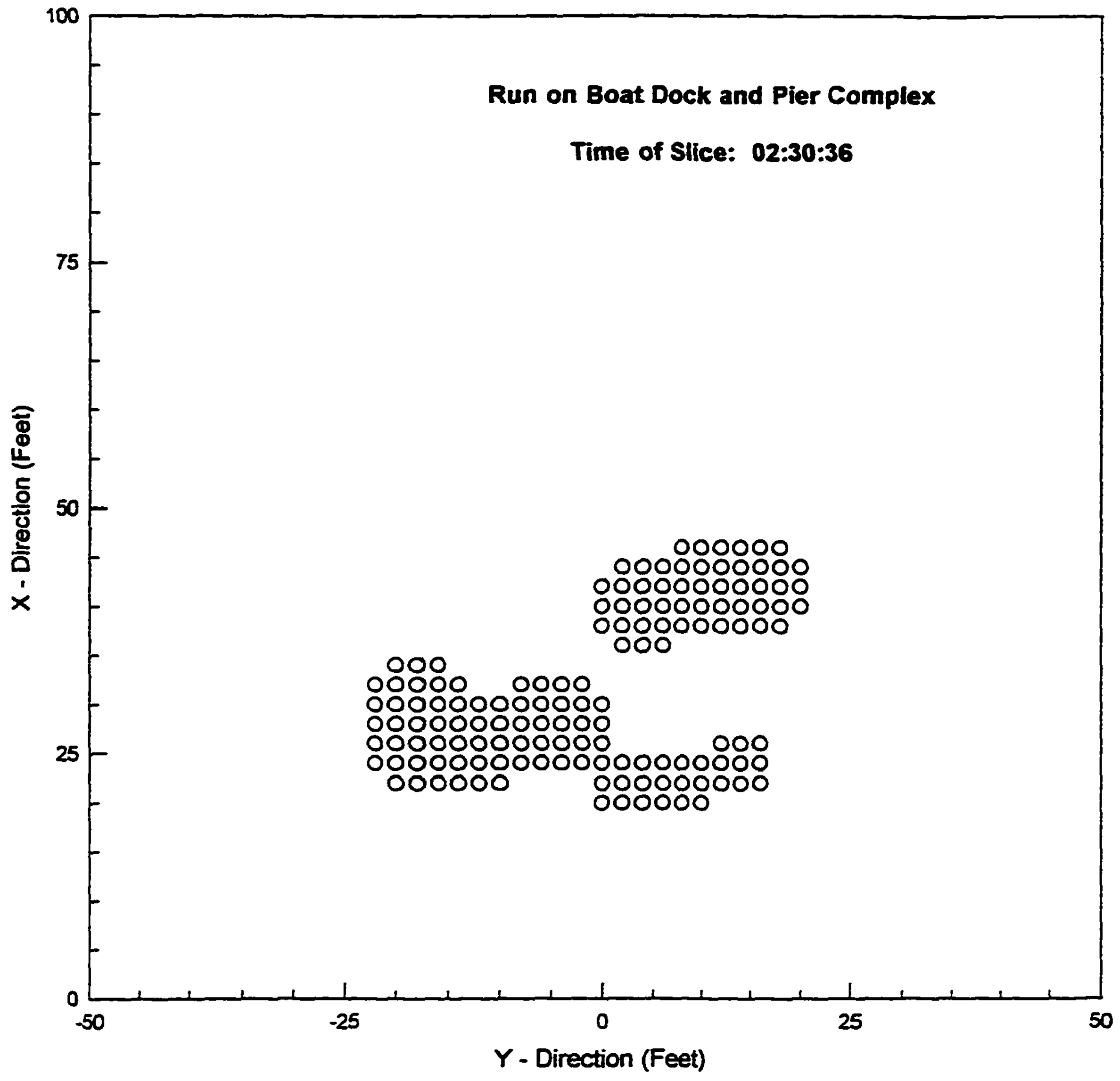
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



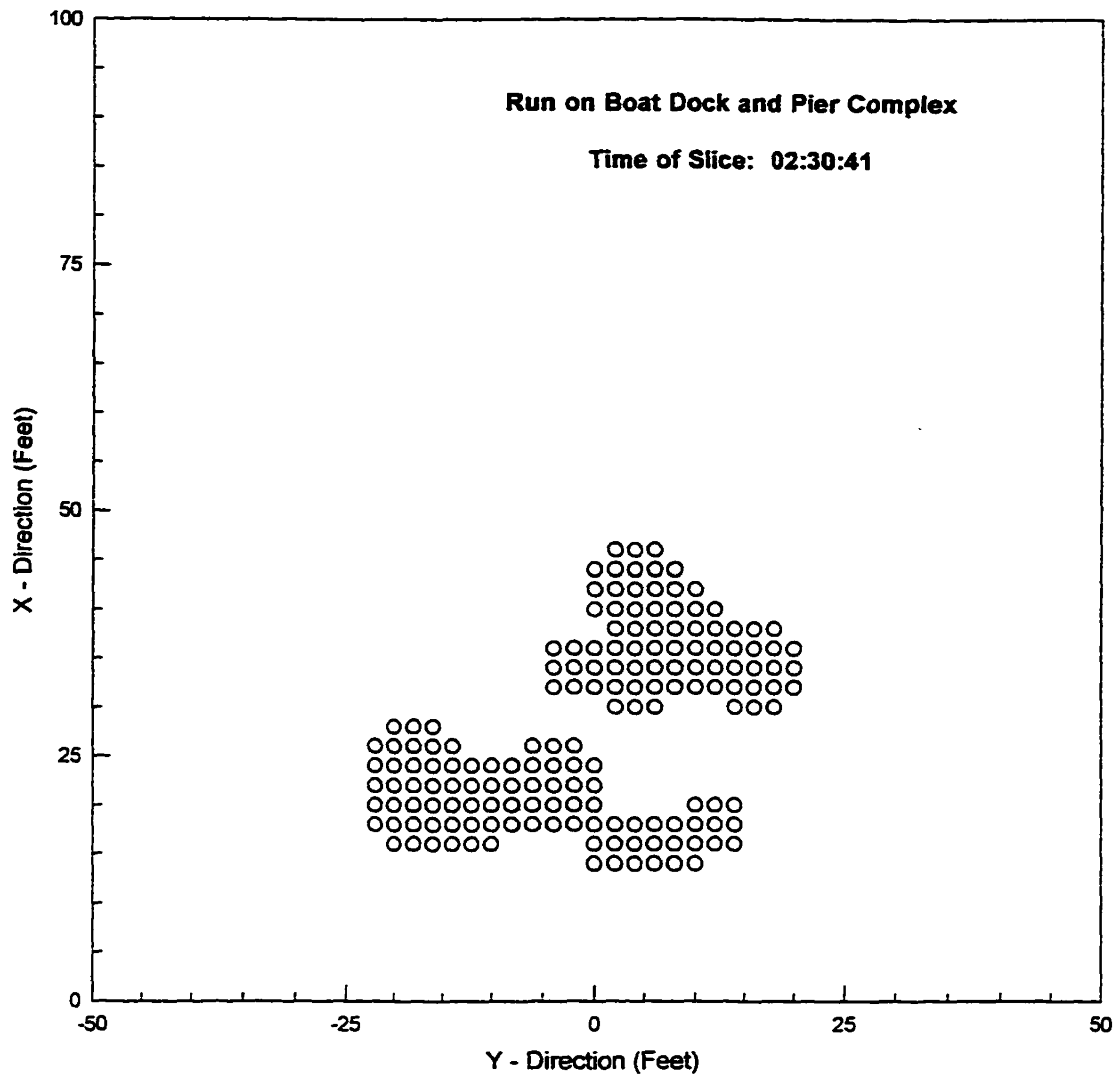
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



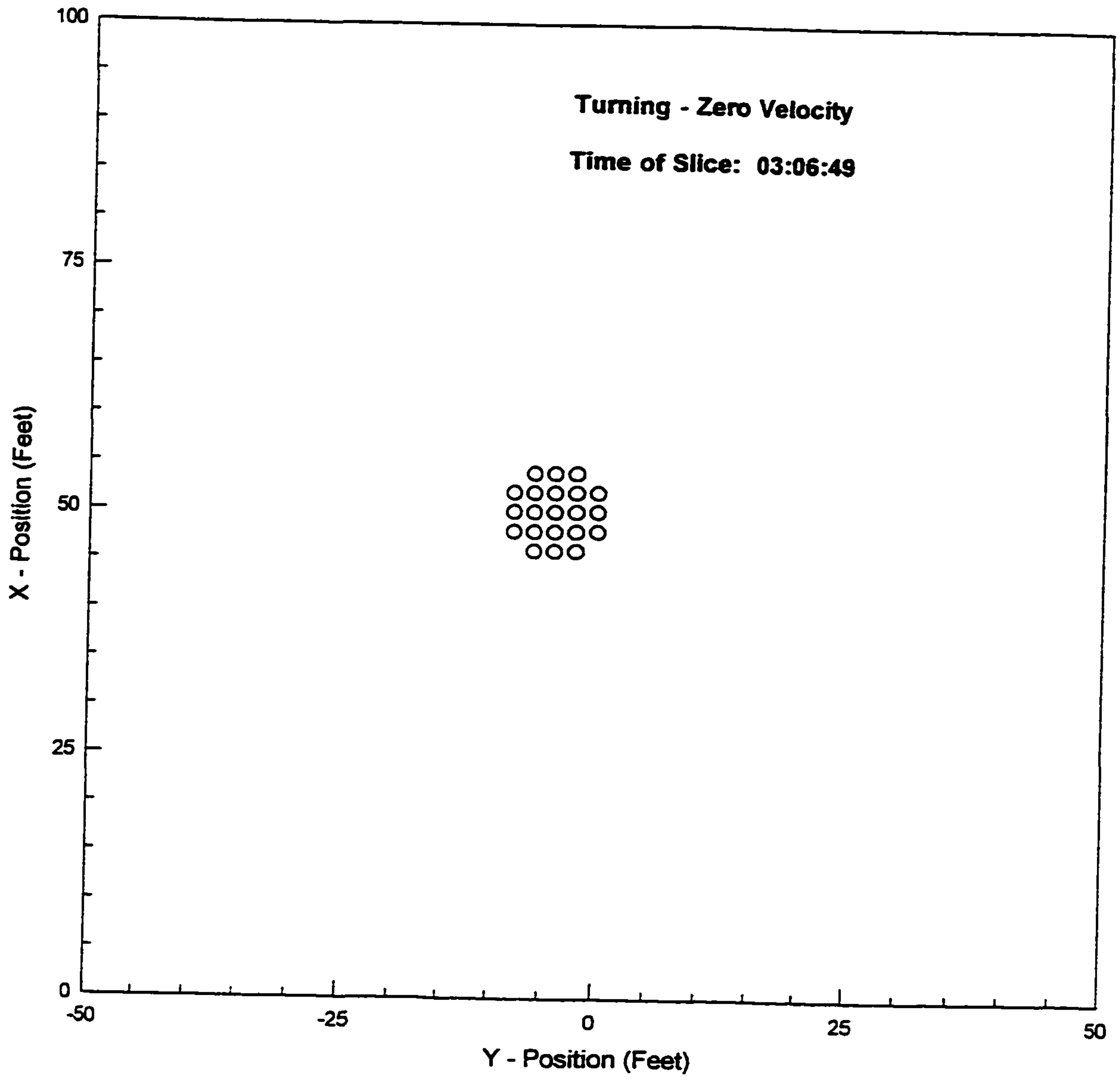
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



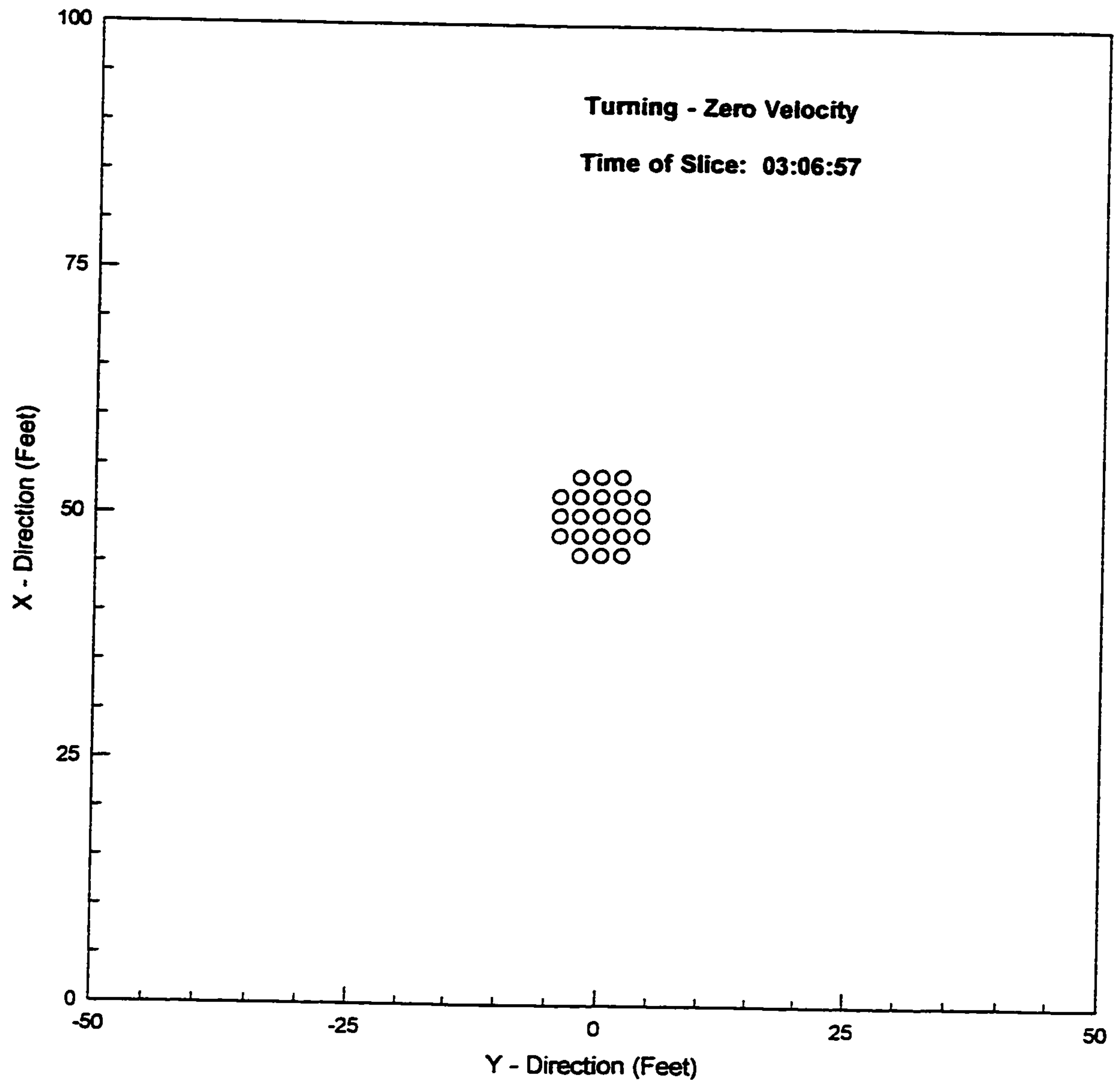
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



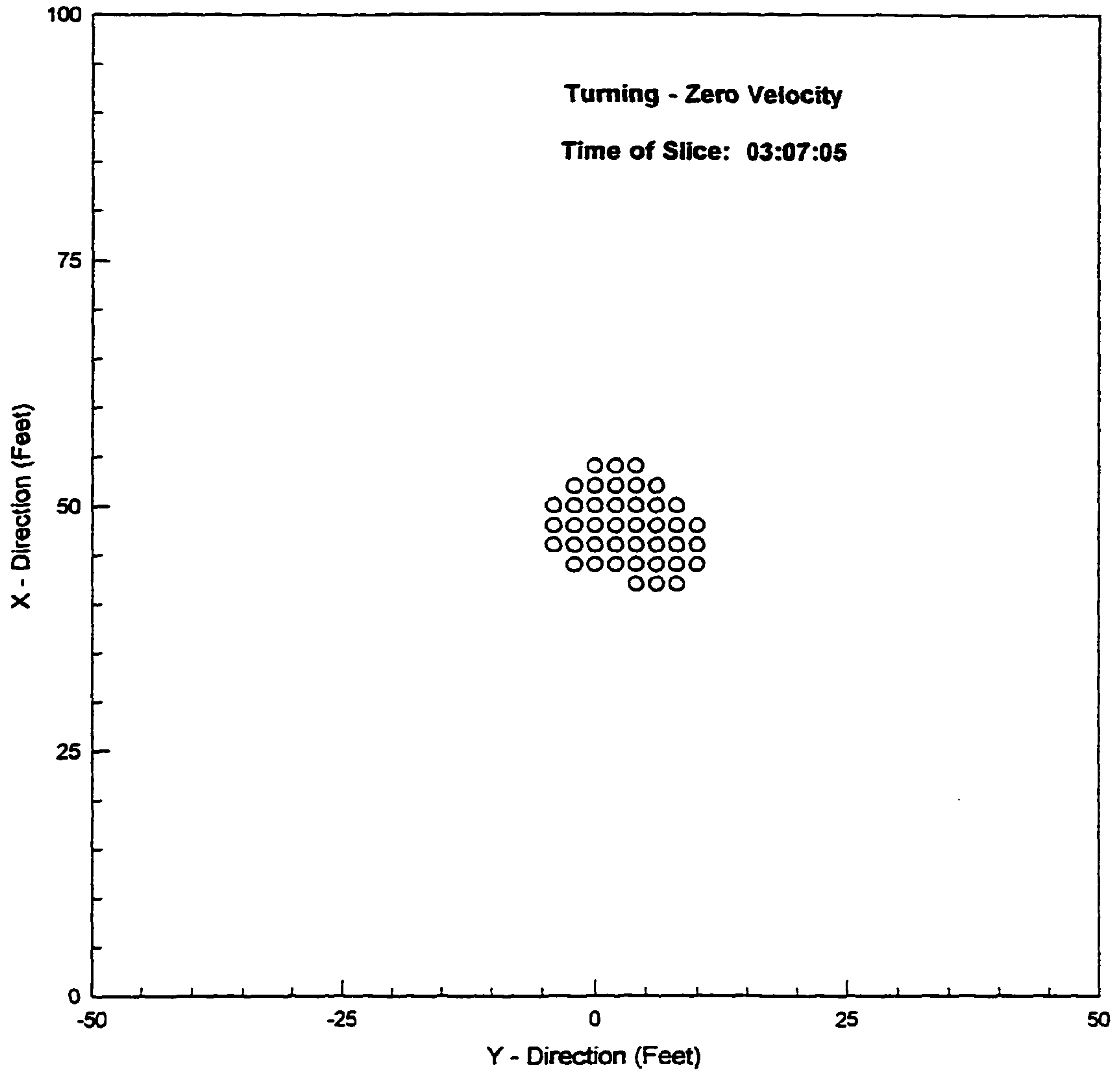
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



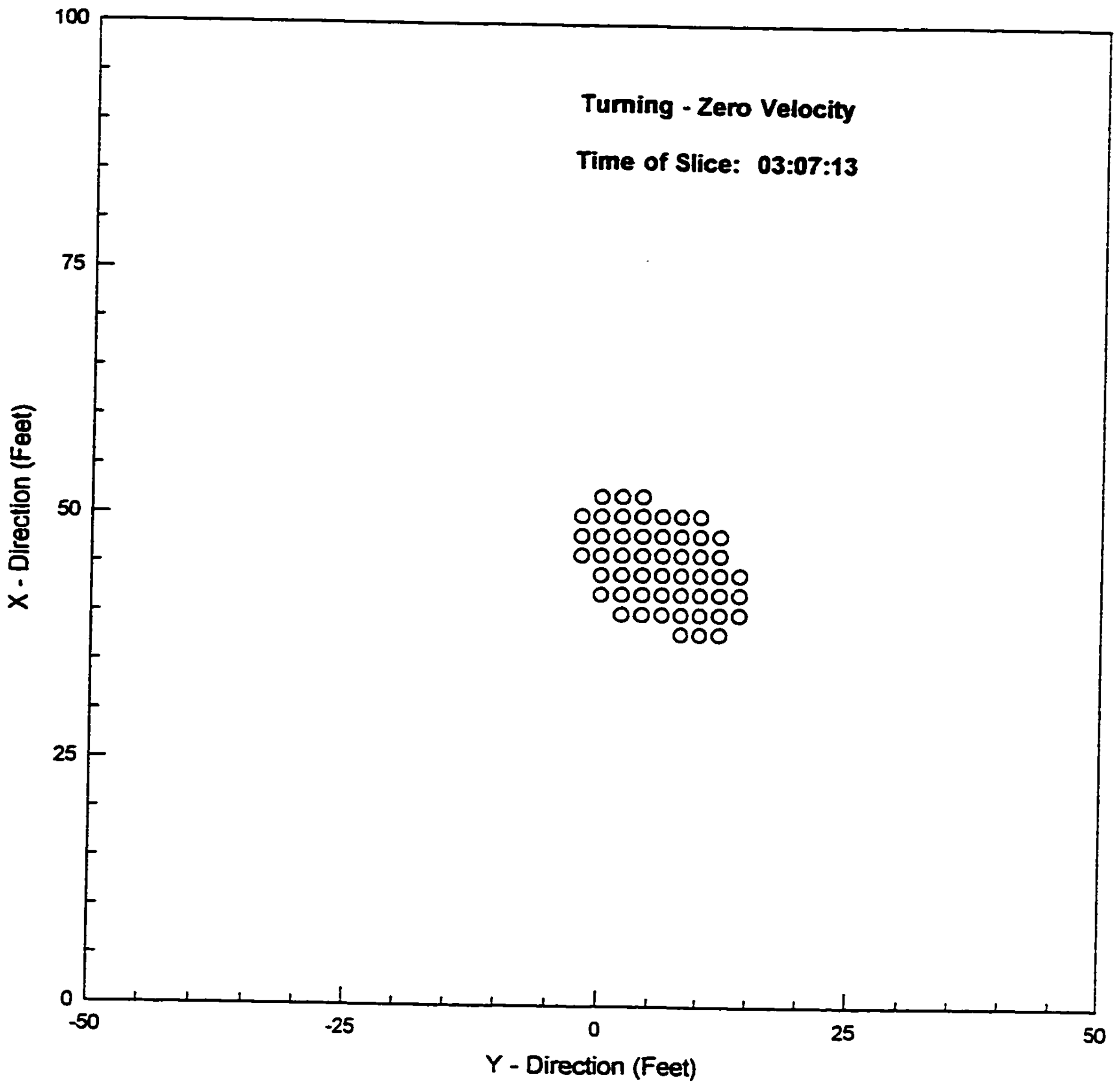
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



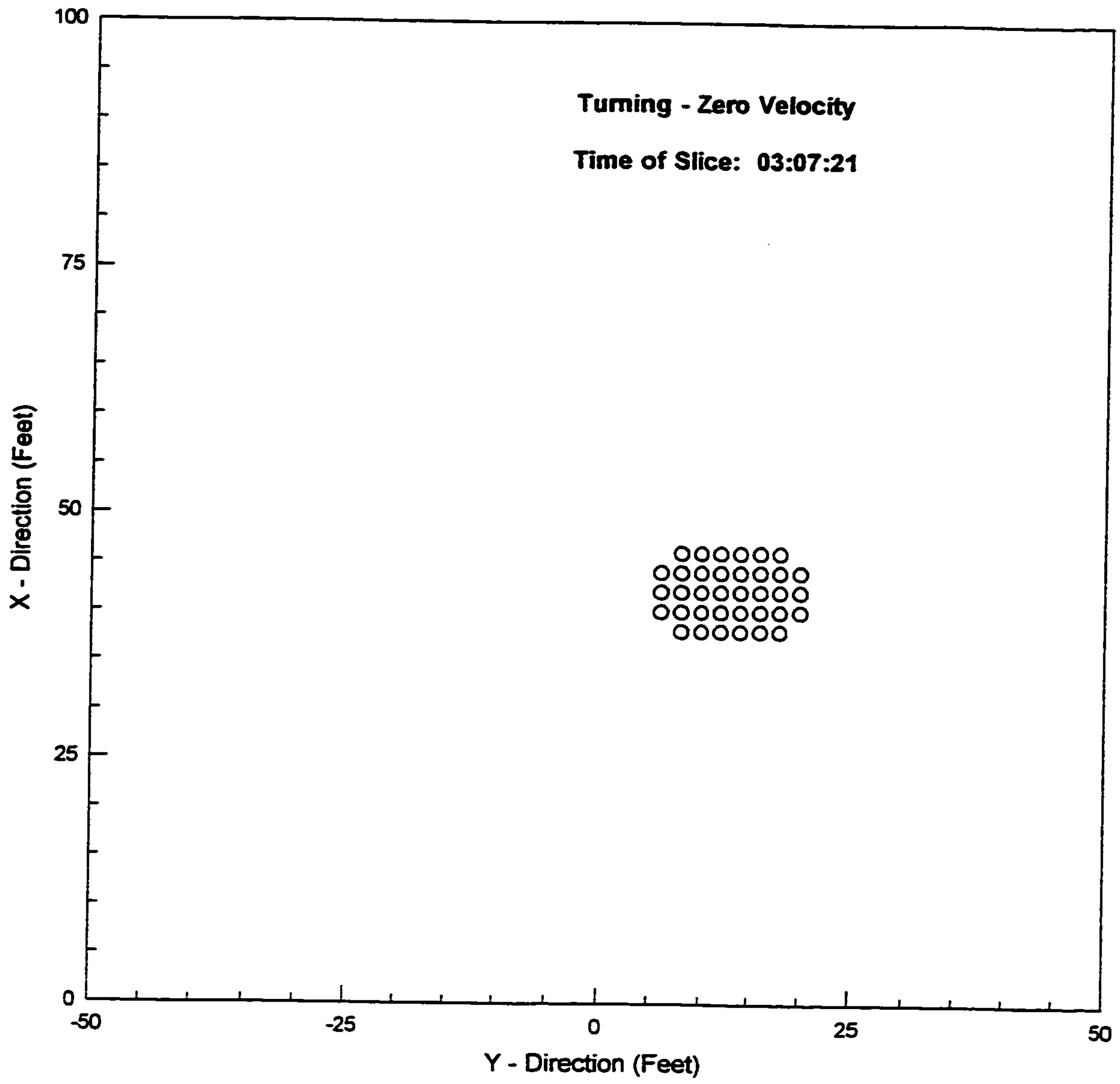
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



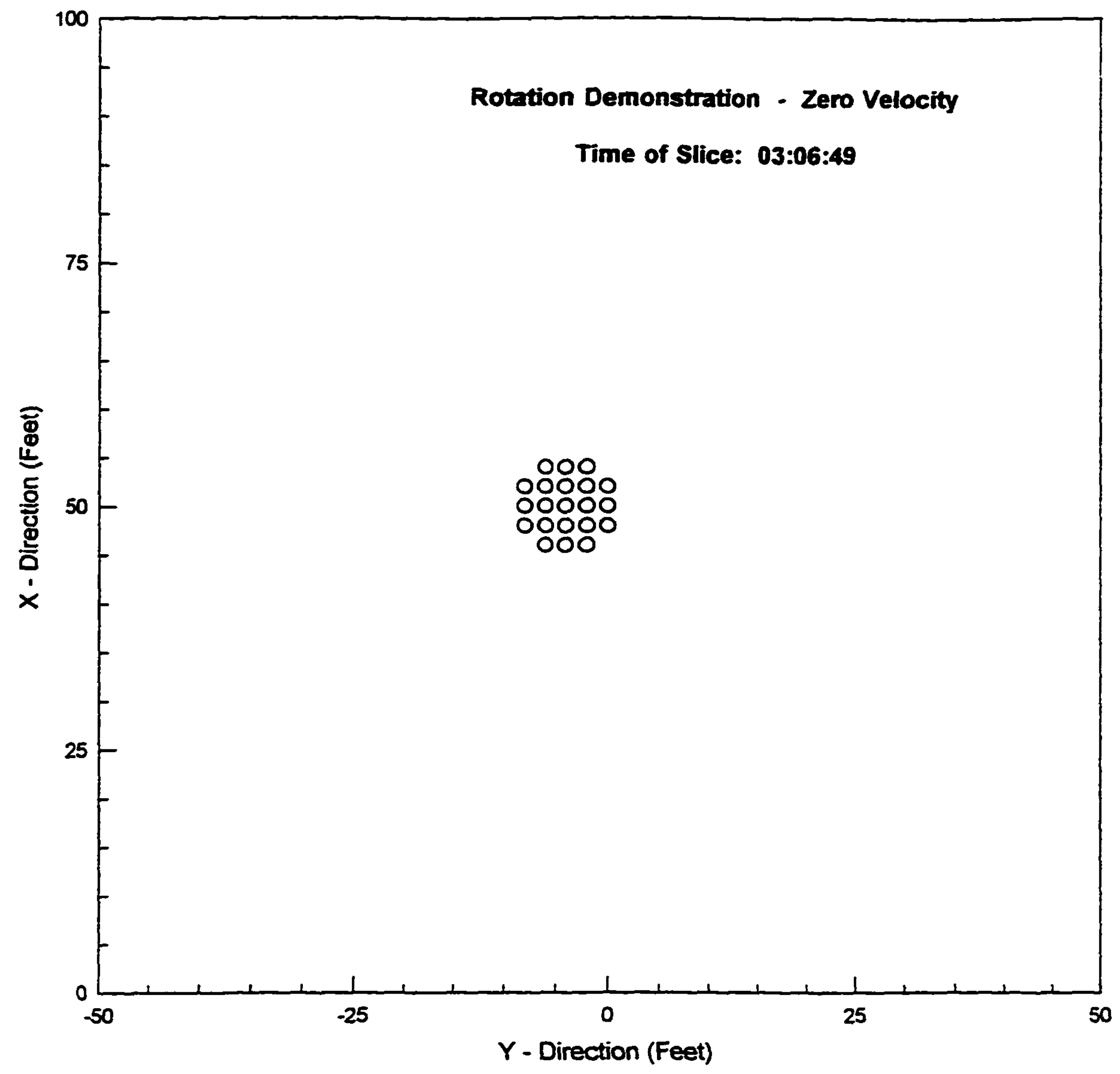
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



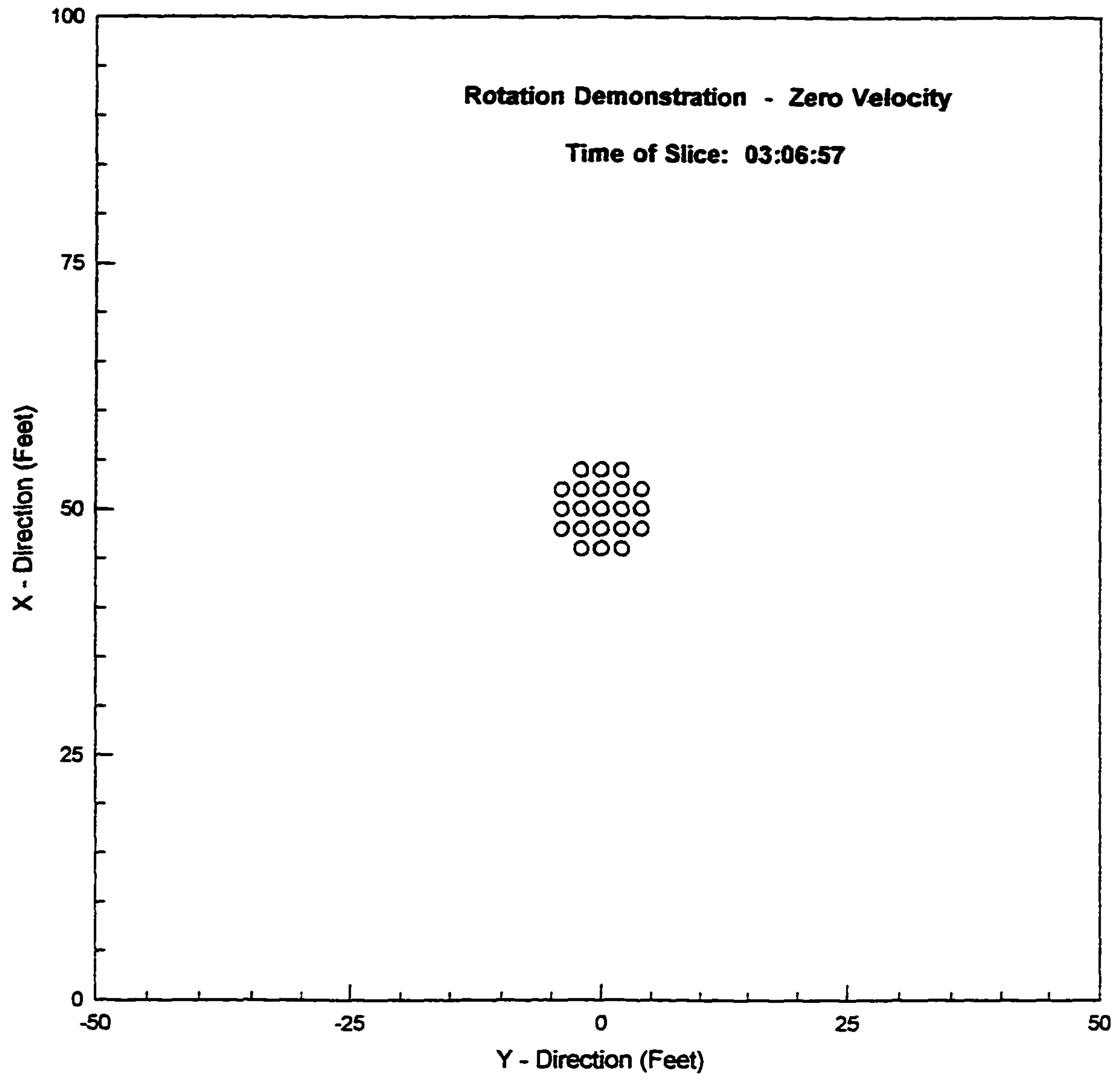
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



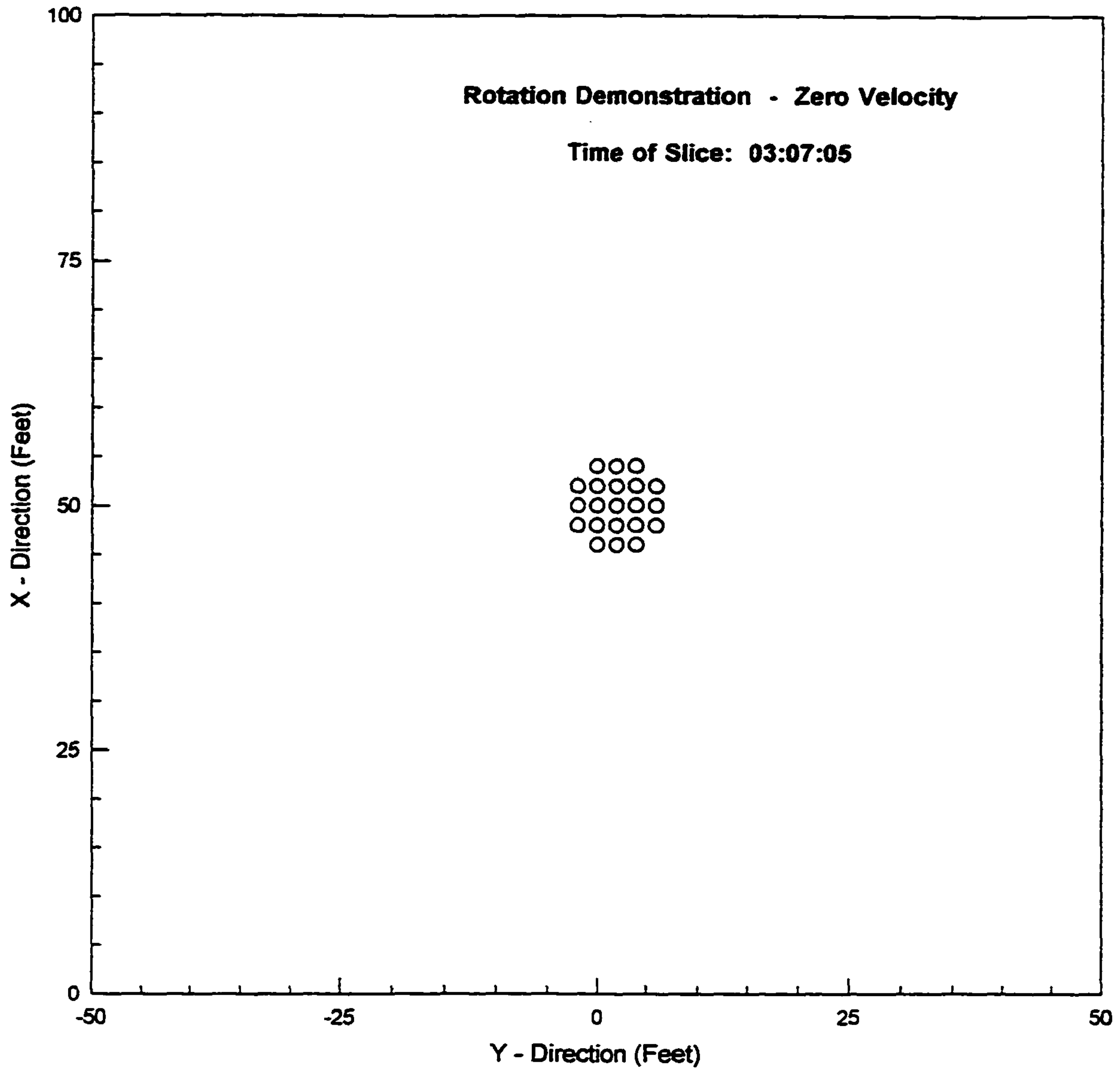
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



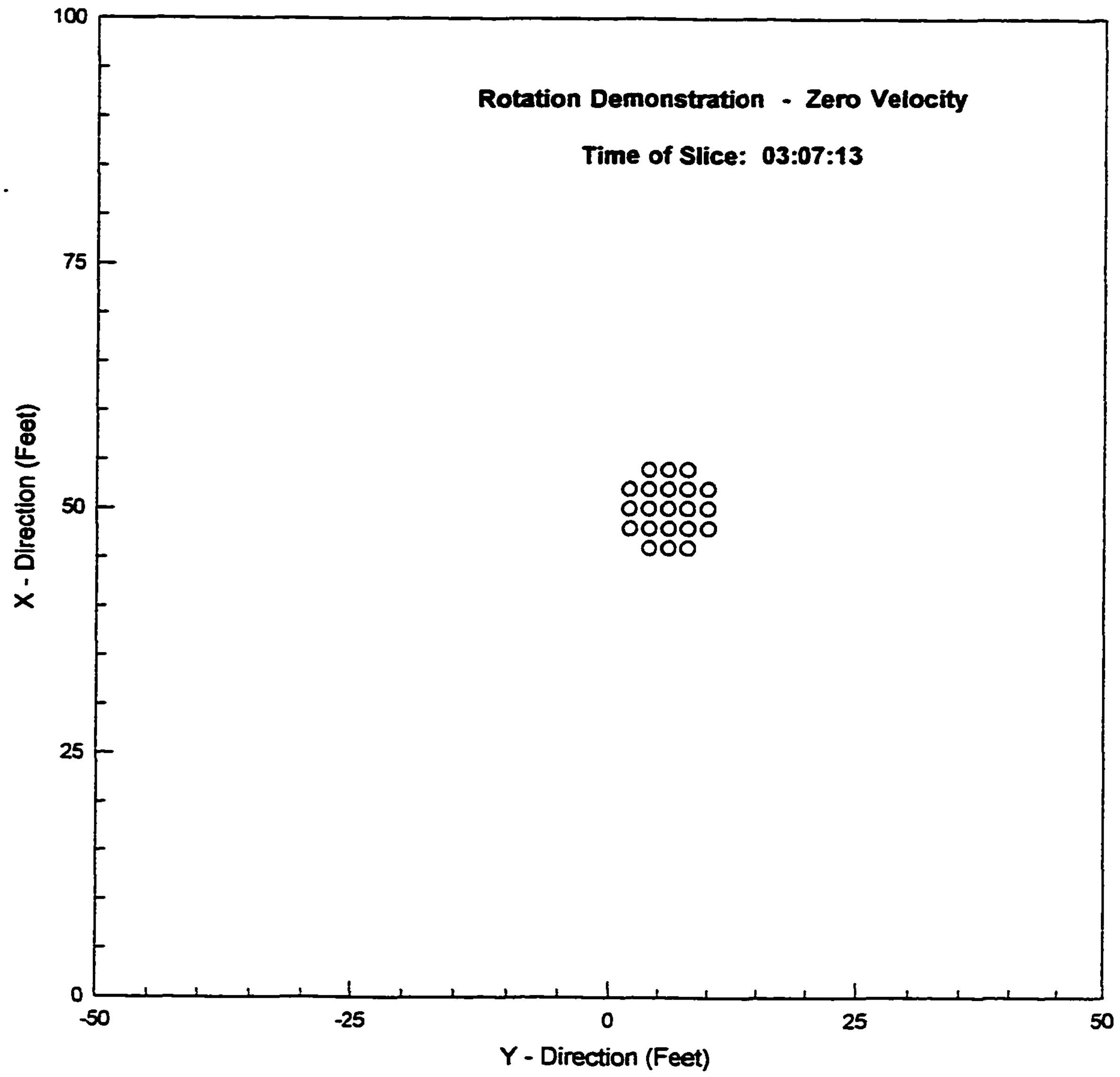
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



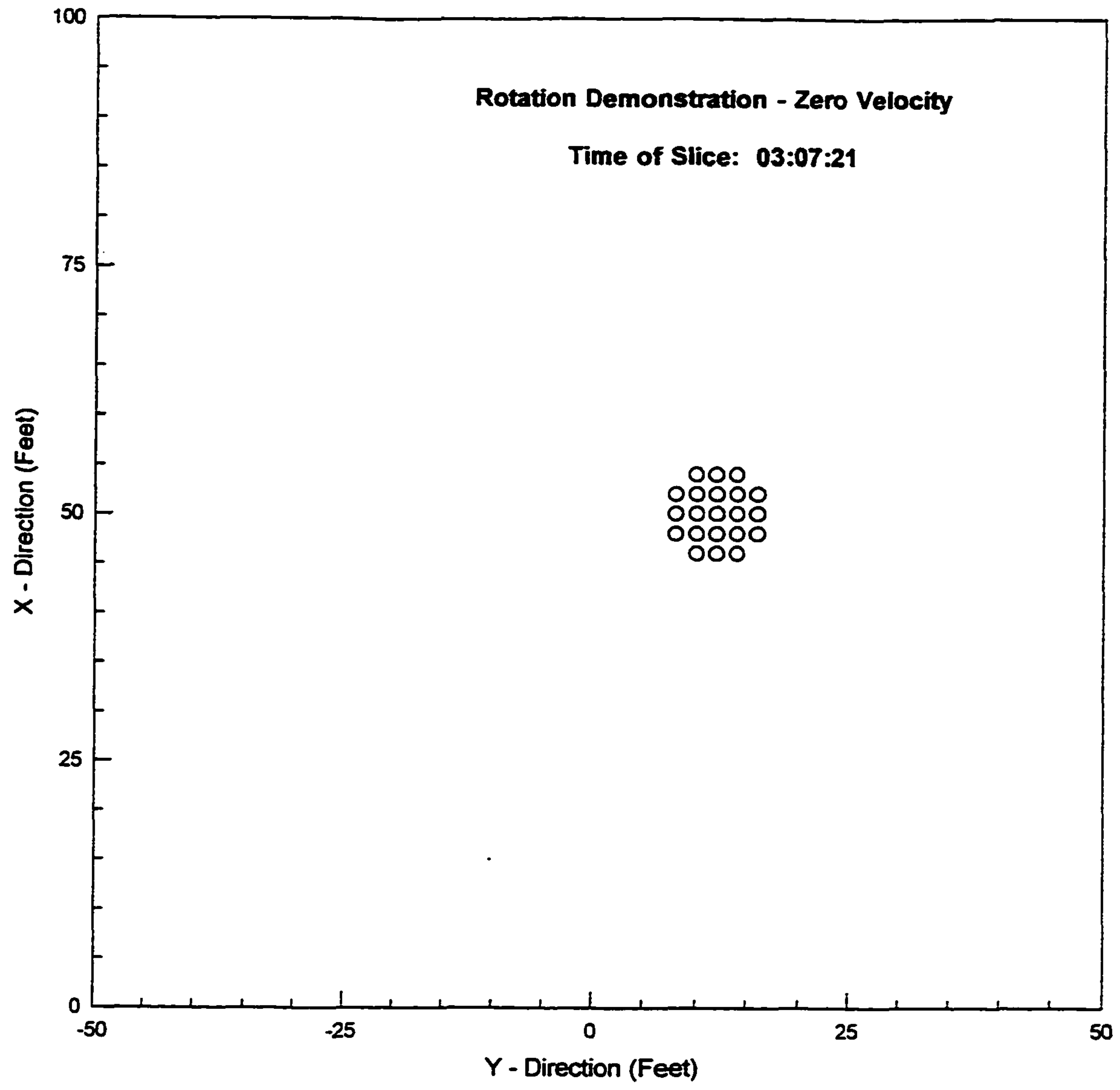
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

[1] Agarwal, Krishna, Quote on C/C++ Computer Language Comparisons, Professor and Chair of Computer Science Dept., Louisiana State University in Shreveport, January, 1999.

[2] Auran, Per G. and Malvig, Kjell E., "Clustering and Feature Extraction in a 3D Real-Time Echo Management Framework", IEEE Symposium on Autonomous UnderwaterVehicle Technology, Ocean Engineering Society of the IEEE, June 2-6 1996, Monterey, CA.

[3] Auran, P. G., and Silven, O., "Ideas for Underwater 3D Sonar Range Sensing and Environmental Modeling", Modeling, Identification and Control, Vol. 17, No. 1, p63-67, January 1996.

[4] Auran, Per G. and Malvig, Kjell E., "Real-time Extraction of Connected Components in 3-D Sonar Range Images", IEEE Computer Society Press, Los Alamitos, CA, 1996.

[5] Auran, P.G. and Silven, O., "Underwater Sonar Range Sensing and 3D Image Formation", Control Engineering Practice, Vol. 4, No. 3, pp. 393-400, March 1996.

[6] Azhazha, V. G., and Shishkova, E. V., <u>Fish Location by Hydroacoustic Devices</u>, <u>Israel Program for Scientific Translations Ltd, 1967</u>.

[7] CGN, Drawing No. CGN1001-232 Rev. A, CGN, Sunnyvale, CA., 1998.

[8] Clarke, John E. Hughes, "Detecting Small Seabed Targets Using High Frequency Multibeam Sonar", Sea Technology, pp 87-90, June 1998.

[9] Cringely, Robert X., "A Fight to the Finish" The Pulpit, PBS Online, September 10, 1998.

[10] Cuschieri, J. M., LeBlank, L., Singer, M., Beaujean, P.P., "Development of a 3D Forward Look Electronically Scanned Sonar System", Oceans'96 MTS/IEEE Conference Proceedings, pp. 778-783.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[11] Davis, Paul, Quotation on C/C++ Computer Language Comparisons, UNIX Software Developer for BMC Corporation, Houston, Texas, January 22, 1999.

[12] Hackmann, William, Seek and Strike - Sonar, Anti-submarine Warfare and the Royal Navy 1914-54, Her Majesty's Stationary Office, London, England, 1984.

[13] Halliday, David and Resnick, Robert, Fundamentals of Physics Second Edition, John Wiley and Sons, New York, NY, 1981.

[14] Hearn, Donald and Baker, Pauline, Computer Graphics, Prentice Hall, 1986.

[15] Hsieh, W., "Atlantic Circulation", http://www.science.ubc.ca/~ocgy308/chap14/ch14_b.html, Earth and Ocean Sciences, University of British Columbia, Feb. 1999.

[16] Knuth, Donald E., The Art of Computer Programming, Volume 2 - Seminumerical Algorithms, Addison-Wesley Publishing Company, New York, NY, 1969.

[17] Microsoft Corporation, Microsoft MS-DOS User's Guide and Reference, Microsoft Corporation, 1991.

[18] Motorola, Inc., Linear and Interface Integrated Circuits, Motoroll, Inc., 1983.

[19] Motorola, Inc., M68HC11 Reference Manual, Motorola, Inc., 1991.

[20] Motorola, Inc., MC68HC11EVBU Universal Evaluation Board User's Manual, Motorola, Inc., 1990.

[21] National Semiconductor Corporation, Linear Databook, National Semiconductor Corporation, 1982.

[22] Precision Navigation, Inc., Vector Electronic Modules Application Notes, Version 1.08, Precision Navigation, Inc., 1998.

[23] Radio Corporation of America, RCA Power Devices, RCA corporation, 1978.

[24] Samet, Hanan, Applications of Spatial Data Structures, Addison-Wesley Publishing Company, New York, NY, 1990.

[25] Shankland, Stephen, "Unix Trounces Windows NT in Testing", CNET News.com, December 1, 1998, 9:15 p.m. PT.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[26] Shohat, Murry, "Engineers Speak Out: LINUX vs Windows NT, Part 1", Cover Story, July 1998.

[27] Smith, Ferrel C., Introduction to Communications Systems, Third Edition, Addison Wesley Publishing Company, Inc., Reading MA, 1990.

[28] Spitzak, Sharon E., Caress, David W., and Miller, Stephen P., "Advances in Realtime Multibeam SurveyVisualization and Quality Control", Oceans'98 MTS/IEEE Conference Proceedings, Vol. 2, pp. 975-977, Sept 1996.

[29] State of Louisiana Cypress-Black Bayou Recreation and Water Conservation District, Site No. 1 Reservoir Coutour Map, June 1972.

[30] Stevenson, Alexander, "Voxels and Volumetric Representation", University of British Columbia, Vancouver, BC, http://www.intergate .bc.ca/sagax/ voxels/voxels.htm, Feb. 1999.

[31] Tsao, Che-Chih, and Chen, Jyhshing, "Moving Screen Projection: A New Approach for Volumetric Three-Dimensional Display", SPIE, Vol. 2650, pp 254-264.

[32] United States Navy, "Frontier-Based Exploration - Detecting Frontiers", http://www.aic.nrl.navy.mil/~schultz/research/ frontier/detect.html, Feb. 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

Gary R. Boucher was born on April 25, 1950 and grew up in the town of Springhill, Louisiana. He was from an early age interested in technology and the physical sciences. Throughout his school years he continued his interest in science as he developed skills to support his home-made engineering efforts. In middle school he was interested in chemistry and in high school his primary interest was electronics.

In 1972 he obtained a B.S. degree in Electronic Engineering Technology from Northwestern State University in Natchitoches, Louisiana. Four years later he earned a masters degree in the same field of study. In graduate school he studied math and computer programming to augment his main course of study.

In 1976 Boucher moved to Houston, Texas where he founded a business, Microtex, Inc., involved in the sales, service, and programming of early microcomputers. After four years and much knowledge gained in this field, he returned to North Louisiana and operated Data Tech, Inc., a small computer consulting firm, while also managing the family clothing business. Both businesses did well under his management through the 1980s.

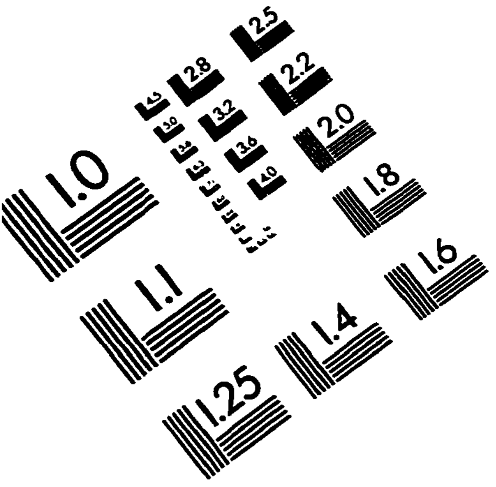
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

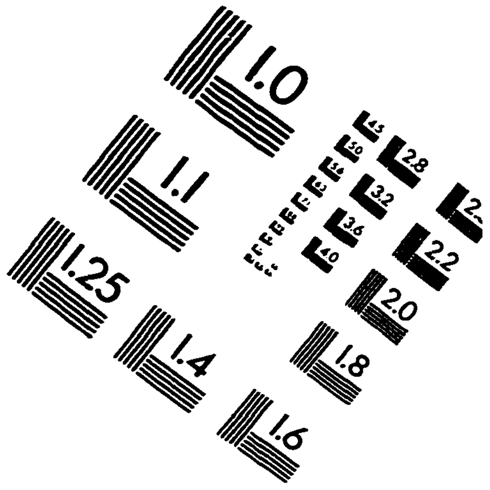
In 1984 Boucher took a position at Northwestern State University, where he worked for two years teaching in the Industrial Technology Department. In 1987, he took a position with Louisiana State University in Shreveport. There he currently works in the departments of Chemistry-Physics, Computer Science, and Math. His main teaching responsibilities are sophomore level physics, five electronics courses, and two computer architecture courses.

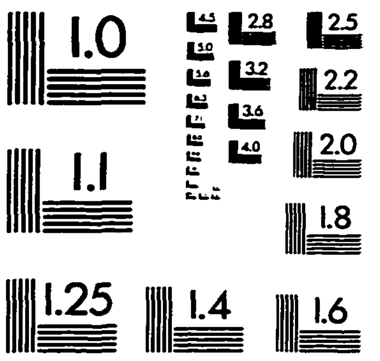
A masters degree in Electrical Engineering was awarded to Boucher in 1995 from Louisiana Tech University in Ruston, Louisiana. Currently he is finishing the dissertation for the Doctor of Engineering Degree from Louisiana Tech.

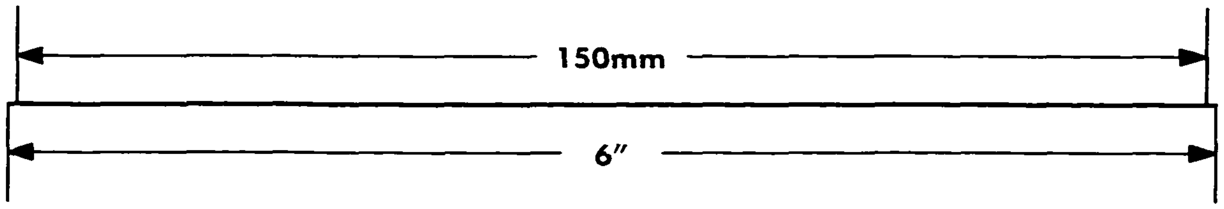
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

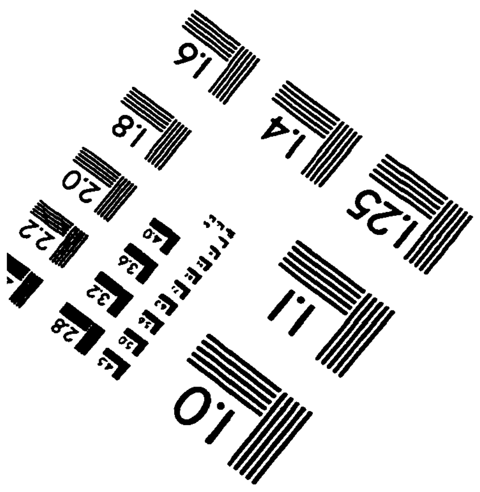
IMAGE EVALUATION
TEST TARGET (QA-3)

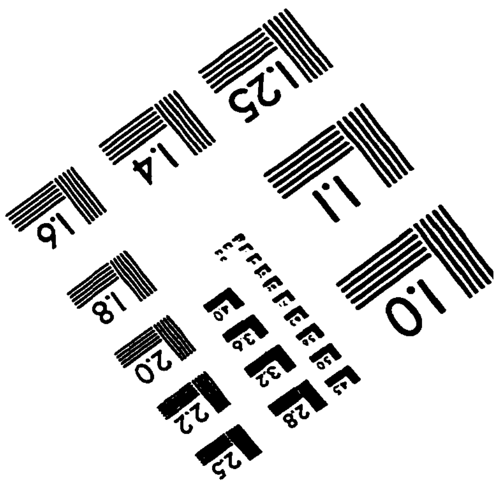












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.