Sensor Integration for Autonomous Robotic Watercraft

Bruce White
Master of Science in Electrical Engineering
Applied Research Project
San Francisco State University
May 2006

Table of Contents

1. Background

In today's world, many tasks considered dull, dirty, or dangerous are increasingly performed by robotic and autonomous systems. Such is the case with the collection of scientific data. Exhibiting superior efficiency, precision, reliability and repeatability, autonomous systems allow measurement without human oversight or intervention. This streamlines the scientific process, allowing resources once required for data collection to be reallocated to data analysis, facilitating faster interpretation and hastening breakthroughs.

The goal of this project was to investigate sensor integration possibilities for an autonomous boat created in the San Francisco State University Autonomous Vehicle Lab[1]. Built by Mechanical & Electrical Engineering students and overseen by Dr Michael Holden, PhD, Assistant Professor of Mechanical Engineering, the boat was intended for use by students in San Francisco State's Civil Engineering program to autonomously describe lake bottom contours and monitor water quality in Lake Merced[1], a reservoir in San Francisco.

2. Logistics

The boat was to be driven by a navigation computer attached to a Global Positioning System (GPS)[2]. The minutia of this system will not be discussed here, but a brief overview is necessary for full comprehension of navigation and sensor system linkages. A predetermined course would be entered into the navigation computer with specific waypoints along the route[2], and the boat would be placed in the water and activated. The navigation computer would then use the real-time data from the GPS to determine the boat's current location, and steer and throttle accordingly to reach the next waypoint[2]. These waypoints would be ideal locations to take sensor readings, as any data would be meaningless without an associated location.

The purpose of the boat was to describe lake bottom contours and monitor water quality, and each of these tasks would require its own dedicated sensor. In the early planning stages it was noted that the water quality sensor likely to be used would be bulky and require a deeper submersion than the hull of the boat itself, whereas the depth sensor would not have the same constraints. As such, it was decided the boat would set out initially with only the depth sensor. One course

would be completed by the boat, with a depth measurement taken at each waypoint along the route.

Once the course was completed, the boat would return to shore and the sensor computer would be connected to a laptop for transfer of depth results. Based on this, the water quality sensor tow cable length would then be changed to prevent the sensor from being accidentally dragged on the bottom of the lake and damaged. Assuming a safe depth at each waypoint, the boat would then be sent back out with only the water quality sensor, to repeat the same course. Upon completion of the course, the boat would return to shore and the results would be transferred to the laptop for later analysis.

3. Sensors

Lake Bottom Depth: To map lake bottom contours, a depth sensor was required. Humminbird fish finders was contacted, and they graciously donated one of their Matrix 55 depth finders. To detect lake bottom depth, the Matrix uses a transom-mount sonar transducer coupled with a transceiver/central processing unit.

Figure 1 – Humminbird Matrix 55 Transceiver and Sonar Transducer[3]

Between 10-20Vdc is required to power the sensor[4], and as such it had to be wired directly to the boat power supply, a 12V sealed lead-acid battery.

The Matrix transducer is connected to a transceiver, which performs analog-to-digital conversion of the sonar signal. Digital data are sent from the transceiver via RS-232 serial communication protocol at 4800 baud, 8 data bits, no parity, and 1 stop bit[5]. Only a passive mode of operation is available, so the Matrix cannot be externally triggered to take a reading. A string of 8-bit ASCII characters is output once per second from the device, each "sentence" adhering to the NMEA 0183 standard[5], and reading the data is a matter of recognizing and storing the desired parameters within the string. The depth reading is output only in meters.

Water Quality: To adequately monitor water quality, the obvious first choice of instrument was an integrated device with multiple sensors. Selection was left to students in the San Francisco State University Civil Engineering Department, who would be doing the analysis of any water quality data. The final choice was the Hach Hydrolab MS4a MiniSonde. Measuring water temperature [C, F, K], pH [pH Units], sensor depth [m, f, psi], dissolved oxygen [%sat, mg/L], and specific conductivity [mS, µS][6], it was an appropriate choice.

Figure 2 – Hach Hydrolab MS4a MiniSonde[7]

The Sonde runs on 12Vdc supplied by 8 internal AA batteries[6]. This simplified power requirements for the boat, as it was not necessary to connect the instrument to the boat's internal battery.

The Sonde communicates digitally via RS-232 serial communication protocol at 9600 baud, 8 data bits, even parity, and 1 stop bit[8]. It features Modbus 3 and TTY modes of operation[8] that allow for both

active and passive communication, respectively. In Modbus mode, the Sonde is idle until a specific 8-byte hexadecimal string is sent to wake it up[8]. Once awake, further 8-byte hex strings can be sent to activate one or more of its internal sensors[8]. Once a sensor reading is taken, the result is returned as a 9-byte hex string[8]. In TTY mode, the Sonde outputs a string of 8-bit ASCII characters once per second[6], and reading the data is a matter of recognizing and storing the desired parameters within the string.

4. Sensor Computer

Hardware: To take water-quality measurements, the most flexible option was to use the Sonde's Modbus 3 mode, which required active triggering and detection of a reading. Conversely, the Matrix's effectively constant data stream only required passive detection. Either way, the results had to be received and stored by an outside system. Moreover, a low-power, small-form factor solution was also desired since space and battery life were at a premium. These constraints were met by using an Atmel ATmega32 microcontroller as the heart of the sensor computer.

Power to all boat hardware, with the exception of the Sonde, is provided by a 12V 7Ah sealed lead-acid battery. From this battery, an LM7805 voltage regulator provides 5Vdc to the Atmega32 and other peripherals that comprise the sensor computer. Peak power usage for the entire sensing system is approximately 4.265W (figure 3).

Component	Power
ATmega32	35mW[9]
Power Indicator LED	160mW
MAX232	696mW[10]
Matrix 55	1.344W
LM7805	2.03W
Total	**4.265W**

Figure 3 – Sensing System Component Power Usage

To facilitate asynchronous serial communication, the ATmega line of chips is commonly used in conjunction with a MAX232 RS-232 driver integrated circuit to convert between the RS-232 (±3-25V) logic of a PC or other peripherals, and the TTL (0-5V) logic of the microcontroller itself. Dr Michael Holden already had a printed circuit board laid out for general Atmel ATmega usage with a MAX232. This board is widely

used by students in the SFSU Autonomous Vehicle Lab, and was an appropriate choice for the main board of this project. The sensor computer was assembled on one of these using mostly surface-mount parts.

Components include one Atmel ATmega32L, one MAX232, five 1µF ceramic capacitors required for MAX232 operation[10], and various header pins for outside connections to pins on the ATmega32. Serial communication is achieved via the T1OUT (transmit) and R1IN (receive) pins of the MAX232[10], connected to pins 3 and 2 of a DB-9 connector, respectively. Pin 5 of the connector is wired to ground. See figure 4 for the full schematic.

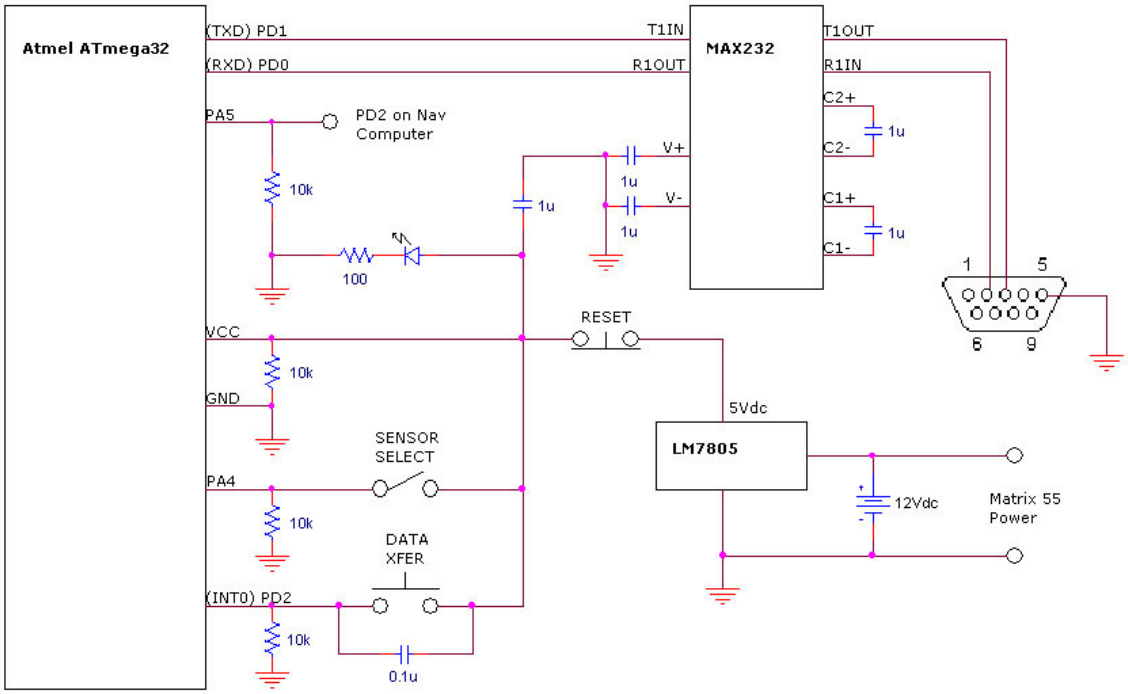


Figure 4 – Sensor Computer Schematic

An interface had to be created for the sensor computer to facilitate sensor selection, chip reset, and data transfer. A small square of prototyping PCB serves as the base for this interface board. For sensor selection, one terminal of a single-pole single-throw switch connects to pin 4 of port A (PA4) on the ATmega32, and the other terminal to 5Vdc. When the switch is open (0Vdc at PA4), the ATmega32 goes into water-quality measurement mode. When the switch is closed (5Vdc at PA4), lake bottom depth measurement mode is selected. A 10kΩ pull-down resistor is also connected between PA4 and ground, to assure 0Vdc at the pin when the switch is in the open state.

For chip reset, one terminal of a normally-closed momentary switch is connected to the VCC pin of the ATmega32, and the other terminal to 5Vdc. The switch normally allows power to flow to the ATmega32 and MAX232 chips for standard operation. When pressed, power to both devices is cut until the switch is released, at which time any data stored in the ATmega32 are erased and the program reloaded.

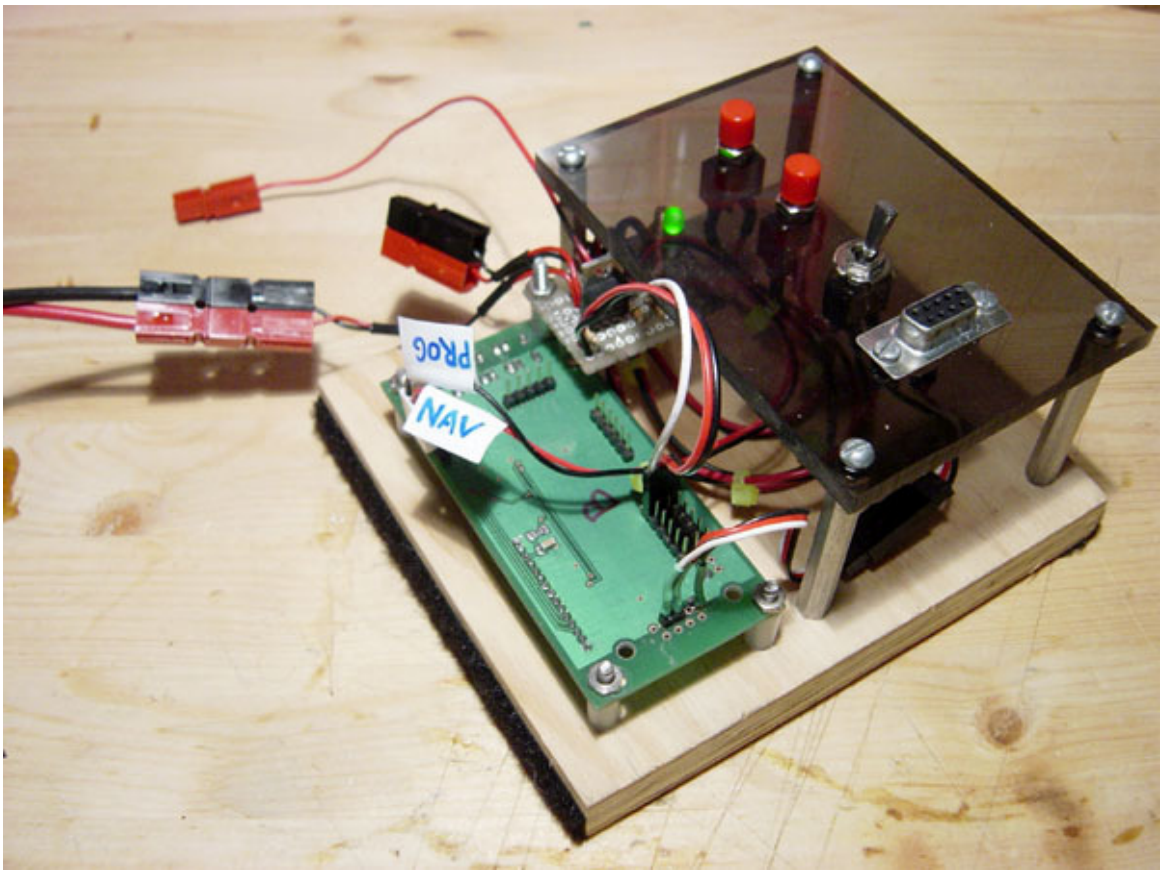


Figure 5 – Fully Assembled Sensor Computer

For data transfer, one terminal of a normally-open momentary switch is connected to pin 2 of port D (PD2) on the ATmega32, and the other terminal to 5Vdc. When the switch is pressed, the voltage of PD2 goes from 0 to 5V, triggering an interrupt in the software that dumps all stored data to the serial port. A 0.1μF capacitor is connected in parallel with the switch to prevent any bounces from registering as multiple presses in the software[11]. A 10kΩ pull-down resistor is connected between PD2 and ground, to assure 0Vdc at the pin when the switch is in the open state.

The final interface connection is a simple wire from pin 5 of port A (PA5) on the ATmega32 to pin 2 of port D (PD2) on the navigation computer. When the boat reaches a waypoint, software in the navigation computer sets its PD2 to 5V which is registered at PA5 of the ATmega32, triggering the software to take a reading. A 10kΩ pull-down resistor is connected between PA5 and ground, to assure 0Vdc at the wire when not at a waypoint.

Between 5V and ground, an LED is wired in series with a 100Ω resistor to serve as a power indicator light. The LM7805 voltage regulator mentioned earlier is also contained on this board, as well as an input for 12V from the boat battery and a 12V output to power the Matrix.

A 5” x 5.5” piece of tinted lexan with strategically drilled holes serves as a panel for the reset button, data dump button, sensor select switch, and DB-9 connector. The main board, interface board, and lexan piece are all mounted on standoffs of varying heights, which are screwed onto a small piece of plywood. The result is an organized and compact user interface (figure 5).

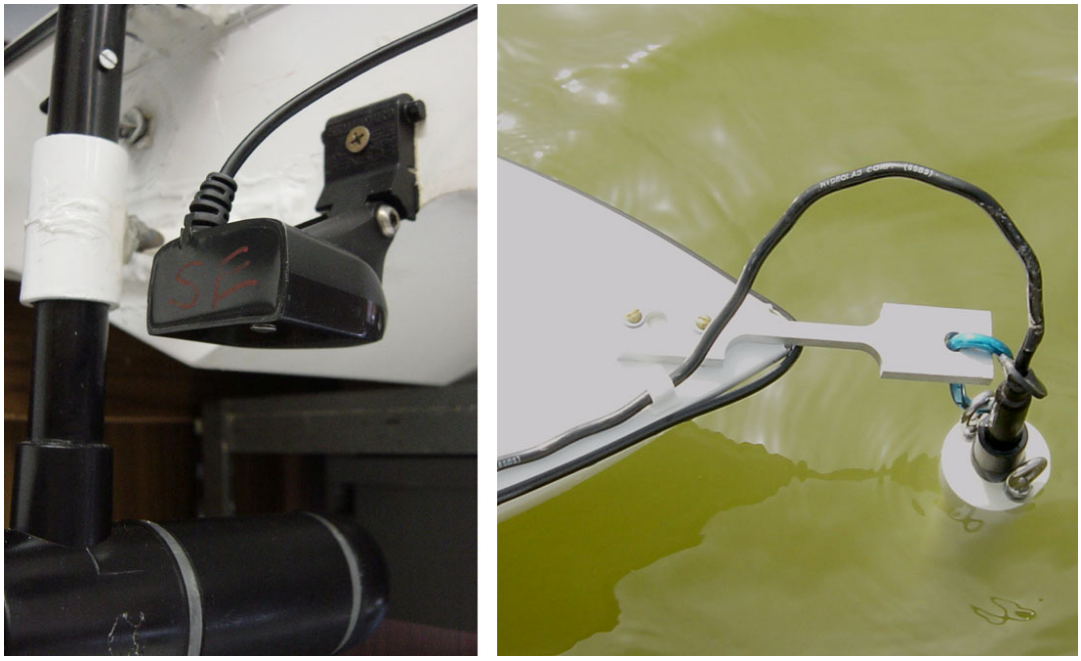


Figure 6 – Sensor Mount Configurations

Two strips of velcro, adhered to the inside of the boat hull just inside of the front access hatch, secure the sensor computer in place. A similar system is used to secure the Matrix transceiver just aft of that. The Matrix’ sonar transducer is screw-mounted to the transom at the aft end of the boat. An aluminum plate is bolted to the fore deck, and

a carabiner is threaded through a hole drilled in the end of it. The Sonde is then secured in place by clipping it to the carabiner via a screw eye on the Sonde collar (figure 6).

Software: The ATmega32 can be programmed using assembly or C languages[9]. For this project, C was used exclusively. The full code can be found in the appendix, and specific line numbers will be used in this section for reference to it. Upon power-up, the software in the ATmega32 enters an initialization process (1-35). All program variables are defined, including characters, arrays, floating point numbers, and counters. In any C code written for a PC, getchar and putchar routines retrieve characters from the keyboard and send characters to the screen respectively. In an embedded system such as this one that has no keyboard or monitor, the getchar and putchar routines use the serial port to retrieve and send characters. At this point in the software, these routines are redefined to check for even parity with every character transmitted or received in even parity mode, as communication with the Sonde requires this (38-82). Next comes the data transfer interrupt code, which will be discussed later. Various ports and timers are then initialized, and the ATmega32's USART (serial port) and interrupts are enabled (138-238).

The program then checks PINA.4 of the ATmega32 to see whether 5V is present. This pin is connected to the sensor select switch on the interface panel. If the pin is at 5V, the program enters Matrix (lake bottom depth) mode (247). Otherwise, Sonde (water quality) mode is selected (473). See *Initialization Program Flowchart* in the appendix for a visual representation of initialization program flow.

If Matrix mode is selected, the program initializes the ATmega32 to 4800 baud, 8 data bits, no parity, & 1 stop bit; required parameters for communicating with the Matrix (475-477). This is followed by an indefinite idle period until 5V is seen at PINA.5 (482-489). This pin is connected to the navigation computer, and a 5V signal indicates that the boat has reached a waypoint. Once PINA.5 returns to 0V, the program analyzes each incoming character at the serial port until hex 0x50 (ASCII letter P) is detected (494-497). The only time the Matrix outputs P is just before a depth reading, which can be viewed in a terminal window as DPT,X.X, where the X's are depth digits[5]. Once P is detected, the next five characters are temporarily stored (498-502). The fourth character, which is the first digit of the depth reading, is reassigned to a slot in an array via a digit counter. The digit counter is then incremented, and the sixth character, which is the second digit of the depth reading, is stored in a similar manner. The digit counter is

then incremented again (503-506). At this point a variable, *cntstop*, is assigned the number in the master counter, and the master counter incremented (508,509). The ATmega32 has 1000 bytes of storage[9], and each Matrix reading is 2 bytes, so no more than 500 depth readings may be taken. Using the number assigned to the variable *cntstop* from the master counter, a *while* loop stops depth operations if 500 readings have been taken (510-513). At this point the program again idles indefinitely until 5V is seen at PINA.5, when the loop resumes again for another reading. See *Matrix Program Flowchart* in the appendix for a visual representation of Matrix program flow. In this configuration the sensor computer cannot measure lake depths greater than 9.9 meters. However, this is still greater than the deepest water level recorded at Lake Merced since at least the mid-1960's[13], so it is more than adequate.

If Sonde mode is selected, the program initializes the ATmega32 to 9600 baud, 8 data bits, even parity, & two stop bits; required parameters for communicating with the Sonde (249-251). The Sonde specification stipulates only one stop bit, but this proved problematic (see *Troubleshooting*). The program then waits for the Sonde's plug-in string. When the Sonde is plugged into any active serial port, it automatically sends an initial hexadecimal string of 0x3F3F3F3FFB. The ATmega32 software gets each of these characters at the serial port until 0xFB is detected (253-256). After waiting for one second, an 8-byte hex string is sent to the Sonde three times, with a one second delay between each query, to wake the Sonde from sleep mode permanently[8] (257-288). This is followed by an indefinite idle period until 5V is seen at PINA.5 (295-302). This pin is connected to the navigation computer, and a 5V signal indicates that the boat has reached a waypoint. Once PINA.5 returns to 0V, a round of sensor readings begins. Each of the five sensors in the Sonde can be triggered by a specific 8-byte hex string (figure 7)[8].

Sensor	Hex Activation String
Sensor Depth [m]	01 03 00 30 00 02 C4 04
Temperature [C]	01 03 00 00 00 02 C4 0B
pH [pH Units]	01 03 00 06 00 02 24 0A
Dissolved Oxygen [mg/L]	01 03 00 16 00 02 25 CF
Conductivity [mS]	01 03 00 0A 00 02 E4 09

Figure 7 – Hex Activation String for Each Sonde Sensor[8]

Each byte of a given activation string is critical to commanding the specified reading from the Sonde (figure 8). If any byte is incorrect, the sensor will not function. The ATmega32 queries the Sonde for a

sensor depth reading by sending the string 0x010300300002C404 (307-314). The Sonde responds immediately with a result in the form of a 9-byte hex string (figure 9)[8].

Byte	Hex Number	Assignment
1	01	Slave Address
2	03	Modbus Command 3 – Read Holding Registers
3	Varies	Address of First Register – High Byte
4	Varies	Address of First Register – Low Byte
5	Varies	Number of Registers to Read – High Byte
6	Varies	Number of Registers to Read – Low Byte
7	Varies	Cyclic Redundancy Check – High Byte
8	Varies	Cyclic Redundancy Check – Low Byte

Figure 8 – Byte Designations of a Sonde Activation String[8]

The first two bytes of the response string are ignored. The ATmega32 proceeds to take in the result once the third byte, 0x04, is detected. The four bytes of data following contain the sensor result, and are stored in a dummy array. The final two cyclic redundancy check bytes are ignored (316-325).

Byte	Hex Number	Assignment
1	01	Slave address
2	03	Modbus Command 3 – Read Holding Registers
3	04	Number of Registers Read x 2
4	Varies	Sensor Reading – Low Word, High Byte
5	Varies	Sensor Reading – Low Word, Low Byte
6	Varies	Sensor Reading - High Word, High Byte
7	Varies	Sensor Reading - high Word, Low Byte
8	Varies	Cyclic Redundancy Check – High Byte
9	Varies	Cyclic Redundancy Check – Low Byte

Figure 9 – Byte Designations of a Sonde Response String[8]

For correct display later, the sensor result must be converted to a floating point number. The 4-byte IEEE floating point number format[12], readable by C, stipulates a byte order of high word/high byte, high word/low byte, low word/high byte, low word/low byte. The four stored bytes containing the sensor reading, as received from the Sonde, are not in this order. The issue is resolved by taking the four bytes from the dummy array and rearranging them into a second array in the correct order. At this point they are copied sequentially into the memory location of a float variable and are ready for later output in the correct format (327-332).

The program waits one second before proceeding to repeat the same query, response, and rearrangement process for temperature, pH,

dissolved oxygen, and conductivity (338-456). Then a variable, *cntstop*, is assigned the number in the master counter, and the master counter incremented (461,462). The ATmega32 has 1000 bytes of storage, and each Sonde reading is 4 bytes x 5 sensors, so no more than 50 readings may be taken. Using the number assigned to the *cntstop* variable from the master counter, a *while* loop stops depth operations if 50 readings have been taken (463-466). At this point the program again idles indefinitely until 5V is seen at PINA.5, when the loop resumes again for another reading. See *Sonde Program Flowchart* in the appendix for a visual representation of Sonde program flow.

Once all the required Matrix or Sonde readings are recorded, the user must be able to access the data. This is done by pressing the DATA button on the interface panel, which briefly sends 5V to pin 2 of port D (PD2) on the ATmega32, which is also the chip's external interrupt pin[9]. Once an external interrupt is detected, the software breaks from whatever it is doing and executes the interrupt code. If the button is pressed and the *cntstop* variable is 0 (no readings taken), "No sensor readings taken!" is printed to the serial port (93-96). Otherwise the program looks to see which sensor is currently selected by detecting the voltage at PINA.4 of the ATmega32 (99).

If the Sonde is selected (PINA.4 is 0V), one waypoint number and its associated sensor depth, temperature, pH, dissolved oxygen, and conductivity readings are all printed to the serial port (figure 10). The master counter, set to zero at the start of the interrupt code, increments and prints the next reading in the same fashion (104-111). The process ends when the number in the master counter is greater than the number of total readings saved in the *cntstop* variable (102).

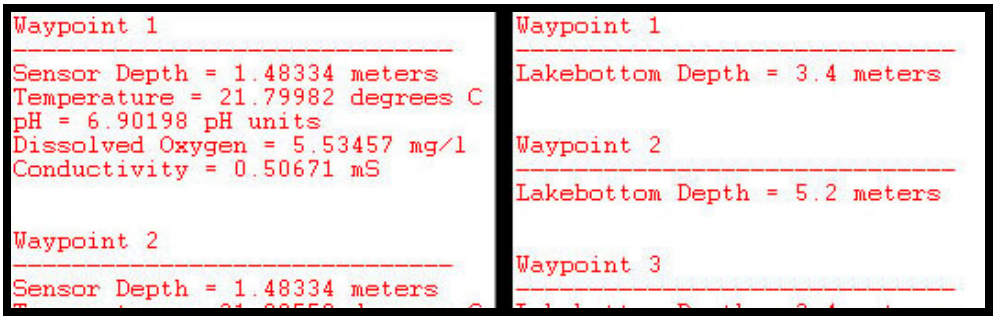


Figure 10 – Data Transfer Screenshots from Both Sensors

If the Matrix is selected (PINA.4 is 5V), one waypoint number and its associated lake bottom depth reading are printed to the serial port (figure 10). The master and digit counters, set to zero at the start of

the interrupt code, increment once and twice respectively in the course of printing one reading (120-129). The process ends when the number in the master counter is greater than the number of total readings saved in the *cntstop* variable (118). See *Interrupt Program Flowchart* in the appendix for a visual representation of interrupt program flow.

5. Troubleshooting

A number of problems were encountered during the course of this project. The Sonde communicates at 9600 baud, so the ATmega32 had to be set to this baud rate as well. ATmega32s come with their system clock set to a default frequency of 1 MHz. At this clock rate, at 9600 baud, an error was introduced which made communication impossible. Transmissions from the Sonde were seemingly random strings of strange characters, a far cry from the specific 9-byte string expected. The baud rate of an ATmega32 is set via its USART Baud Rate Register (UBBR) which can only contain integers[9]. Communication at 9600 baud using a 1MHz system clock requires that UBBR be set to 5.51, which is not possible. The closest settings of 5 or 6 allow for baud rates of 10417 or 8929, errors of 8.5 and 6.9% respectively. These errors are too large for effective communication. The solution was to set the system clock to 4MHz. At this frequency UBBR can be set to 25, which allows for a baud rate of 9615. The resulting error was only 0.15% off of the desired baud rate of 9600, well within reasonable limits. This also required modifying all baud rate initializations in the code to comply with the new system clock frequency.

By far the most perplexing problem encountered was an anomaly where the Sonde would readily respond to an 8-byte hex string sent from a terminal program on a PC, but would ignore the exact same string sent from the ATmega32. Cables and adaptors that allowed for viewing of transmissions between the Sonde and ATmega32 were purchased and soldered together, verifying that the hex strings sent from the terminal window and the ATmega32 were seemingly identical. Finally, an oscilloscope was used to analyze the outputs of both devices, and the results were telling. At 9600 baud, one bit should be 104 µS long. Since one frame from each device contained one start bit, 8 data bits, one parity bit, and one stop bit for a total of 11 bits of data, the total frame length should have been 1144 µS. However, in this case the stop bit had to be discounted; it could not be seen since it was a 0V bit that blended seamlessly into the 0V idle signal that followed it. So the lengths of the frames were considered using only the visible bits, which now should have been 1040 µS long.

The oscilloscope showed that the frame length of the terminal was longer than 1040 µS, approximately 1175 µS, and that the frame length of the ATmega32 was shorter, approximately 1000 µS (figure 11). It was immediately evident from this that the Sonde prefers a too-long frame to a too-short one. The most probable explanation is that while the frame from the terminal was too long, the excess length of the stop bit easily allowed the Sonde to see the end of the string and ready itself for the next one. Conversely, the Sonde did not register the shortened stop bit of the ATmega32 as a full stop bit, so was never able to recognize the end of the string. The problem was entirely resolved by modifying the ATmega32 code to send two stop bits with every frame, which the Sonde could easily recognize as the end of the string.

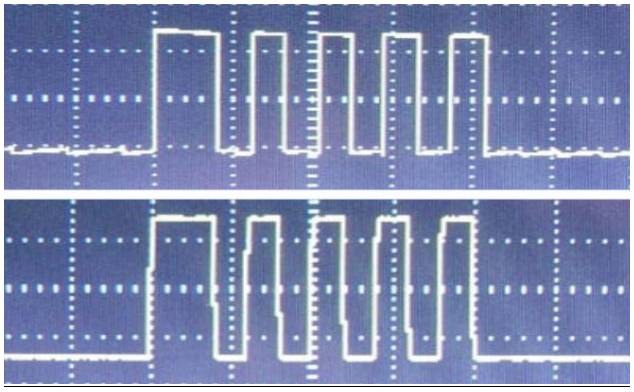


Figure 11 – Terminal Frame Length (Top) & ATmega32 Frame Length (Bottom) @ 250µS/Div

The final problem encountered involved the ATmega32's inability to recognize the first incoming byte of a response just after sending out a string to the Sonde. Each Sonde reading was triggered by an 8-byte hex string sent by the ATmega32, and the Sonde immediately responded at a rate seemingly too quick for the ATmega32. Many different code tricks were tried, without success, to obtain the first byte of the response string. The string could not be delayed because the Sonde itself could not be reprogrammed. The first three bytes of any Sonde response do not vary, so the problem was resolved by looking for the third byte (0x04) of the response string instead of the first one.

6. Operation

Lake Bottom Depth: To prepare for a lake bottom contour course,

open the fore deck hatch on the boat. Be sure that power is not connected to the sensor computer. Check to see that the sonar transducer, serial and power cables are all plugged into their respective sockets on the Matrix. Each socket is specific to its plug, so there should be no confusion (figure 12).

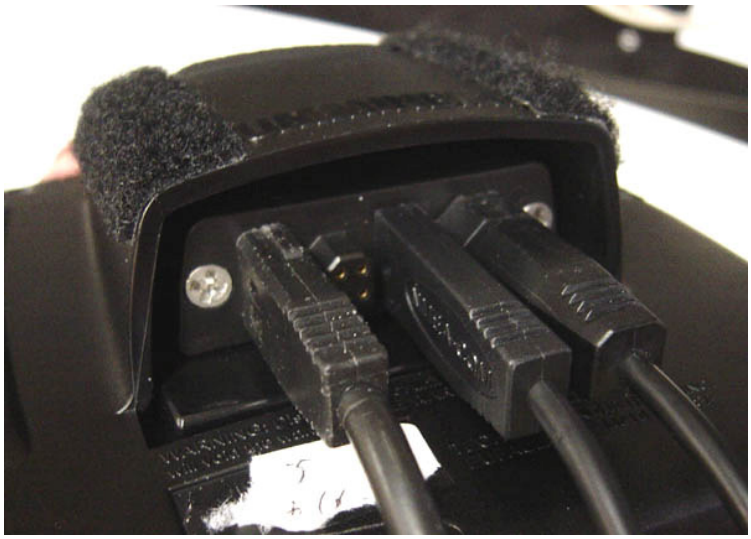


Figure 12 – Matrix Sensor Connections: (Right to Left) Power, Serial Cable & Sonar Transducer

Connect the red & black power connector from the Matrix into *either* red & black power connector on the interface board, it does not matter which (figure 13).

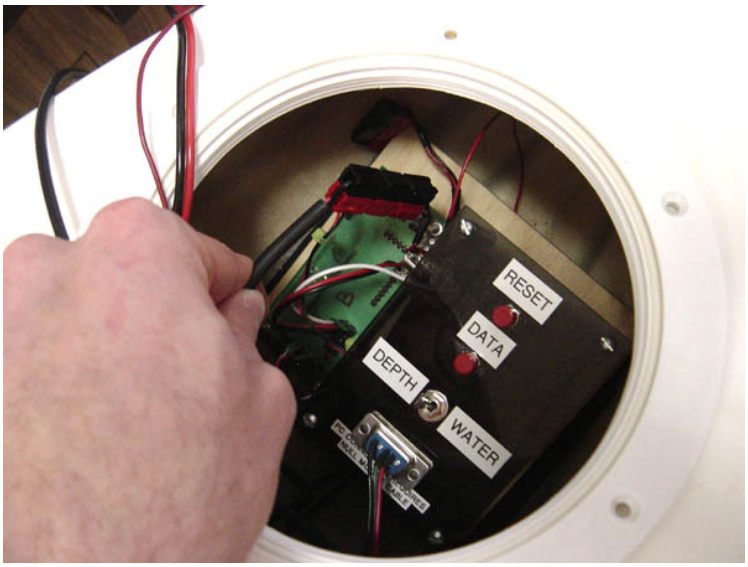


Figure 13 – Connecting Power to the Matrix

Connect power from the battery to the interface board on the sensor computer via the remaining red & black connector; the power LED should light up. Connect the red waypoint indicator cable from the navigation computer to the same on the sensor computer (figure 14).

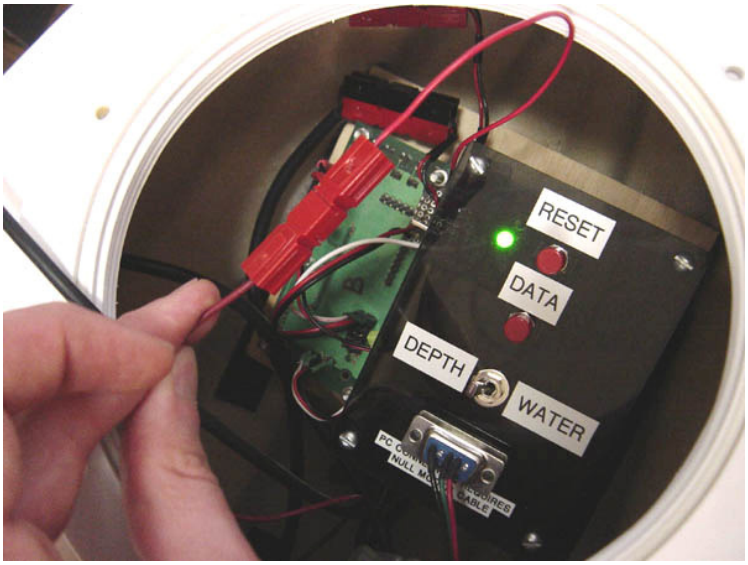


Figure 14 – Connecting the Navigation Computer Waypoint Indicator Cable

Press the POWER button on the Matrix 55. The device will turn on and eventually stop at a government warning screen. Press the EXIT button one time to exit this screen; the Matrix will automatically go into depth mode.

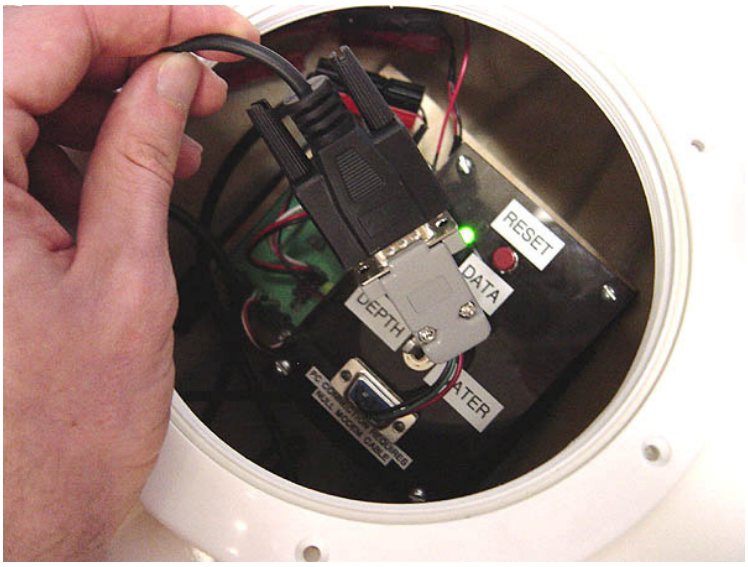


Figure 15 – Connecting The Matrix Serial Cable

Connect the serial cable from the Matrix to the DB-9 connector on the sensor computer (figure 15), and close the fore deck hatch.
The boat is now ready for a lake bottom depth course, and the sensor computer will proceed to take a reading when alerted by the navigation computer that the boat has reached a waypoint.

Water Quality: With the boat on land, clip the Sonde into the carabiner at the front of the boat via either screw eye on the Sonde collar. Be sure that the cable is plugged into the Sonde, and the cable collar tightly screwed down to prevent water from entering. Check that the other end of the cable runs through the cable-stays on the deck and into the entry notch on the aft deck hatch, with no slack cable on the deck. Thread the Sonde cable through the hull until it is reachable at the fore deck hatch. Flip the sensor select switch on the sensor computer to WATER. Connect power from the battery to the interface board on the sensor computer via *either* red & black connector; It does not matter which (figure 16). The power LED should light up.

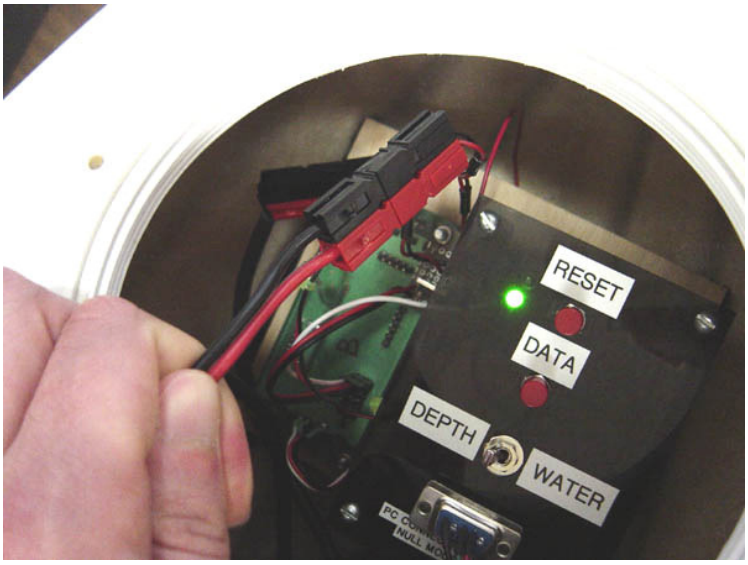


Figure 16 – Connecting Power to the Sensor Computer

If the sensor computer is already connected to power from a previous depth reading, press the RESET button before proceeding. Otherwise, connect the red waypoint indicator cable from the navigation computer to the same on the sensor computer. Plug the Sonde cable into the DB-9 connector on the sensor computer. The Sonde should emit an audible beep. Wait three seconds for the Sonde to wake up. The boat is now ready for a water quality course, and the sensor computer will proceed to take a reading when alerted by the navigation computer that the boat has reached a waypoint.

Data Transfer: Once a sensor course is complete and the boat has returned to shore, open the fore deck hatch and unplug the sensor's serial cable from the sensor computer. Take care not to bump the RESET button as this will erase any stored readings. Connect a *null modem* serial cable to the DB-9 connector on the sensor computer. *If a null modem cable is not used, the transfer will not work!* Connect the other end of the null modem cable to the serial port of any laptop running a terminal program, such as *Hyperterminal.* The terminal connection should have the following properties: 4800 baud, no parity, 1 stop bit, no flow control. Once connected to an active terminal window, press the DATA button on the sensor computer one time (figure 17).

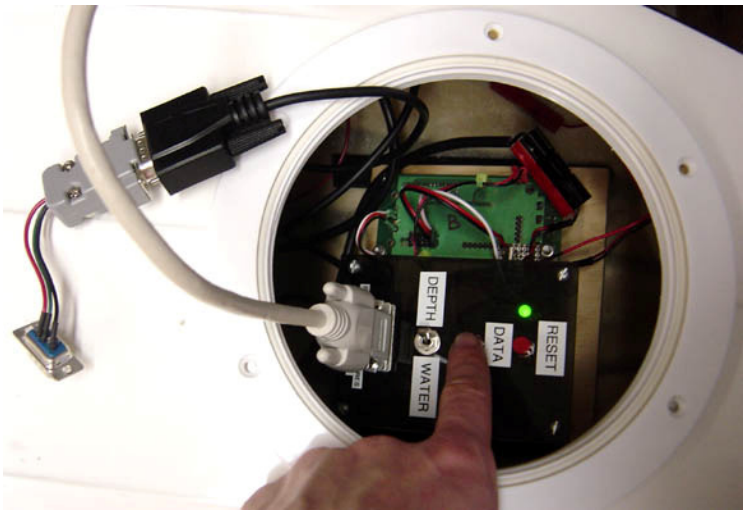


Figure 17 – Sensor Computer Data Transfer Configuration

Assuming the boat passed through at least one waypoint, data should appear in the terminal window. If no waypoints were reached, "No sensor readings taken!" will appear.

7. Results

Test runs were carried out on South Lake, the largest body of water in the Lake Merced complex, and meaningful data was successfully gathered. The course consisted of four waypoints in a roughly rectangular shape (figure 18).

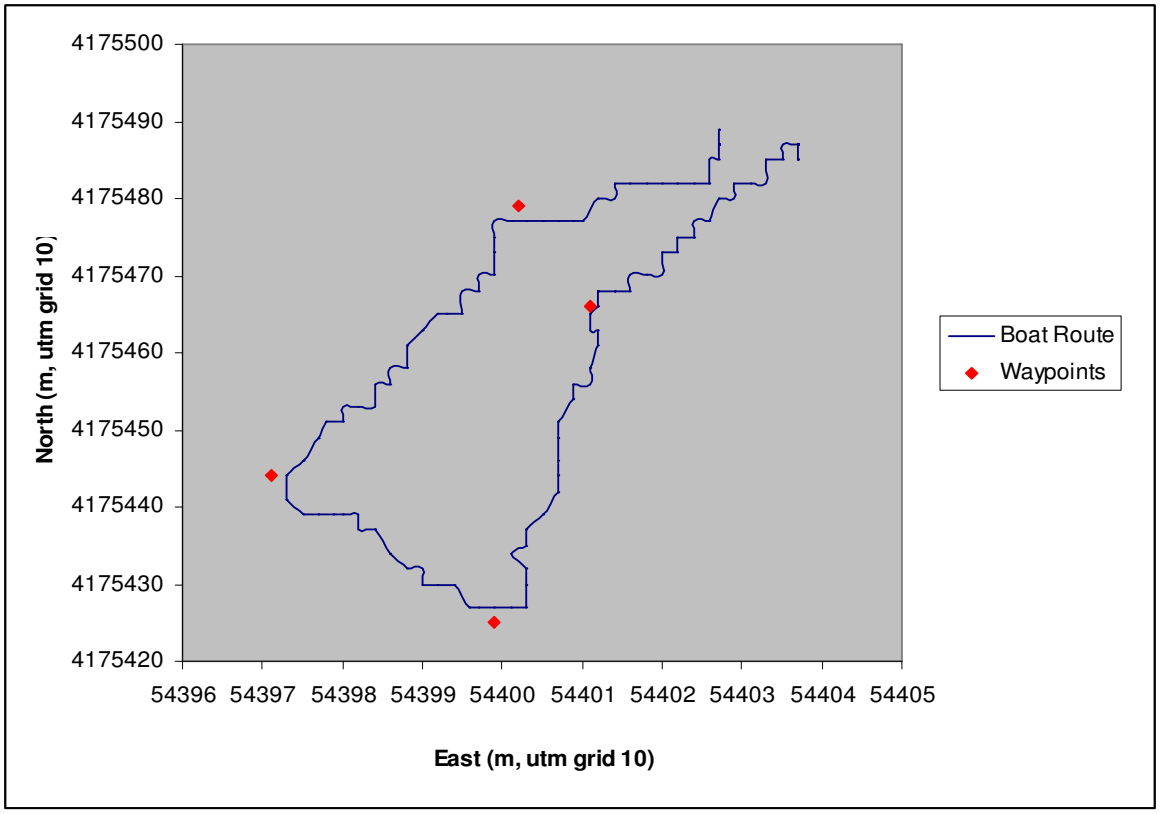


Figure 18 – Test Run Boat Route and Waypoints

Examples of captured test run data can be found in the appendix. To verify validity of results, comparisons were made between the data from the boat's first waypoint and past data taken by outside parties. Where possible, data from approximately the same time of year (late May) was used. Although deviations were to be expected because of temperature, rainfall, and pollution variation between samples, the results compared well (figure 19). All indications are that the boat and its sensing capabilities are a viable method of scientific data collection.

Measurement	Autonomous Boat (Waypoint 1)	Outside Party Results
Depth [ft]	13.4	13.0[14]
Temperature [degrees C]	19.6	19.1[15]
pH [pH units]	9.3	8.6[15]
Dissolved Oxygen [mg/L]	11.9	10.0[16]
Conductivity [mS]	0.51	0.65[16]

Figure 19 – Autonomous Boat & Outside Party Data Comparison

8. Conclusion

This project was a success. The sensor computer, on cue from the navigation computer, was able to activate the sensors and record their returning data. Once sensing was finished, the sensor computer successfully transmitted the data to a terminal program on a PC. Without question it is ready for future use by students looking to investigate lake bottom depth and water quality, as well as scientists looking to gather large amounts of data with few logistics.

This project was subjected to just about every asynchronous serial communications setback possible. Many lessons were learned. Some of the more significant examples include:

- How to deal with multiple devices communicating at different baud rates and parities within a single program.
- How to compensate for devices that are not communicating at exactly their specified baud rates.
- How to construct a single floating point number from four 8-bit hex characters in C.
- Baud rate errors can be corrected by adjusting system clock frequency.
- Occasionally a device may not be able to react quickly enough to another device to carry out a given command.

Future Work: There is room for much future work on this platform, including but not limited to the following:

- Currently the data obtained from the sensor computer must be cross-referenced to a separate list of waypoints reached by the boat during its water course, to determine which waypoint the data came from. One possible solution would be to implement serial communication from the navigation computer to the sensor computer via their respective SPI ports, allowing latitude & longitude to be appended to associated sensor data.
- The Matrix also outputs water temperature. This could be used to cross-check temperature readings taken by the Sonde.
- Code to calculate a checksum or cyclic redundancy check could be implemented to reduce errors.

9. References

1 – Holden, M. "Autonomous Water Quality Measurements"
2 – Holden, M. "Low-Cost Autonomous Vehicles Using Just GPS", American Society of Engineering Education general conference, Salt Lake City, Utah, June 2004
3 - Image taken from www.humminbird.com
4 – Humminbird, "Matrix 55 & 65 Operations Manual"
5 – Humminbird, "Outputting Digital Depth from a Matrix Product to a PC"
6 – Hach Environmental, "DataSonde 4 and MiniSonde Water Quality Multiprobes User's Manual"
7 - Image taken from www.hachenvironmental.com
8 – Hach Environmental, "Modbus - Function 3 - READ Holding Register"
9 – Atmel Corp, "8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash - ATmega32/ATmega32L"
10 – Maxim Integrated Products, "+5V-Powered, Multichannel RS-232 Drivers/Receivers"
11 - "http://www.ece.utep.edu/courses/web3376/concepts/debounce.html"
12 – "http://en.wikipedia.org/wiki/IEEE_floating-point_standard"
13 - "http://www.lakemerced.org/Data/data.html#water"
14 – "http://sfwater.org/Files/Statistics/LakeLevelManagementPlanReport_Dec.pdf"
15 – "http://sfwater.org/Files/Statistics/LLMPApp%20Bpart_2.pdf"
16 - "http://bss.sfsu.edu/holzman/lakemerced/water.htm"

10. Appendix

Test Run Examples:

```
Waypoint 1
------------------------------
Lakebottom Depth = 4.1 meters


Waypoint 2
------------------------------
Lakebottom Depth = 4.1 meters


Waypoint 3
------------------------------
Lakebottom Depth = 4.1 meters


Waypoint 4
------------------------------
Lakebottom Depth = 4.2 meters


\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\


Waypoint 1
------------------------------
Lakebottom Depth = 4.2 meters


Waypoint 2
------------------------------
Lakebottom Depth = 4.2 meters


Waypoint 3
------------------------------
Lakebottom Depth = 4.2 meters


Waypoint 4
------------------------------
Lakebottom Depth = 4.3 meters
```

```
Waypoint 1
------------------------------
Sensor Depth = 2.18455 meters
Temperature = 19.55774 degrees C
pH = 9.31965 pH units
Dissolved Oxygen = 11.89111 mg/l
Conductivity = 0.51776 mS


Waypoint 2
------------------------------
Sensor Depth = 2.09320 meters
Temperature = 19.49988 degrees C
pH = 9.32011 pH units
Dissolved Oxygen = 11.89142 mg/l
Conductivity = 0.51667 mS


Waypoint 3
------------------------------
Sensor Depth = 2.13870 meters
Temperature = 19.46908 degrees C
pH = 9.32028 pH units
Dissolved Oxygen = 12.02866 mg/l
Conductivity = 0.51805 mS


Waypoint 4
------------------------------
Sensor Depth = 2.07160 meters
Temperature = 19.53631 degrees C
pH = 9.32753 pH units
Dissolved Oxygen = 12.03609 mg/l
Conductivity = 0.51494 mS


\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\


Waypoint 1
------------------------------
Sensor Depth = 2.24751 meters
Temperature = 20.01107 degrees C
pH = 8.89823 pH units
Dissolved Oxygen = 10.61682 mg/l
Conductivity = 0.53149 mS


Waypoint 2
------------------------------
Sensor Depth = 2.15318 meters
Temperature = 20.18652 degrees C
pH = 8.94813 pH units
Dissolved Oxygen = 10.66874 mg/l
Conductivity = 0.53187 mS


Waypoint 3
------------------------------
Sensor Depth = 2.14175 meters
Temperature = 20.15548 degrees C
pH = 8.97146 pH units
Dissolved Oxygen = 10.70356 mg/l
Conductivity = 0.53068 mS


Waypoint 4
------------------------------
Sensor Depth = 2.16265 meters
Temperature = 20.07570 degrees C
pH = 8.96583 pH units
Dissolved Oxygen = 10.76794 mg/l
Conductivity = 0.53322 mS
```

ATmega32 C Code:

```
/*****************************************************

Project : Autonomous Boat Sensor Code
Date    : Spring 2006 (Finished on 17May2006)
Author  : Bruce White
Company : San Francisco State University

Chip type           : ATmega32
Clock frequency     : 4.000000 MHz

 *****************************************************/

 #include <mega32.h>
 #include <mega32.h>
 #include <stdio.h>
 #include <string.h>
 #include <delay.h>

 char wakeup=0xFF;
 char plugin;
 char data[4];
 char ignore=0xFF;
 char dummy[6];
 char fake[4];
 char dpt[75];
 char cntstop=0;

 int counter=1;
 int counter2=1;

 float depth[20];
 float temp[20];
 float ph[20];
 float ldo[20];
 float cond[20];


 // PARITY CHECK-----------------------------------------------------------------------------------------------------

 #define RXB8 1
 #define TXB8 0
 #define UPE 2
 #define OVR 3
 #define FE 4
 #define UDRE 5
 #define RXC 7

 #define FRAMING_ERROR (1<<FE)
 #define PARITY_ERROR (1<<UPE)
 #define DATA_OVERRUN (1<<OVR)
 #define DATA_REGISTER_EMPTY (1<<UDRE)
 #define RX_COMPLETE (1<<RXC)

 // Get a character from the USART Receiver
 #ifndef _DEBUG_TERMINAL_IO_
 #define _ALTERNATE_GETCHAR_
 #pragma used+
 char getchar(void)
 {
 char status,data;
 while (1)
       {
       while (((status=UCSRA) & RX_COMPLETE)==0);
       data=UDR;
       if ((status & (FRAMING_ERROR | PARITY_ERROR | DATA_OVERRUN))==0)
         return data;
       };
 }
 #pragma used-
 #endif

 // Write a character to the USART Transmitter
 #ifndef _DEBUG_TERMINAL_IO_
 #define _ALTERNATE_PUTCHAR_
 #pragma used+
 void putchar(char c)
 {
 while ((UCSRA & DATA_REGISTER_EMPTY)==0);
 UDR=c;
 }
 #pragma used-
 #endif
```

```
83
84
85      // INTERRUPT-----------------------------------------------------------------------------------------------------------
86
87      interrupt [EXT_INT0] void ext_int0_isr(void)
88      {
89         UCSRC=0x86;                                                    // 4800 baud, 8 data bits, no parity, 1 stop bit
90         UBRRH=0x00;
91         UBRRL=0x33;
92
93         if(cntstop==0)
94         {
95         printf("No sensor readings taken!\n\r");
96         }
97         else
98         {
99              if(PINA.4==0)
100              {
101              counter=1;
102              while(counter<=cntstop)                                   // max 50 readings printed
103                {
104                printf("Waypoint %d\n\r", counter);                     // print waypoint number
105                printf("------------------------------\n\r");          // print separator
106                printf("Sensor Depth = %f meters\n\r", depth[counter]); // print depth value
107                printf("Temperature = %f degrees C\n\r", temp[counter]); // print temperature value
108                printf("pH = %f pH units\n\r", ph[counter]);            // print ph value
109                printf("Dissolved Oxygen = %f mg/l\n\r", ldo[counter]); // print dissolved oxygen value
110                printf("Conductivity = %f mS\n\r\n\r\n\r\n\r", cond[counter]); // print conductivity value
111                counter++;
112                }
113              }
114              else
115              {
116              counter=1;
117              counter2=1;
118              while(counter<=cntstop)                                   // max 500 readings printed
119                {
120                printf("Waypoint %d\n\r", counter);                     // print waypoint number
121                printf("------------------------------\n\r");          // print separator
122                printf("Lakebottom Depth = ");                          // print "Lakebottom Depth = X.X"
123                putchar(dpt[counter2]);
124                counter2++;
125                printf(".");
126                putchar(dpt[counter2]);
127                printf(" meters\n\r\n\r\n\r");
128                counter++;
129                counter2++;
130                }
131              counter=1;
132              counter2=1;
133              }
134        }
135     }
136
137
138     // INITIALIZATIONS-----------------------------------------------------------------------------------------------------
139
140     void main(void)
141     {
142     // Declare your local variables here
143
144     // Input/Output Ports initialization
145     // Port A initialization
146     // Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
147     // State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
148     PORTA=0x00;
149     DDRA=0x00;
150
151     // Port B initialization
152     // Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
153     // State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
154     PORTB=0x00;
155     DDRB=0x00;
156
157     // Port C initialization
158     // Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
159     // State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
160     PORTC=0x00;
161     DDRC=0x00;
162
163     // Port D initialization
164     // Func7=In Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
165     // State7=T State6=T State5=T State4=T State3=T State2=T State1=T State0=T
166     PORTD=0x00;
167     DDRD=0x00;
168
169     // Timer/Counter 0 initialization
170     // Clock source: System Clock
```

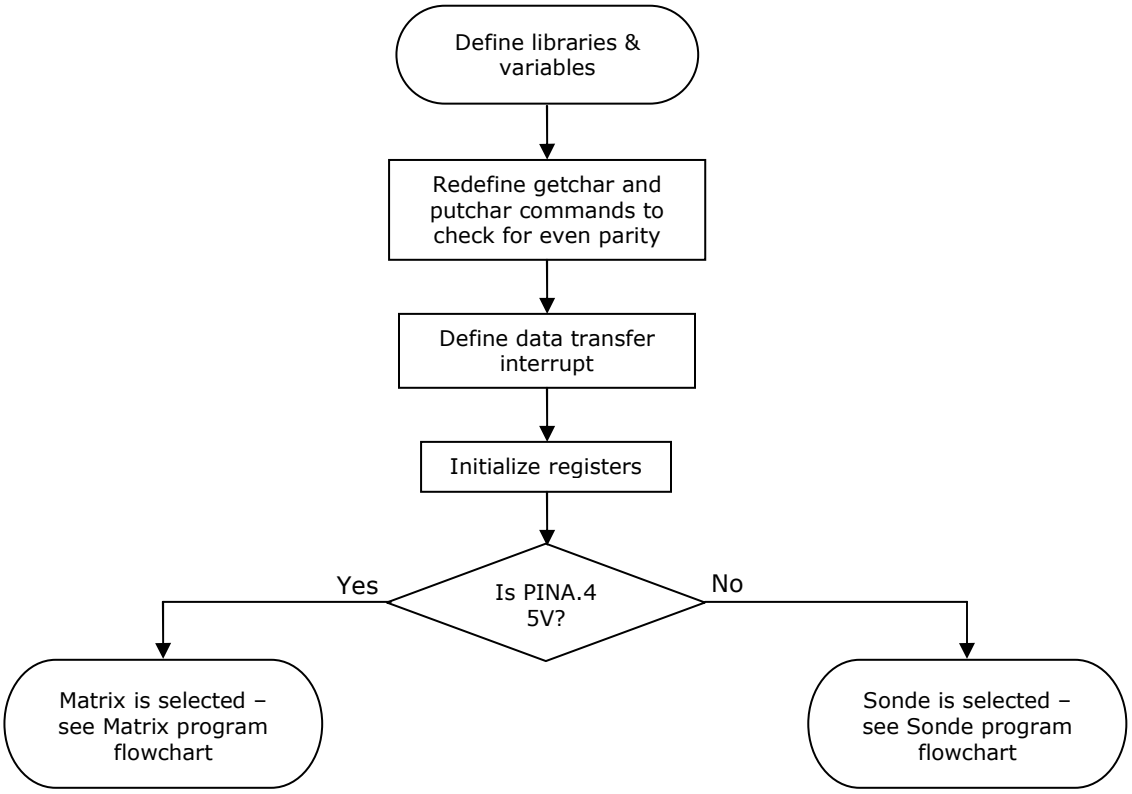
```
171     // Clock value: Timer 0 Stopped
172     // Mode: Normal top=FFh
173     // OC0 output: Disconnected
174     TCCR0=0x00;
175     TCNT0=0x00;
176     OCR0=0x00;
177
178     // Timer/Counter 1 initialization
179     // Clock source: System Clock
180     // Clock value: Timer 1 Stopped
181     // Mode: Normal top=FFFFh
182     // OC1A output: Discon.
183     // OC1B output: Discon.
184     // Noise Canceler: Off
185     // Input Capture on Falling Edge
186     // Timer 1 Overflow Interrupt: Off
187     // Input Capture Interrupt: Off
188     // Compare A Match Interrupt: Off
189     // Compare B Match Interrupt: Off
190     TCCR1A=0x00;
191     TCCR1B=0x00;
192     TCNT1H=0x00;
193     TCNT1L=0x00;
194     ICR1H=0x00;
195     ICR1L=0x00;
196     OCR1AH=0x00;
197     OCR1AL=0x00;
198     OCR1BH=0x00;
199     OCR1BL=0x00;
200
201     // Timer/Counter 2 initialization
202     // Clock source: System Clock
203     // Clock value: Timer 2 Stopped
204     // Mode: Normal top=FFh
205     // OC2 output: Disconnected
206     ASSR=0x00;
207     TCCR2=0x00;
208     TCNT2=0x00;
209     OCR2=0x00;
210
211     // External Interrupt(s) initialization
212     // INT0: On
213     // INT0 Mode: Falling Edge
214     // INT1: Off
215     // INT2: Off
216     GICR|=0x40;
217     MCUCR=0x02;
218     MCUCSR=0x00;
219     GIFR=0x40;
220
221     // Timer(s)/Counter(s) Interrupt(s) initialization
222     TIMSK=0x00;
223
224     // USART initialization
225     // USART Receiver: On
226     // USART Transmitter: On
227     // USART Mode: Asynchronous
228     UCSRA=0x00;
229     UCSRB=0x18;
230
231     // Analog Comparator initialization
232     // Analog Comparator: Off
233     // Analog Comparator Input Capture by Timer/Counter 1: Off
234     ACSR=0x80;
235     SFIOR=0x00;
236
237     // Global enable interrupts
238     #asm("sei")
239
240
241     while(1)
242       {
243
244
245     // SENSOR SELECTION-----------------------------------------------------------------------------------------------------
246
247     while(PINA.4==0)                                        // if PINA.6 is low, sonde is selected sensor
248       {                                                     // otherwise depth sensor selected
249        UCSRC=0xAE;                                          // 9600 baud, 8 data bits, even parity, 2 stop bits
250        UBRRH=0x00;
251        UBRRL=0x19;
252
253       while(plugin!=0xFB)                                   // look for sonde plugin string ending in FB
254       {
255           plugin=getchar();
256       }
257           delay_ms(1000);                                   // wait 1 second for sonde to go into modbus/even parity
mode
```

```
258
259         putchar(0x01);                                  // send modbus string to wake sonde up
260         putchar(0x03);
261         putchar(0x00);
262         putchar(0x30);
263         putchar(0x00);
264         putchar(0x02);
265         putchar(0xC4);
266         putchar(0x04);
267
268         delay_ms(1000);                                 // wait 1 second
269
270         putchar(0x01);                                  // send modbus string to wake sonde up
271         putchar(0x03);
272         putchar(0x00);
273         putchar(0x06);
274         putchar(0x00);
275         putchar(0x02);
276         putchar(0x24);
277         putchar(0x0A);
278
279         delay_ms(1000);                                 // wait 1 second
280
281         putchar(0x01);                                  // send modbus string to be sure sonde is awake
282         putchar(0x03);
283         putchar(0x00);
284         putchar(0x30);
285         putchar(0x00);
286         putchar(0x02);
287         putchar(0xC4);
288         putchar(0x04);
289
290
291     // WAIT FOR WAYPOINT-------------------------------------------------------------------------------------------
292
293       while(1)
294       {
295         while(PINA.5==0)                                // PINA.5 is guidance computer waypoint indicator
296         {
297           ;                                             // do nothing when not at a waypoint
298         }
299         while(PINA.5!=0)                                // when PINA.5 is 5V, waypoint reached
300         {
301           ;                                             // do nothing until PINA.5 returns to 0
302         }
303
304
305     // DEPTH READING-----------------------------------------------------------------------------------------------
306
307         putchar(0x01);                                  // query for depth reading
308         putchar(0x03);
309         putchar(0x00);
310         putchar(0x30);
311         putchar(0x00);
312         putchar(0x02);
313         putchar(0xC4);
314         putchar(0x04);
315
316         while(ignore!=0x04)
317         {
318           ignore=getchar();                             // ignore first three bytes of response
319         }
320         fake[0]=getchar();                              // last four bytes are relevant data
321         fake[1]=getchar();
322         fake[2]=getchar();
323         fake[3]=getchar();
324         ignore=getchar();
325         ignore=getchar();
326
327         data[3]=fake[2];                                // number rearrangement for float calculation
328         data[2]=fake[3];
329         data[1]=fake[0];
330         data[0]=fake[1];
331
332         memcpy(&depth[counter], data, sizeof data);     // copy four chars into a single floating point number
333         delay_ms(1000);
334
335
336     // TEMPERATURE READING-----------------------------------------------------------------------------------------
337
338         putchar(0x01);                                  // query for temperature reading
339         putchar(0x03);
340         putchar(0x00);
341         putchar(0x00);
342         putchar(0x00);
343         putchar(0x02);
344         putchar(0xC4);
345         putchar(0x0B);
```

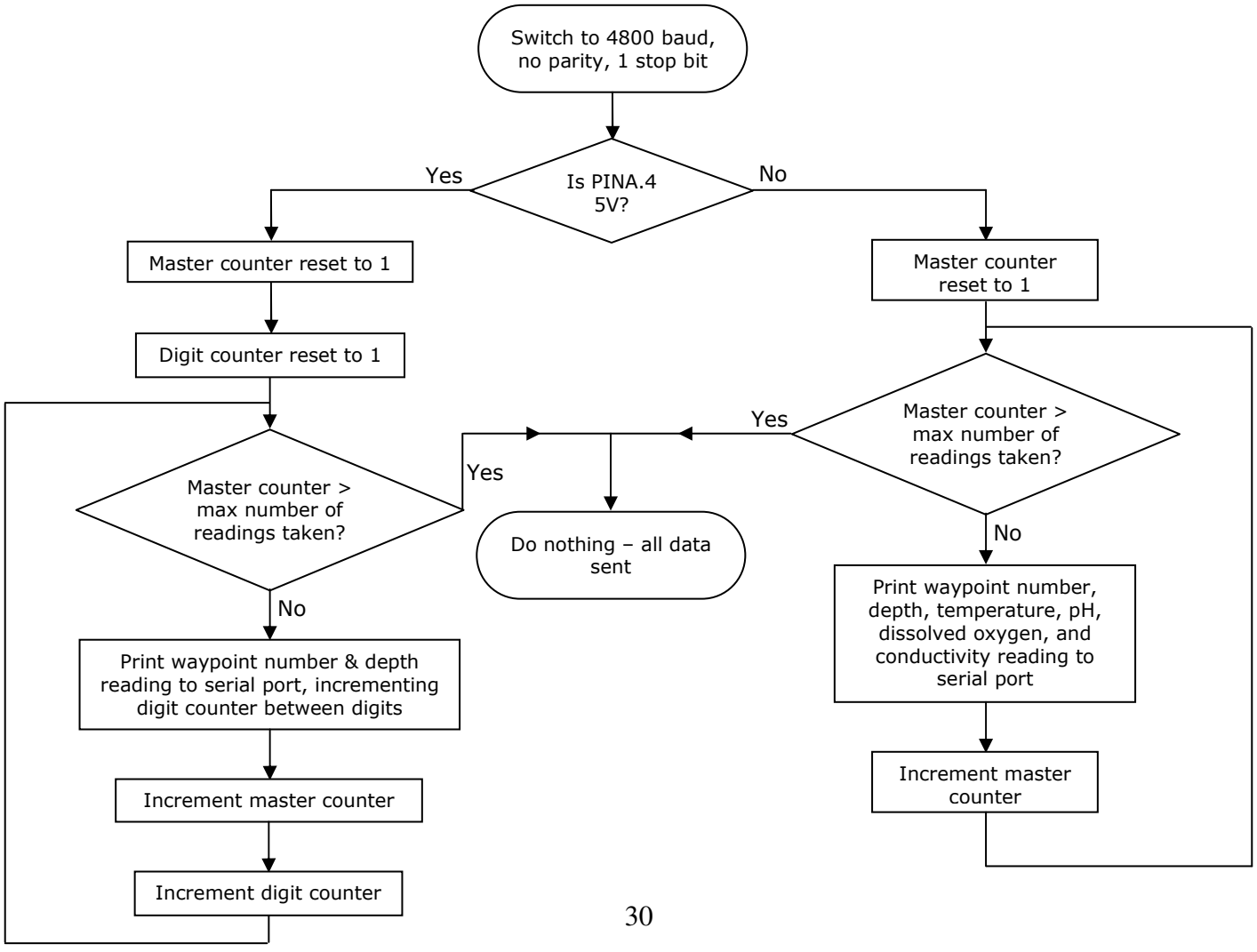
```
346
347         while(ignore!=0x04)
348         {
349            ignore=getchar();                          // ignore first three bytes of response
350         }
351         fake[0]=getchar();                            // last four bytes are relevant data
352         fake[1]=getchar();
353         fake[2]=getchar();
354         fake[3]=getchar();
355         ignore=getchar();
356         ignore=getchar();
357
358         data[3]=fake[2];                              // number rearrangement for float calculation
359         data[2]=fake[3];
360         data[1]=fake[0];
361         data[0]=fake[1];
362
363         memcpy(&temp[counter], data, sizeof data);    // copy four chars into a single floating point number
364         delay_ms(1000);
365
366
367      // PH READING-------------------------------------------------------------------------------------------------
368
369         putchar(0x01);                                // query for ph reading
370         putchar(0x03);
371         putchar(0x00);
372         putchar(0x06);
373         putchar(0x00);
374         putchar(0x02);
375         putchar(0x24);
376         putchar(0x0A);
377
378         while(ignore!=0x04)
379         {
380            ignore=getchar();                          // ignore first three bytes of response
381         }
382         fake[0]=getchar();                            // last four bytes are relevant data
383         fake[1]=getchar();
384         fake[2]=getchar();
385         fake[3]=getchar();
386         ignore=getchar();
387         ignore=getchar();
388
389         data[3]=fake[2];                              // number rearrangement for float calculation
390         data[2]=fake[3];
391         data[1]=fake[0];
392         data[0]=fake[1];
393
394         memcpy(&ph[counter], data, sizeof data);      // copy four chars into a single floating point number
395         delay_ms(1000);
396
397
398      // DO READING-------------------------------------------------------------------------------------------------
399
400         putchar(0x01);                                // query for dissolved oxygen reading
401         putchar(0x03);
402         putchar(0x00);
403         putchar(0x16);
404         putchar(0x00);
405         putchar(0x02);
406         putchar(0x25);
407         putchar(0xCF);
408
409         while(ignore!=0x04)
410         {
411            ignore=getchar();                          // ignore first three bytes of response
412         }
413         fake[0]=getchar();                            // last four bytes are relevant data
414         fake[1]=getchar();
415         fake[2]=getchar();
416         fake[3]=getchar();
417         ignore=getchar();
418         ignore=getchar();
419
420         data[3]=fake[2];                              // number rearrangement for float calculation
421         data[2]=fake[3];
422         data[1]=fake[0];
423         data[0]=fake[1];
424
425         memcpy(&ldo[counter], data, sizeof data);     // copy four chars into a single floating point number
426         delay_ms(1000);
427
428
429      // CONDUCTIVITY READING---------------------------------------------------------------------------------------
430
431         putchar(0x01);                                // query for conductivity reading
432         putchar(0x03);
433         putchar(0x00);
```

```
434           putchar(0x0A);
435           putchar(0x00);
436           putchar(0x02);
437           putchar(0xE4);
438           putchar(0x09);
439
440           while(ignore!=0x04)
441           {
442              ignore=getchar();                          // ignore first three bytes of response
443           }
444           fake[0]=getchar();                            // last four bytes are relevant data
445           fake[1]=getchar();
446           fake[2]=getchar();
447           fake[3]=getchar();
448           ignore=getchar();
449           ignore=getchar();
450
451           data[3]=fake[2];                              // number rearrangement for float calculation
452           data[2]=fake[3];
453           data[1]=fake[0];
454           data[0]=fake[1];
455
456           memcpy(&cond[counter], data, sizeof data);    // copy four chars into a single floating point number
457
458
459     // ADVANCE COUNTER & RESET FOR NEXT WAYPOINT-----------------------------------------------------------------------
460
461           cntstop=counter;
462           counter++;                                    // advance counter for next waypoint
463           while(cntstop>=50)
464           {
465              ;                                          // prevent more than 50 readings
466           }
467        }                                                // end while loop inside water sensor loop
468       }                                                 // end water sensor loop
469
470
471     // DEPTH SENSOR---------------------------------------------------------------------------------------------------
472
473        while(PINA.4!=0)                                 // depth sensor selected
474        {
475          UCSRC=0x86;                                   // 4800 baud, 8 data bits, no parity, 1 stop bit
476          UBRRH=0x00;
477          UBRRL=0x33;
478
479
480     // WAIT FOR WAYPOINT----------------------------------------------------------------------------------------------
481
482           while(PINA.5==0)                              // PINA.5 is guidance computer waypoint indicator
483           {
484             ;                                          // do nothing when not at a waypoint
485           }
486           while(PINA.5!=0)                              // when PINA.5 is 5V, waypoint reached
487           {
488             ;                                          // do nothing until PINA.5 returns to 0
489           }
490
491
492     // DEPTH READING--------------------------------------------------------------------------------------------------
493
494           while(dummy[0]!=0x50)                         // look for letter P at serial port
495           {
496             dummy[0]=getchar();
497           }
498           dummy[1]=getchar();                           // ignore letter T
499           dummy[2]=getchar();                           // ignore comma
500           dummy[3]=getchar();                           // store first digit of depth reading
501           dummy[4]=getchar();                           // ignore period
502           dummy[5]=getchar();                           // store second digit of depth reading
503           dpt[counter2]=dummy[3];                       // rearrange depth reading numbers for later output
504           counter2++;
505           dpt[counter2]=dummy[5];
506           counter2++;
507           dummy[0]=0;
508           cntstop=counter;
509           counter++;
510           while(cntstop>=500)
511           {
512              ;                                          // prevent more than 500 readings
513           }
514        }                                                // end depth sensor loop
515
516     }                                                   // end program while loop
517   }                                                     // end main loop
518
```

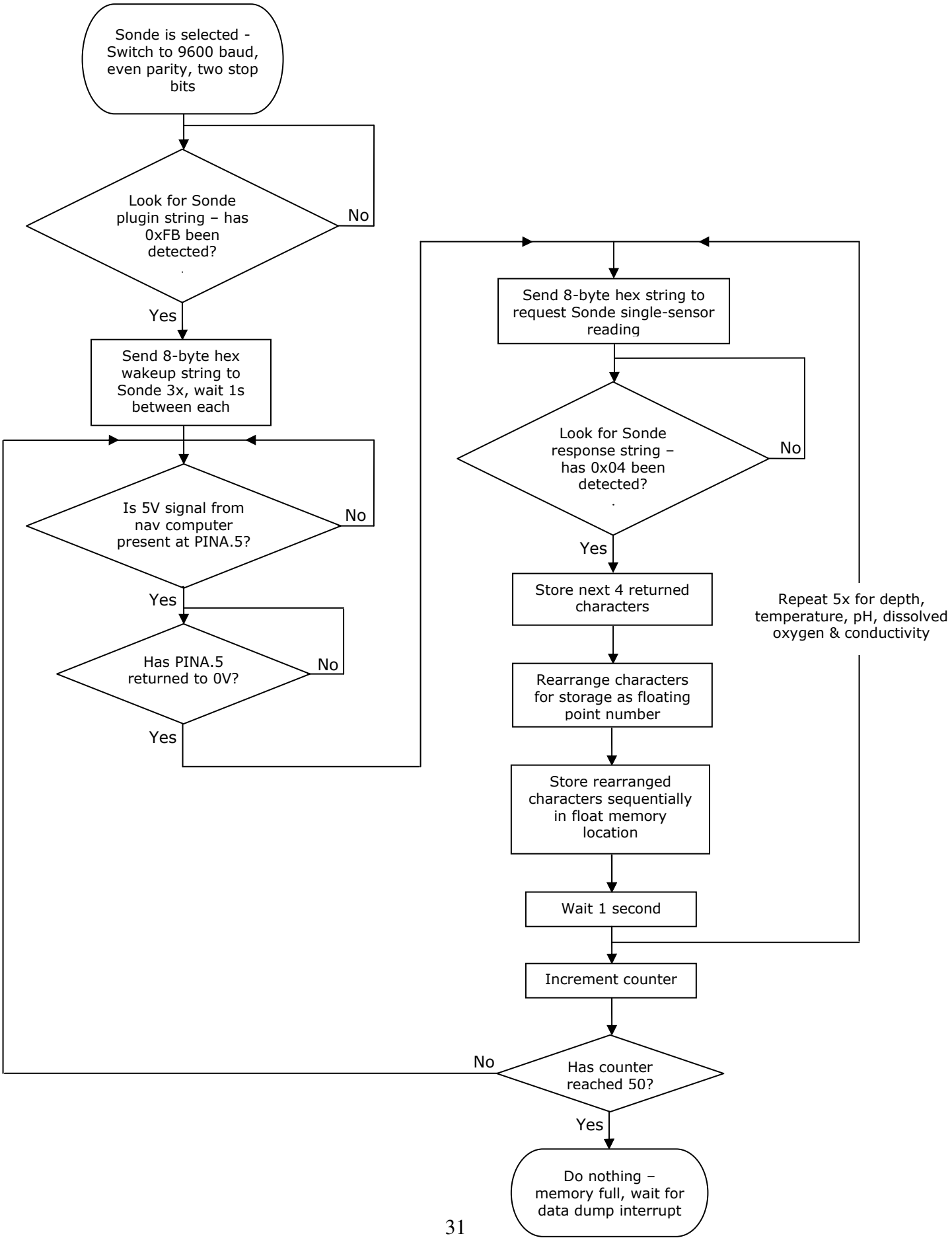
Initialization Program Flowchart:



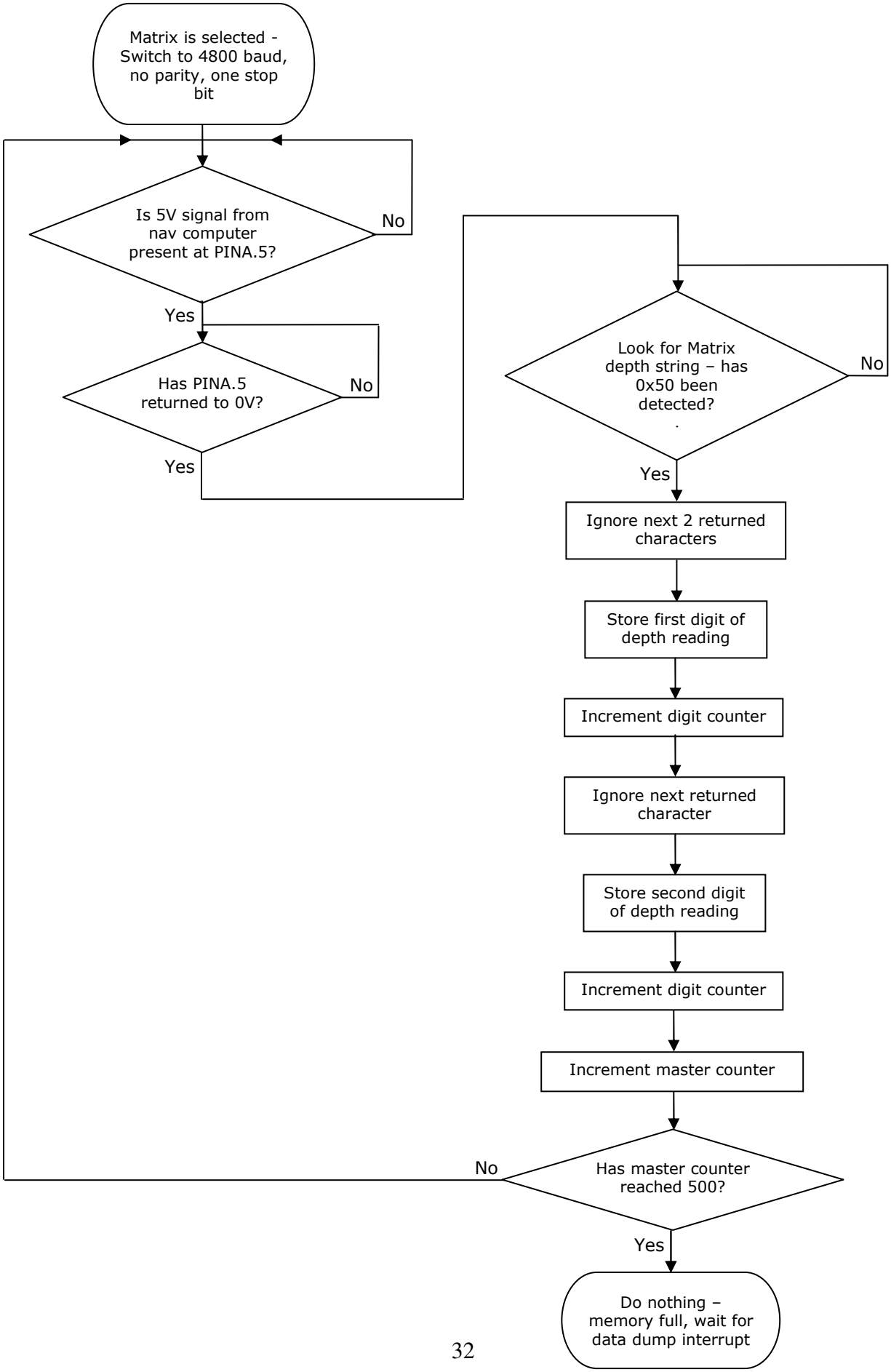
Interrupt Program Flowchart:



Sonde Program Flowchart:



Matrix Program Flowchart:



DIGITAL SENSOR INTEGRATION MODULE

Bruce White

07-May-06