

2014

DATA COLLECTION AND TELEMETRY SYSTEM DESIGN FOR MARINE INSTRUMENTATION: THE SON-O-MERMAID

Donaldo Guevara
University of Rhode Island, donaldog@gmail.com

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Terms of Use

All rights reserved under copyright.

Recommended Citation

Guevara, Donaldo, "DATA COLLECTION AND TELEMETRY SYSTEM DESIGN FOR MARINE INSTRUMENTATION: THE SON-O-MERMAID" (2014). *Open Access Master's Theses*. Paper 359.
<https://digitalcommons.uri.edu/theses/359>

This Thesis is brought to you by the University of Rhode Island. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu. For permission to reuse copyrighted content, contact the author directly.

DATA COLLECTION AND TELEMETRY SYSTEM DESIGN FOR MARINE
INSTRUMENTATION: THE SON-O-MERMAID

BY

DONALDO GUEVARA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTERS OF SCIENCE

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2014

MASTER OF SCIENCE THESIS

OF

DONALDO GUEVARA

APPROVED:

Thesis Committee:

Major Professor	Godi Fischer
	Harold Vincent
	Jien-Chung Lo
	Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2014

ABSTRACT

This Thesis presents a prototype design for Sonobuoy Mobile Earthquake Recorder in Marine Areas by Independent Divers (Son-O-MERMAID), a floating instrument that acts as a freely drifting seismometer that captures acoustic signals caused by distant seismic activity. A first version of Son-O-MERMAID was built and deployed in October of 2012, but due to Hurricane Sandy the device was destroyed and data lost. A major limitation of this device was the fact that acoustic data was collected and stored in a submerged computer at a depth of 2,000 feet and data was only accessible once the device was pulled out of the water at test completion. A new prototype system, version 2 is described, constructed and successfully tested which provides additional features not offered by the previous version. These additional features consist of providing an effective algorithm to transfer acoustic data from the submerged computer to the surface in real time to be stored there for future analysis. Additionally, a stratum one NTP server has been implemented with time synchronized to GPS so acoustic data can be time stamped. Testing demonstrates that data is transferred in its entirety and in real time to the surface unit which will subsequently enable the transfer of data to a land-based station via IRIDIUM for real time analysis.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my major professor Dr. Godi Fischer for the continuous support and patience during this investigation project. My special thanks go also to Dr. Harold Vincent, my advisor. His knowledge and guidance were vital for the conduction of this study. I am also thankful for the support received from URI personnel: Andrew Duane, Hayden Radke, Cathy Cipolla and Gary Savoie.

Many thanks to my employer, the Naval Undersea Warfare Center, for providing the economic support and allowing me the necessary time to conduct this investigation. Thanks also to Dr. Richard Katz for his guidance and meaningful feedback.

Last but not least, I thank my family for the unlimited support and patience during my long hours of absence from home. With their support it was possible to reach my goals.

Table of Contents

ABSTRACT	ii
ACKNOWLEDGEMENTS	iii
Table of Contents	iv
List of Tables	vi
List of Figures	vi
1 INTRODUCTION	1
2 REVIEW OF LITERATURE	5
2.1 THE NETWORK TIME PROTOCOL: FEATURES AND ALGORITHMS	5
2.1.1 NETWORK TIME PROTOCOL DAEMON	5
2.1.2 NETWORK TIME PROTOCOL (NTP)	5
2.1.2.1 BASIC FEATURES OF NTP	6
2.1.2.2 NEW FEATURES OF NTP V4	7
2.1.2.3 HOW NTP WORKS	8
2.1.2.4 NTP TIMESCALE AND DATA FORMANTS	8
2.1.2.5 ARCHITECTURE AND ALGORITHMS	11
2.1.2.5.1 CLOCK FILTER ALGORITHM	13
2.1.2.5.2 CLOCK SELECT ALGORITHM	15
2.1.2.5.3 CLOCK CLUSTER ALGORITHM	16
2.1.2.5.4 CLOCK DISCIPLINE ALGORITHM (NTP V4)	18
2.1.2.5.5 NTP POLL PROCESS	20
2.1.2.5.6 NTP CLOCK STATE MACHINE	22
2.2 THE GLOBAL POSITIONING SYSTEM DAEMON (GPSD)	24
2.3 PULSE PER SECOND (PPS)	24
2.4 SYNCHRONIZATION OF THE NTP SERVER WITH GPS AND PPS	25
2.4.1 THE GPS DEVICE	25
2.4.2 NTPD REFERENCE CLOCKS	26
2.4.3 SHM REFERENCE CLOCK	28
3 METHODOLOGY	30
3.1 Son-O-MERMAID SYSTEM DESIGN	30
3.1.1 THE SUBMERGED COMPONENT	31
3.1.1.1 3-HYDROPHONE ARRAY	31

3.1.1.2	HYDROPHONES AND ADC INTERFACE BOARD	33
3.1.1.3	ANALOG TO DIGITAL CONVERTER (ADC)	35
3.1.1.4	USB TO RS-485 ADAPTER (Model: xs885)	37
3.1.1.5	THE CENTRAL PROCESSING UNIT (CPU)	38
3.1.1.6	CONFIGURATION OF THE SUBMERGED COMPONENT	39
3.1.1.6.1	PHIDGET SBC2 SOFTWARE CONFIGURATION	41
3.1.2	PART 2 –SURFACE COMPONENT	44
3.1.2.1	SINGLE BOARD COMPUTER (SBC) SELECTION AND GPS TIME SYNCHRONIZATION OF Son-O-MERMAID SURFACE COMPONENT	44
3.1.2.1.1	SYNCHRONIZATION OF A COMPUTER’S TIME TO A GPS RECEIVER –A PROOF-OF- CONCEPT	46
3.1.2.1.2	COMPUTER SELECTION FOR THE SURFACE UNIT	48
3.1.2.2	SELECTION OF A GPS RECEIVER TO BE USED AS TIME SOURCE FOR GPS TIME SYNCHRONIZATION	50
3.1.2.3	TIME SYNCHRONIZATION OF RASPBERRY PI MODEL B TO ADAFRUIT ULTIMATE GPS BREAKOUT BOARD Rev 3	53
3.1.2.3.1	APPROACH ONE: SYNCHRONIZATION OF NTP USING KERNEL DRIVERS.....	54
3.1.2.3.2	APPROACH TWO: SYNCHRONIZATION OF NTP WERVER WITH GPSD AND PPS ...	60
3.1.2.3.3	VERIFY PERFORMANCE OF TIME SYNCHRONIZATION OPERATIONS	66
3.1.3	INTERFACE BETWEEN PART 1 AND PART 2	71
3.2	PROTOCOLS AND MEDIA FOR DATA TRANSMISSION IMPLEMENTATION	71
3.2.1	10BASE-T (Twisted Pair Ethernet)	71
3.2.2	100BASE-FX ETHERNET OVER FIBER	72
3.2.3	RS-485 PROTOCOL –A SUCCESSFUL AND FINAL APPROACH.....	73
3.3	DATA TELEMETRY	74
3.3.1	APPROACH ONE: COMPLETE FILE TRANSMISSION ALGORITHM	75
3.3.2	APPROACH TWO: SAMPLE BY SAMPLE TRANSMISSION ALGORITHM	78
3.4	POWER CONSUMPTION OF Son-O-MERMAID AND STORAGE SPACE REQUIRED	82
3.4.1	POWER CONSUMPTION BY THE SUBMERGED UNIT	82
3.4.2	POWER CONSUMPTION BY THE SURFACE UNIT	83
3.4.3	STORAGE SPACE NEEDED	83
3.4.3.1	COMPLETE FILE TRANSMISSION	83
3.4.3.2	SAMPLE BY SAMPLE TRANSMISSION	84

4	FINDINGS.....	85
4.1	HARDWARE REQUIRED FOR BUILDING Son-O-MERMAID PROTOTYPE	85
4.1.1	SUBMERGED COMPONENT.....	85
4.1.2	SURFACE COMPONENT	86
4.1.3	WIRING BETWEEN SUBMERGED AND SURFACE COMPONENTS	86
4.2	SOFTWARE SOLUTION FOR DATA TELEMETRY OF Son-O-MERMAID	87
4.3	GPS TIME AND DATA SYNCHRONIZATION ACROSS SEVERAL Son-O-MERMAID SYSTEMS.....	87
5	CONCLUSION.....	90
APPENDIX	92

List of Tables

Table 1: Hydrophones Specifications	109
Table 2: ADC specifications.....	109
Table 3: Fit PC2i specifications.....	110
Table 4: Phidget SBC2 product specifications.....	111
Table 5: Raspberry Pi Model B specifications.....	111
Table 6: USB to RS-485 Adapter specifications.....	112
Table 7: Ultimate GPS Breakout version 3 features	112
Table 8: RS-485 Wiring	113

List of Figures

Figure 1-1: Son-O-MERMAID as defined by Harold Vincent and Frederik Simons.....	3
Figure 2-1: NTP Data Formats.....	10
Figure 2-2: NTP Daemon Processes and Algorithms.....	11
Figure 2-3: NTP ARCHITECTURE [6].....	13
Figure 2-4: Cluster Algorithm [7]	18
Figure 2-5: Clock Discipline Algorithm	19
Figure 2-6: FLL/PLL Prediction Functions.....	20
Figure 2-7: Clock State Machine	23
Figure 2-8: Garmin GPS 18x LVC to RS232 and USB wiring	27
Figure 3-1: Architecture of Son-O-MERMAID	30
Figure 3-2: Submerged component as deployed in 2012, on the first implementation of Son-O-MERMAID.....	31
Figure 3-3: Hydrophone array for Son-O-MERMAID.	32
Figure 3-4a: Graphic description of array and telemetry simulation	32
Figure 3-4b: Test bed development of Son-O-Mermaid with Array Simulation.	33

Figure 3-5a: Hydrophones & ADC Interface board	34
Figure 3-5b: Hydrophones & ADC Interface circuit.	34
Figure 3-6: DAQFlex Framework.....	35
Figure 3-7: USB-7202 16-bit Analog to Digital Converter	37
Figure 3-8: USB to RS-485/RS-422 converter	38
Figure 3-9: Phidget sbc2 version 1072_0.....	39
Figure 3-10: Configuration of Son-O-MERMAID submerged component	40
Figure 3-11: Data collected at pre-launch test of Son-O-Mermaid prototype one in October, 2012.	41
Figure 3-12: Son-O-MERMAID submerged unit simulation.....	42
Figure 3-13: A triangular wave sampled by the ADC board, and stored in the submerged unit into one-minute files.	42
Figure 3-14: Zooming into Figure 3-13 to verify a triangular wave was received.	43
Figure 3-15: Zooming into the figure 3-14 to verify that the three inputs (channels) are received.	43
Figure 3-16: Wiring of a Garmin 18x LVC GPS	46
Figure 3-17: Raspberry Pi model B.....	50
Figure 3-18: Garmin 18x LVC GPS with USB and RS232 connector	51
Figure 3-19: Garmin 19x HVS GPS.....	52
Figure 3-20: Ultimate GPS Breakout Version 3.....	53
Figure 3-21: Wiring of GPS antenna (left) to a Raspberry Pi for GPS time synchronization....	57
Figure 3-22: Adding rules to “udev” so NTP driver would process the correct data.....	59
Figure 3-23: Step by step instructions to install NTP version 4.2.7p319 with specific set of flags.....	59
Figure 3-24: Partial output of the “ntpq -p” command to monitor NTP time synchronization status.	66
Figure 3-25: Output of the “ntpq -p” command to monitor NTP time synchronization status.	68
Figure 3-26: Two SBC boards connected via “USB to RS-485” converters at both ends of a 2,000 feet long CAT5 Ethernet cable.....	74
Figure 3-27: Test bed set up for telemetry approaches development.....	75
Figure 3-28: Three channels of data contained on each data file	77
Figure 3-29: Two one-minute files concatenated to demonstrate data continuity between files.....	77
Figure 3-30: Zooming into Figure 3-29 shows no data between files boundary.....	78
Figure 3-31: Sending Algorithm Flowchart	80
Figure 3-32: Receiving Algorithm Flowchart.....	81
Figure 4-1: Test setup of a simulation of two Son-O-Mermaid systems.	87
Figure 4-2: Simulation of two Son-O-Mermaid systems both with time synchronized to GPS.	88

1 INTRODUCTION

Son-O-MERMAID is a concept that evolved from MERMAID [1], a floating instrument that acts as a freely drifting seismometer equipped with a hydrophone that captures acoustic signals caused by distant seismic activity. MERMAID is equipped with sensitive on-board acoustics, a battery life measured in months, and with the latest in seismic event detection and discrimination technology. It floats at depth, but it surfaces upon event detection to acquire its GPS position and relay the seismic data via the IRIDIUM satellite constellation. MERMAID was developed by Guust Nolet, formerly from Princeton University and now at the University of Nice.

The Son-O-MERMAID instrument is a next generation drifting prototype, jointly developed at the University of Rhode Island (URI) by Harold Vincent and at Princeton University by Frederik Simons, that combines a surface buoy with instruments dangling from an untethered cable. The surface unit enables the GPS and IRIDIUM capabilities to be always engaged. The submerged portion of the device consists of a vertical array of three hydrophones and electronics located at a depth of ~750 meters. The purpose for the vertical array of hydrophones is to separate non-propagating noise from seismic arrivals with removal of surface reverberations [2]. Figure 1-1 below, displays a complete view of Son-O-MERMAID as envisioned by its inventors. In 2012, a first version of a Son-O-MERMAID was built, and it had been designed to record acoustic data and store it in the submerged unit, keeping at the surface only the GPS and IRIDIUM communications modules to report its position and time to a land-based station. Data could be analyzed only after the device is pulled out of the water at the conclusion of a planned test. The greatest risks of this design

were: one, a wiring disconnect between the submerged and surface units due to a storm or other unpredicted event; two, to experience a data failure in the submerged unit with no means to be detected before the system is pulled out of the water at the conclusion of the test event.

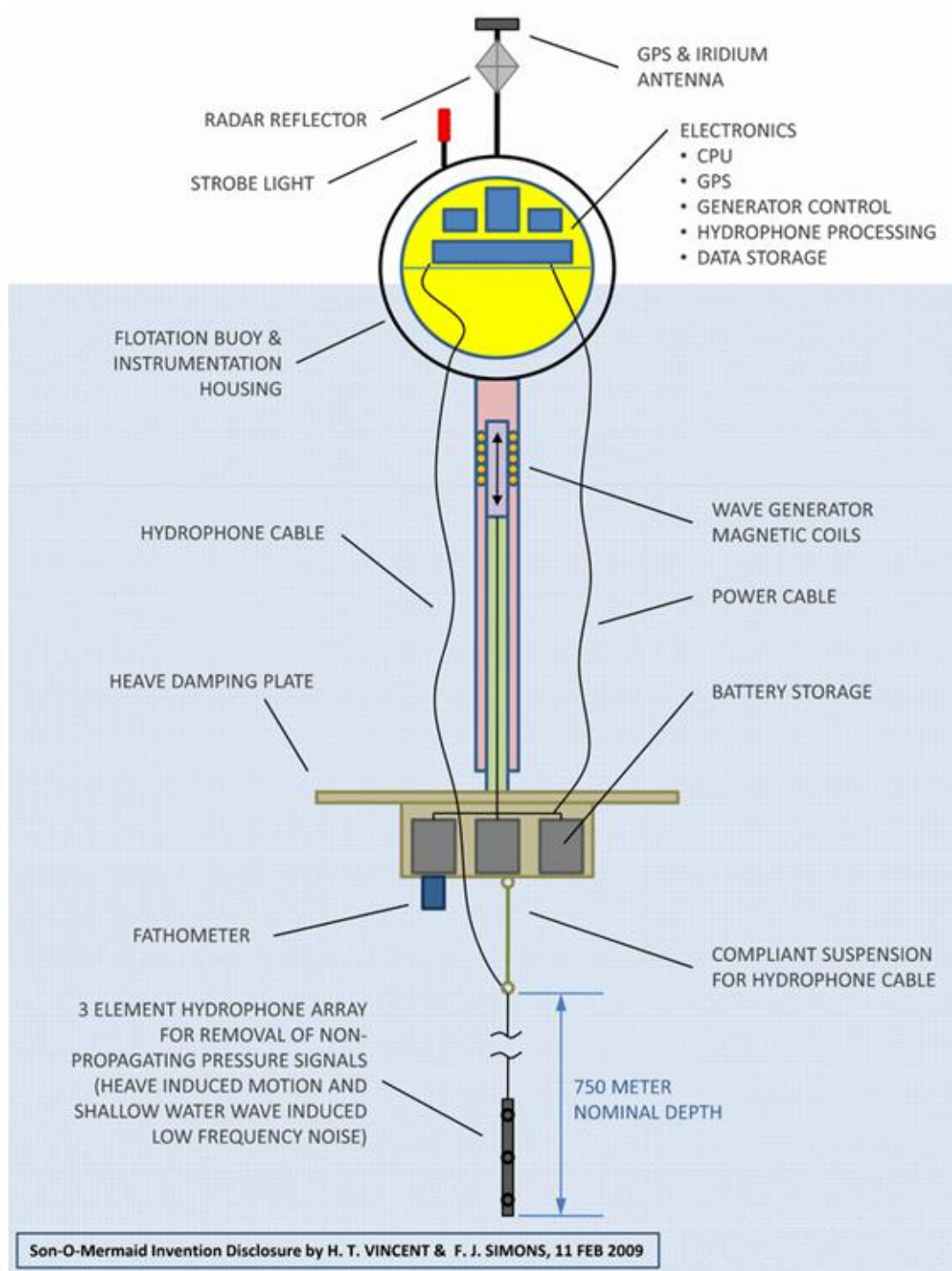


Figure 1-1: Son-O-MERMAID as defined by Harold Vincent and Frederik Simons.

The objectives of this project are as follows: first, select the necessary hardware components based on cost and power efficiency for prototype implementation of Son-O-MERMAID. Second, design a telemetry algorithm that will reliably transfer the acoustic data from the submerged unit to a computer at the surface in real time which will additionally enable data transmission via IRIDIUM communications system to a land-based station for real time data analysis. Third, synchronize the system time of the surface unit to a GPS receiver to provide data timestamp within one millisecond accuracy. Four, design, build and test the prototype.

2 REVIEW OF LITERATURE

2.1 THE NETWORK TIME PROTOCOL: FEATURES AND ALGORITHMS

2.1.1 NETWORK TIME PROTOCOL DAEMON

The ntpd program is an operating system daemon that synchronizes the system clock to remote NTP time servers or local reference clocks. It is a complete implementation of NTP version 4 defined by RFC-5905, but also retains compatibility with version 3 defined by RFC-1305 and versions 1 and 2, defined by RFC-1059 and RFC-1119, respectively. The program can operate in any of several modes, including client/server, symmetric and broadcast modes, and with both symmetric-key and public key-cryptography [3]. Ordinarily, the ntpd program requires a configuration file which contains configuration commands described on the previous cited documentation. This is all described in detail in section 3.1.2.3.2 under the NTP update configuration file. Clients can also discover remote servers and configure them automatically without previous configuration details.

The ntpd program normally operates continuously while adjusting the system time and frequency; however, the user can control and decide how the ntpd should work by selecting the desired command line options. The next section presents a full description of NTP, what it is and how it works.

2.1.2 NETWORK TIME PROTOCOL (NTP)

The Network Time Protocol (NTP) is an Internet protocol used to synchronize the clocks of computers to a time reference. This standard protocol was developed by Professor David L. Mills at the University of Delaware. Time synchronization across a network is very

important if communication programs are running on different computers. If the time is not synchronized, from the perspective of an external observer, switching between these systems would cause time to jump forward and back, a non-desirable effect. As a consequence, isolated networks may run under their own wrong time, but effects will be visible as soon as a connection to the internet is established. Using available technology with existing workstations and Internet paths, it has been demonstrated that computers can be reliably synchronized to better than a millisecond in LANs and better than a few tens of milliseconds in most places in the global Internet [4]. The majority if not all references used in this section are to professor Miller's work; he created the NTP protocol two decades ago and today he continues his investigation to improve its performance.

2.1.2.1 BASIC FEATURES OF NTP

- a. NTP needs a reference clock that defines the true time to operate. All clocks in the network will be set towards that true time.
- b. NTP uses Universal Time Coordinated (UTC) as reference time. UTC is an official standard for the current time which evolved from the former Greenwich Mean Time (GMT). This time is independent from time zones and is based on a quantum resonance of a cesium atom, being more accurate than GMT which is based on mean solar time.
- c. NTP is a fault-tolerant protocol that will automatically select the best of several available time sources to synchronize to. Insane time sources will be detected and avoided.

- d. NTP is highly scalable synchronization network where nodes exchange time information. The time information from one node to another form a hierarchical graph with reference clocks at the top.
- e. NTP selects the best candidates for its time out of many available sources. It uses a highly accurate protocol with a resolution of less than a nanosecond.
- f. When a network connection is temporarily unavailable, NTP uses measurements from the past to estimate current time.
- g. NTP works on most popular UNIX Operating Systems and Windows. As of December of 2013 there are two versions of NTP available: version 3 is the official Internet standard and version 4 is the current development version with specification RFC 5905, which describes NTP specifics and summarizes information useful for its implementation. In addition, some vendors of operating systems customize and deliver their own versions of NTP. For MERMAID NTP version 4 was used and its installation and configuration was customized to efficiently make it run on the Raspberry Pi and synchronize with a GPS receiver as its time source.

2.1.2.2 NEW FEATURES OF NTP V4

According to the NTP v4 release notes, the new features of version four as compared to version three are:

- a. Use of floating point arithmetic instead of fix-point (integer arithmetic).
- b. Redesigned clock discipline algorithm that improves accuracy, handling of network jitter and polling intervals.
- c. Support for nanokernel kernel implementation that provides nanosecond precision.
- d. Public-Key cryptography known as autokey that avoids having common secret keys.

- e. Automatic server discovery (multicast mode).
- f. Fast synchronization at startup and after network failures (burst mode).
- g. New and revised drivers for reference clocks.
- h. Support for new platforms and operating systems.

2.1.2.3 HOW NTP WORKS

NTP time synchronization services are widely available in the public Internet with several thousand servers distributed in most countries. The NTP subnet operates with a hierarchy of levels where each level is assigned a number called the “stratum”. Stratum 1 (primary) servers are at the lowest level and directly synchronized to national time services via satellite, radio or telephone modem. Stratum 2 (secondary) servers are at the next higher level synchronized to stratum 1 servers and so on. Clients, on the other hand, in order to provide the most accurate, reliable service, typically operate with several redundant servers over diverse network paths.

2.1.2.4 NTP TIMESCALE AND DATA FORMANTS

NTP clients and servers synchronize to the UTC timescale used by national laboratories and disseminated by radio, satellite and telephone modem; corrections for time zone of daylight savings are performed by the operating system. This time scale is determined by the rotation of the Earth about its axis, and since Earth rotation is gradually slowing down relative to International Atomic Time (TAI), in order to correct UTC with respect to TAI a leap second is inserted at intervals of about 18 months, as determined by the International Earth Rotation Services (IERS). There are three approaches to implementing a leap second in NTP. The first approach is to increment the system clock during the leap

second and continue incrementing following the leap. One problem with this approach is that conversion to UTC requires knowledge of all past leap seconds and epoch of insertion. A second approach is to increment the system clock during the leap second and step the clock backward one second at the end of the leap second. The problem is that the resulting timescale is discontinuous and a reading during the leap is repeated one second later. The third approach is to freeze the clock during the leap second allowing the time to catch up at the end of the leap second; this is the approach taken by the NTP conventions. Leap second warnings are disseminated by the national laboratories in the broadcast time-code format, and these warnings are propagated from the NTP primary servers via other servers to the clients by the NTP on-wire protocol. The leap second is implemented by the operating system kernel. About every eighteen months the International Earth Rotation Service (IERS) issues a bulletin announcing the insertion of a leap second in the UTC timescale. This normally happens at the end of the last day of June or December and even though this bulletin is available on the Internet at "www.iers.org", advance notice of leap seconds is given in signals broadcast from national time and frequency stations, in GPS signals and in telephone modem services. Many but not all reference clocks recognize these signals and many but not all drivers can decode the signals and set the leap bits in the time-code accordingly. This means that many but not all primary servers can pass on these bits in the NTP packet heard to dependent secondary servers and clients. Secondary servers will pass these bits to their dependents and so on throughout the NTP subnet. When no means are available to determine the leap bits from a reference clock or downstratum server, a leapseconds file can be downloaded from "time.nist.gov" and installed. If the precision time kernel support is available and enabled at the beginning of the day of the leap event, the leap bits are set by the Unix "`ntp_adjtime ()`" system call to arm the kernel for the leap at the

end of the day, then the kernel will automatically insert one second exactly at the time of the leap, after which the leap bits will be turned off. If the kernel support is not available or disabled, the leap is implemented by setting the clock back one second using the Unix “settimeofday ()” system call, which will repeat the last second. However setting the time backwards by one second does not actually set the system clock backwards, but effectively stalls the clock for one second.

There are two time formats used by NTP, a 64-bit timestamp format and a 128-bit “datestamp” format. The “datestamp” format is used internally, while the timestamp format is used in packet headers exchanged between clients and servers. These time formats are shown in figure 2-1 below [5].The timestamp format spans 136 years, called an era. The current NTP era began on 1 January 1900, and the next one will begin in 2036.

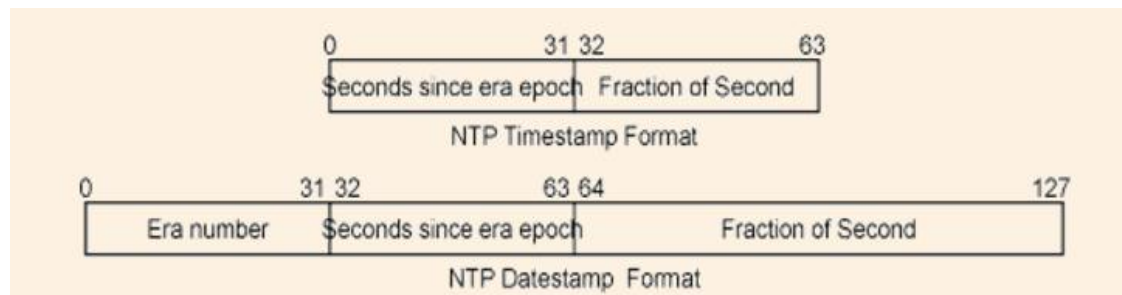


Figure 2-1: NTP Data Formats

2.1.2.5 ARCHITECTURE AND ALGORITHMS

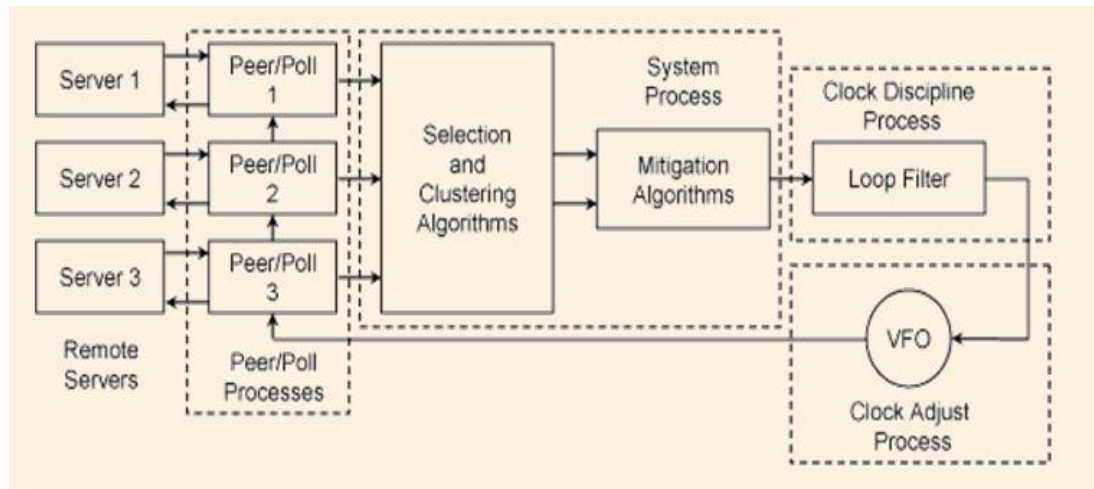


Figure 2-2: NTP Daemon Processes and Algorithms

Figure 2-2 shows the overall organization of the NTP architecture as both a client of upstream lower stratum servers and as a server for downstream higher status clients. The figure shows three servers as the remote synchronization source where each of these servers communicates with a pair of peer/poll processes. Packets are exchanged between the client and server using the on-wire protocol described later in this document. The poll process sends NTP packets at intervals ranging from 8 seconds to 36 hours, and these packets are managed in a way to maximize accuracy while minimizing network load. The peer process receives NTP packets and performs the packet sanity test then it discards the packets that fail the test. For the packets that succeed the test, the peer process runs the on-wire protocol that uses four raw timestamps: the origin timestamp T1 upon departure of the client request, the receive timestamp T2 upon arrival at the server, the transmit timestamp T3 upon departure of the server replay, and the destination timestamp T4 upon arrival at the client. This timestamps are recorded by the “rawstats” option of the “filegen” command, and are used to calculate the clock offset and roundtrip delay samples:

Clock offset

$$\vartheta = [(T2 - T1) + (T3 - T4)] / 2,$$

$(T4 - T1)$ is the time elapsed on the client side between the emission of the request packet and the reception of the response packet.

$(T3 - T2)$ is the time the server waited before sending the answer.

Roundtrip delay

$$\delta = (T4 - T1) - (T3 - T2).$$

The offset and delay statistics are processed by a set of mitigation algorithms, and the offset and delay samples most likely to produce accurate results are selected, and the servers that passed the sanity tests are declared selectable. Later, from the selectable population statistics are used by the “clock select algorithm” to determine a number of truechimers according to Byzantine agreement and correctness principles. Another set of algorithms combine the survivor offsets, designate one of them as the system peer and produces the final offset used by the “Clock Discipline Algorithm” to adjust the system clock time and frequency. The following section describes in more details the above mentioned algorithms.

The NTP software operates in each server and client as an independent process of daemon. The architecture of NTP daemon is illustrated in Figure 2-3. At designated intervals, a client sends a request to each in a set of configured servers and expects a response at some later time. The exchange results in four timestamps readings and these times are used by

the client to calculate the clock offset and roundtrip delay relative to each server separately. The clock filter algorithm discards offset “outliers” associated with large delays, which can result in large errors. These clock offsets produced by the clock filter algorithm for each server separately are then processed by the intersection algorithm in order to detect and discard misbehaving servers called “falsetickers”. The “truechimers” remaining are processed by the clustering algorithm to discard outliers. The survivors remaining are then weighted by synchronization distance and combined to produce the clock correction used to discipline the computer clock by the clock discipline algorithm. This algorithm is described in more detail in the following section.

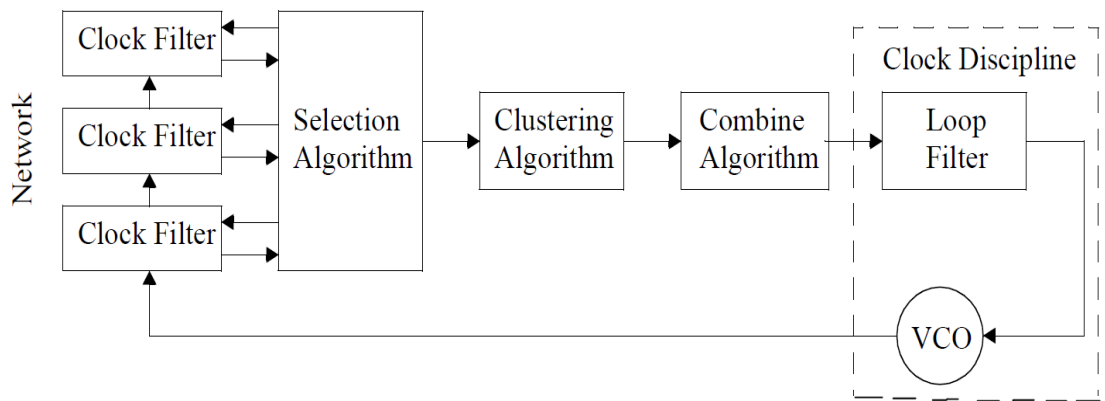


Figure 2-3: NTP ARCHITECTURE [6]

2.1.2.5.1 CLOCK FILTER ALGORITHM

The clock filter algorithm processes the offset and delay samples produced by the on-wire protocol for each peer process separately. It uses a sliding window of eight samples and picks out the sample with the least expected error. As the delay increases, the offset variation increases, so the best samples are those with the lowest delay. If the sample with lowest delay can be found, it would also have the least offset variation and would be the best candidate to synchronize the system clock. The clock filter algorithm works best when the

delays are statistically identical in the reciprocal directions between the server and client. When delays are not reciprocal, or where the transmission delays on the two directions are traffic dependent, this may not be the case. A common case is downloading or uploading a large file using DSL links; typically the delays are significantly different resulting in large errors.

In the clock filter algorithm the offset and delay samples from the on-wire protocol are inserted as the youngest stage of an eight-stage shift register, those discarding the oldest stage. Each time an NTP packet is received from a source, a dispersion sample is initialized as the sum of the precisions of the server and client. Precision is defined by the latency to read the system clock and varies from 1000 nanoseconds (ns) to 100 milliseconds (ms) in modern machines. The dispersion sample is inserted in the shift register along with the associated offset and delay samples, and then the dispersion sample in each stage is increased at a fixed rate of $15 \mu\text{s/s}$ representing the worst case error due to skew between the server and client clock frequencies. In each peer process the clock filter algorithm selects the stage with the smallest delay which generally represents the most accurate data. The peer jitter statistic is then computed as the root mean square (RMS) differences between the offset samples and the offset of the selected stage. The peer dispersion statistic is determined as a weighted sum of the dispersion samples in the shift register. As samples enter the register, the peer dispersion drops from 16 s to 8 s, 4 s, 2 s, and so forth.

When a source becomes unreachable, the poll process inserts a dummy infinity sample in the shift register for each poll sent, and after eight polls the register returns to its original state. Once a sample is selected it remains selected until a newer sample with lower delay is available. This typically occurs when an older selected sample is discarded from the

shift register. The result can be the loss of up to seven samples in the shift register. The output sample rate can never be less than one in eight input samples. The clock discipline algorithm is designed to operate at this rate.

2.1.2.5.2 CLOCK SELECT ALGORITHM

The clock select algorithm determines from a set of sources which are correct (truechimers) and which are not (falsetickers) based on a set of formal correctness assertions. To begin with, a number of sanity checks are performed to sift the selectable candidate from the source population.

- a. A stratum error occurs if the source had never been synchronized, or if the stratum of the source is below the floor option or not below the ceiling option of the “tos” command. The default values for these options are 0 and 15, respectively. It is important to note that 15 is a valid stratum for a server, but a server operating at that stratum cannot synchronize clients.
- b. A distance error occurs for a source if the root distance (also known as synchronization distance) of the source is not below the distance threshold “maxdist” option of the “tos” command. The default value for this option is 1.5 seconds
- c. A loop error occurs if the source is synchronized to the client. This can occur if two peers are configured with each other in symmetric modes.
- d. An unreachable error occurs if the source is unreachable or if the server or peer command for the source includes the “noselect” option.

Sources showing one or more of these errors are considered non-selectable. On the other hand, only the selectable candidates are considered in the following algorithm: given

the measured offset θ_o and root distance λ , the correctness interval is defined as $[\theta_o - \lambda, \theta_o + \lambda]$ of points where the true value of θ lies somewhere on the interval. The problem now consists in determining from a set of correctness intervals which represent truechimers and which represent falsetickers and in search of this solution a new interval is defined: the intersection interval is the smallest interval containing points from the largest number of correctness intervals. A candidate with a correctness interval that contains points in the intersection interval is a truechimer and the best offset estimate is the midpoint of its correctness interval. Furthermore, a candidate with a correctness interval that contains no points in the intersection interval is a “falseticker”. In summary, the midpoint sample produced by the clock filter algorithm is the maximum likelihood estimate and thus best represents the truechimer time.

2.1.2.5.3 CLOCK CLUSTER ALGORITHM

The Clock Cluster algorithm processes the truechimers produced by the clock select algorithm to produce a list of survivors which are used by the mitigation algorithms to discipline the system clock. The cluster algorithm operates in a series of rounds; at each round the truechimer furthest from the offset centroid is pruned from the population. The rounds are continued until a specified termination condition is met. First, the truechimer associations are saved on an unordered list with each candidate entry identified with index i ($i = 1, \dots, n$), where n is the number of candidates. Let $\theta(i)$ be the offset and $\lambda(i)$ be the root distance of the i th entry. Recall that the root distance is equal to the root dispersion plus half the root delay. For the i th candidate a statistic called the select jitter relative to the i th candidate is calculated as follows. Let $d_i(j) = |\theta(j) - \theta(i)| \lambda(i)$, where $\theta(i)$ is the peer offset of the i th entry and $\theta(j)$ is the peer offset of the j th entry, both produced by the clock filter

algorithm. The metric used by the cluster algorithm is the select jitter $\phi_s(i)$ computed as the root mean square (RMS) of the $d_i(j)$ as j ranges from 1 to n . The objective at each round is to prune the entry with the largest metric until the termination condition is met. The select jitter must be recomputed at each round, but the peer jitter does not change. The termination condition has two parts. First, if the number of survivors is not greater than the “minclock” threshold set by the “minclock” option on the “tos” command, the pruning process terminates. The “minclock” defaults is 3, but can be changed to fit special conditions. The second termination condition is more intricate. Figure 2-4 below shows a round where a candidate of (a) is pruned to yield the candidates of (b). Let ϕ_{max} be the maximum select jitter and ϕ_{min} be the minimum peer jitter over all candidates. In (a), candidate 1 has the highest select jitter, so $\phi_{max} = \phi_s(1)$. Candidate 4 has the lowest peer jitter, so $\phi_{min} = \phi_R(4)$. Since $\phi_{max} > \phi_{min}$, select jitter dominates peer jitter so the algorithm prunes candidate 1. In (b), $\phi_{max} = \phi_s(3)$ and $\phi_{min} = \phi_R(4)$. Since $\phi_{max} < \phi_{min}$, pruning additional candidates does not reduce select jitter, and the algorithm terminates with candidates 2, 3 and 4 as survivors. The survivor list is passed on to the mitigation algorithms, which combine the survivors, select a system peer, and compute the system statistics passed on to dependent clients.



Figure 2-4: Cluster Algorithm [7]

2.1.2.5.4 CLOCK DISCIPLINE ALGORITHM (NTP V4)

The Clock Discipline algorithm adjusts the computer clock time as determined by NTP, compensates for the intrinsic frequency error, and adjusts the poll interval and loop time constant dynamically in response to measured network jitter and oscillator stability. The algorithm functions as a hybrid of two different feedback control systems. In a phase-lock loop (PLL) design, the measured time errors are used to discipline a type-II feedback loop which controls the phase and frequency of the clock oscillator. In the frequency-lock loop (FLL) design, the measured time and frequency errors are used separately to discipline type-I feedback loops, one controlling the phase and the other controlling the frequency.

The system processes polls the peer processes at intervals from a few seconds to over a day, depending on peer type. When a new sample of offset, delay and dispersion is available in a peer process, a bit is set in its state variables. The system process, upon noticing this bit, clears it and calls the clock selection, clustering and combining algorithms. This algorithm adjusts the clock oscillator time and frequency with the aid of the clock adjust process, which runs at intervals of one second.

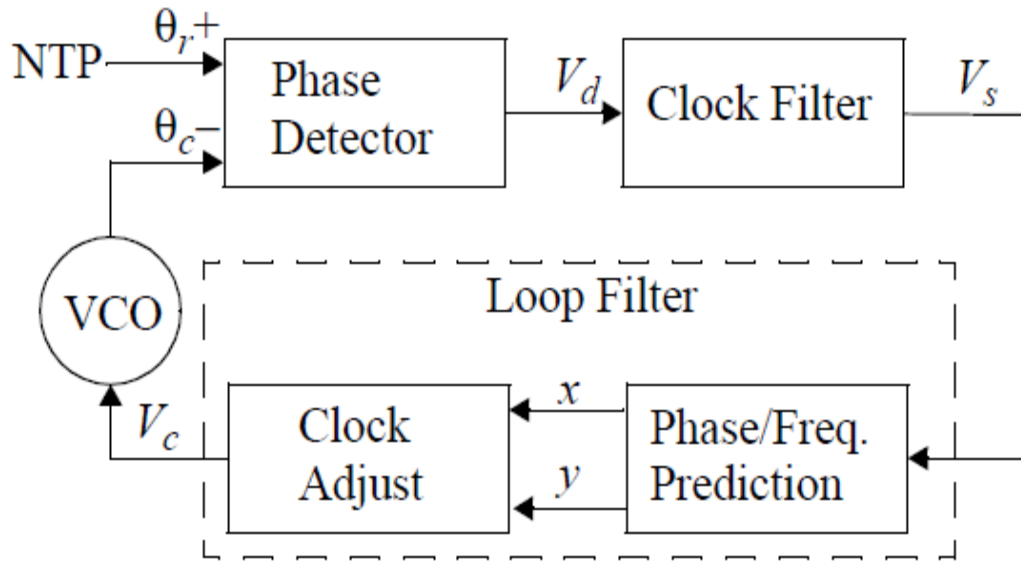


Figure 2-5: Clock Discipline Algorithm

The clock discipline algorithm is implemented as a feedback control loop shown in Figure 2-5. The variable θ_r represents the reference phase provided by NTP and θ_c the control phase produced by the variable-frequency oscillator (VFO), which controls the computer clock. The phase detector produces a signal V_d that represents the instantaneous phase difference in between θ_r and θ_c . The clock filter functions as a tapped delay line, with the output V_s taken at the sample selected by the algorithm. The loop filter, with impulse response $F(t)$ produces a correction V_c which controls the VFO frequency θ_c and thus its phase θ_c . The characteristic behavior of this model, which is determined by $F(t)$ and the various gain factors is studied many text books and summarized in [8].

This redesigned clock discipline algorithm used in NTP v4 is implemented using two sub algorithms, one based on a linear, time-invariant PLL, and the other on a nonlinear, predictive FLL. Both predict a time correction x as a function of phase error θ , represented by V_s in Figure 2-6.

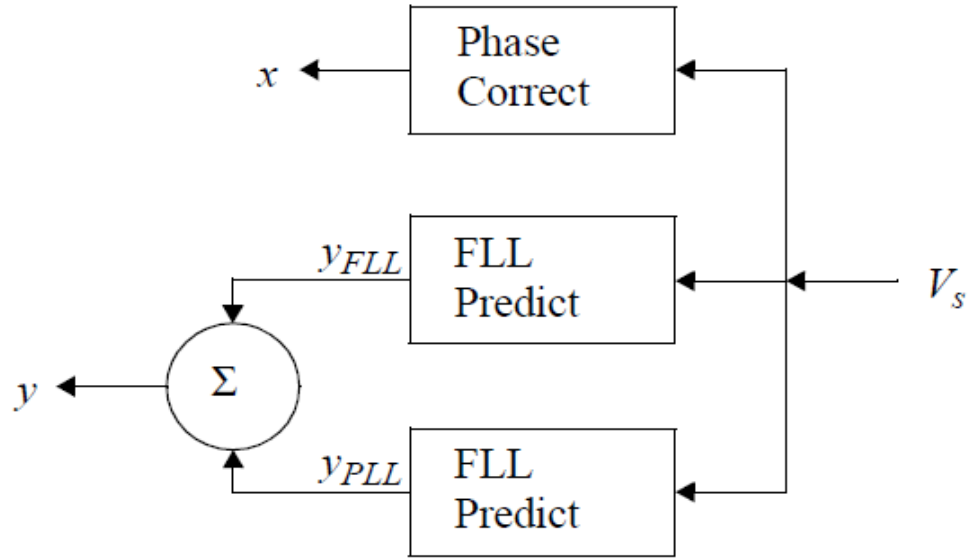


Figure 2-6: FLL/PLL Prediction Functions

The PLL predicts a frequency adjustment y_{PLL} as an integral of past time offsets, while the FLL predicts a frequency adjustment y_{FLL} directly from the difference between the last time correction and the current one. The two adjustments are combined and added to the current clock frequency y , as shown in figure 2-6. Then the x and y are used by the clock adjust process to adjust the VCO frequency and close the feedback loop, as shown in Figure 2-5. A complete mathematical derivation of the clock discipline algorithm is described in [9].

2.1.2.5.5 NTP POLL PROCESS

The poll process sends NTP packets at intervals determined by the clock discipline algorithm. The process is designed to provide a sufficient update rate to maximize accuracy while minimizing network overhead. This rate is determined by a poll (power of 2) exponent with a range between 3 (8 seconds) and 17 (36 hours). The minimum and maximum poll exponent within this range can be set using the “minpoll” and “maxpoll” options of the

server command, with default of 2^6 (64 seconds) and 2^{10} (1024 seconds) respectively in NTP v3. However, in NTP v4 these values can be set to a minimum of 2^4 (16 seconds) and a maximum of 2^{17} (131,072). Within this range, the clock discipline algorithm automatically manages the poll interval based on current network jitter and oscillator wander. The poll interval is managed by a heuristic algorithm developed over several years of experimentation and depends on an exponentially weighted average of clock offset differences, called clock jitter, and a jiggle counter.

As an option of the server command, instead of a single packet, the poll process can send a burst of several packets at 2-s intervals. This is intended to reduce the time to synchronize the clock at initial startup (iburst) and /or to reduce the phase noise at the longer poll intervals (burst). For the iburst option 6 packets are sent in the burst, which is the number normally needed to synchronize the clock; for the burst option, the number of packets in the burst is determined by the difference between the current poll exponent and the minimum poll exponent as a power of 2. For example, with the default minimum poll exponent of 6 (64 seconds) only one packet is sent for every poll, while the full number of eight packets is sent at poll exponents of 9 (512 seconds). This will ensure that the average headway will never exceed the minimum headway. In addition, when ibusrt or burst is enabled, the first packet of the burst is sent, but the remaining packets sent only when the reply to the first packet is received. This means that, even if a server is unreachable, the network load is no more than at the minimum poll interval. A key statistic to control the poll interval is the RMS error measured by the clustering algorithm which sifts the best subset of clocks from the current peer population. This process is called “select dispersion” and expressed by “ ϵ_{SEL} ”. These samples square values are held in a shift register. The system

dispersion “ ϵ_{SYS} ” is then calculated from the RMS sum of “ ϵ_{SEL} ” and the peer dispersion “ ϵ_{PEER} ” of the selected peer.

$$\epsilon_{SYS} = \sqrt{\langle \epsilon_{SEL}^2 + \epsilon_{PEER}^2 \rangle_n}$$

Where $n = 4$ samples chosen by experiment. If $|\theta| > Y\epsilon_{SYS}$, Where $Y = 5$ is experimentally determined, the oscillator frequency is deviating too fast and the poll interval is reduced in stages to the minimum. If the opposite case holds for some updates, the poll interval is slowly increased in steps to the maximum. Under typical operating conditions, the interval hovers close to the maximum, but on occasions, when the oscillator frequency wanders more than about 1 PPM, it quickly drops to lower values until the wander subsides.

2.1.2.5.6 NTP CLOCK STATE MACHINE

The NTP algorithms work well to sift good data under conditions of light to moderate network and server loads, but under conditions of extreme network congestion, operating system latencies, and oscillator wander, linear time-invariant systems (PLL) and predictive systems (FLL) may fail. The results can be frequent time step changes and large time and frequency errors, and in order to work with large transients the clock discipline algorithm in NTP v4 is managed by the state machine shown below in figure 2-7.

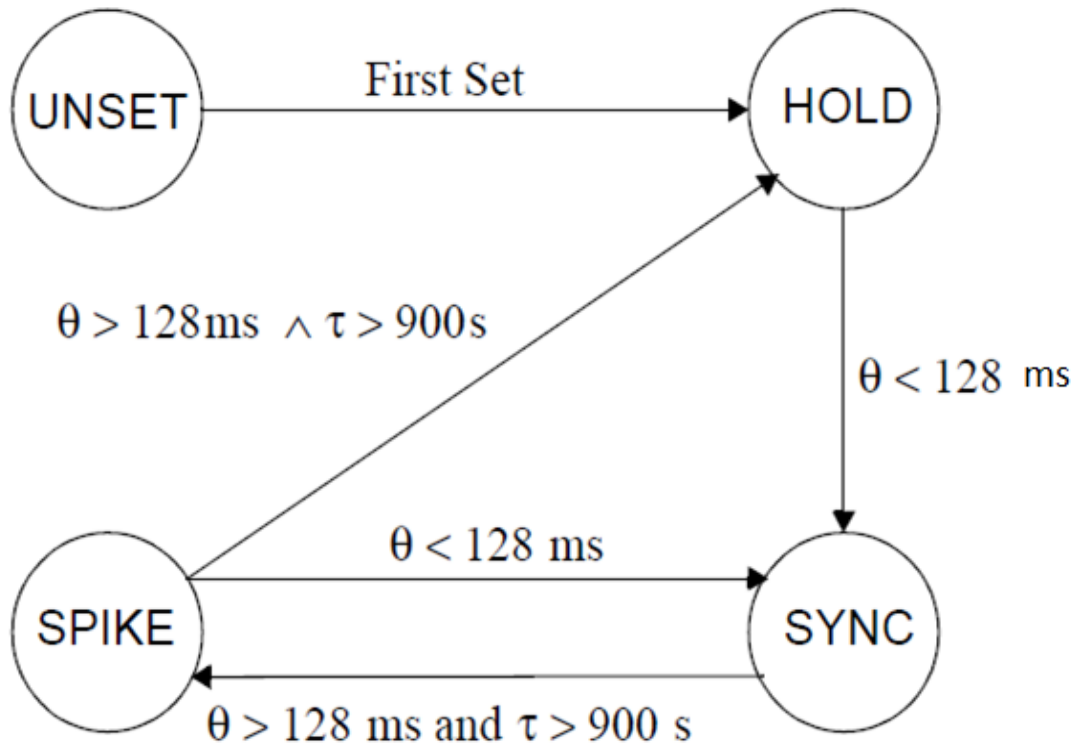


Figure 2-7: Clock State Machine

Initially, the machine is unsynchronized and in UNSET state, then if the minimum poll interval is 1,024 s or greater, the first update received sets the clock and transitions to HOLD state. If the interval is less than 1,024 s these actions will not occur until several updates to allow the synchronization to be reduced below 1 s, and allow the algorithms to accumulate reliable error estimates.

In HOLD state, sanity checks, spike detectors and tolerance clamps are disabled, and the clock discipline algorithm is forced to operate in FLL mode only to allow the fastest adaptation to the particular oscillator frequency. The machine remains in this state for at least 5 updates. After this and the nominal clock offset has decreased below 128 milliseconds, the machine transitions to SYNC state and remains there pending unusual conditions.

In SYNC state, the sanity checks, spike detectors and tolerance clamps are operative. To protect against frequency spikes in FLL predictions at small update intervals, the frequency adjustments are clamped at 1 PPM, and to protect against runaway frequency offsets in FLL predictions at large update intervals, the frequency estimate is clamped at 500 PPM, and finally, to protect against disruptions due to severe network congestion, frequency adjustments are disabled if system dispersion exceeds 128 milliseconds.

2.2 THE GLOBAL POSITIONING SYSTEM DAEMON (GPSD)

GPSD is a software program that monitors one or more GPS or AIS receivers attached to a host computer through serial or USB ports. AIS (Automatic Identification System) is a device installed on some vessels and used to transmit their position, speed and course among other information. A gpsd program makes all data on the location/course/velocity of the sensors available to be queried on TCP port 2947 of the host computer. With gpsd, multiple time and location-aware client applications such as NTP can easily share access to these receivers without contention or loss of data.

2.3 PULSE PER SECOND (PPS)

PPS is an electrical signal that has a width of less than a second and a sharply rising or abruptly falling edge that accurately repeats once per second. This signal is output by radio beacons, GPS receivers and other types of precision oscillators. This signal can be used to discipline the local clock oscillator to a high degree of precision, typically to the order less than 10 μ s in time and 0,01 parts-per-million (PPM) in frequency. The PPS signal can be connected via the data carrier detector (DCD) pin of a serial port or via the acknowledge (ACK) pin of a parallel port. Both connections require operating system support. It is

available in Linux, FreeBSD and Solaris. Support is also available on an experimental basis for several other systems. The PPS application program interface defined in RFC-2783 (PPSAPI) is the only PPS interface supported; older versions are no longer supported.

2.4 SYNCHRONIZATION OF THE NTP SERVER WITH GPS AND PPS

Our objective of this section is to have a Global Positioning System (GPS) receiver device driving a pulse-per-second (PPS) signal to the Network Time Protocol Daemon (NTPD) server for a highly accurate time reference server. This section will include two parts: the first one describes the devices, drivers and daemons necessary to establish the system time synchronization with the GPS receiver, and the second part which will be covered in the next chapter, will summarize the configuration steps and configuration files necessary to get the system up and running and synchronized to a GPS receiver with an accuracy that depends on the GPS receiver type.

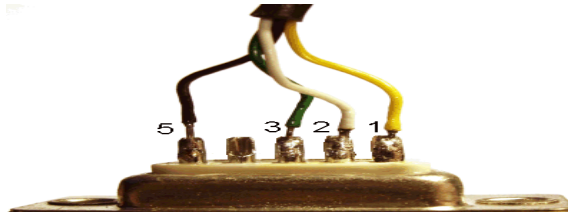
2.4.1 THE GPS DEVICE

The author has found two ways to propagate the PPS signal to the ntpd server, each case presenting its own variants. However, the GPS receiver must be a device capable of sourcing two different types of data: the absolute data and time, and the 1-Hz clock signal (PPS). The first one provides the complete information of the current date and time with poor accuracy since this information is sent over the data line of the serial port (TxD/pin 2) and encoded using some type of protocol, i.e. NMEA. The PPS on the other hand, provides a very accurate clock (1 μ S in the GPS 18LVC receiver) but with no reference at all to the absolute time. This signal is wired to the Data Carrier Detect (DCD) pin 1 of the serial port. PPS indicates with good precision when each out second begins, but it does not tell us which

second it is. Due to this fact this timing information must be combined with the protocol messages sent by the GPS receiver to have both precision and a complete timestamp at the same time. This GPS device must speak the NMEA protocol which sends out NMEA messages every second.

2.4.2 NTPD REFERENCE CLOCKS

The NTPD server supports several types of drivers, these drivers are low level callback functions that are registered within the NTPD core and implement the access to several types of local clocks such as GPSs. Each driver is identified by a pseudo-IP address identifier and listed in the NTP configuration file located at `"/etc/ntp.conf"`. There are two drivers involved within the GPS/NTP time synchronization: `127.127.20.x`: NMEA Reference Clock driver and `127.127.28.x`: SHM (shared memory) driver. The NMEA reference clock driver expects a GPS device sending out NMEA messages to a system via a serial port named `"/dev/gpsX"` and the PPS signal wired through the DCD pin and accessible from a `"/dev/gpsppsX"` device. Figure 2-8 below shows the wiring of the GPS receiver to interface with the serial port.



Wire	DB9 female serial	USB	Function
Red	-	Red	+5 Volt power
Black (thick)	-	Black	Power ground
Black (thin)	5	-	Signal ground
Black (loose in cable)	-	-	not used
Yellow	1	-	PPS pulse
White	2	-	TxD from GPS
Green	3	-	RxD to GPS
Shield	-	Shield	Shield

Figure 2-8: Garmin GPS 18x LVC to RS232 and USB wiring

The `"/dev/gpsX"` appears in the system as a link to the `"/dev/ttyS0"` serial device, and the `"/dev/gpsppsX"` appears as a link to the `"/dev/pps0"` device which is provided by the kernel PPS API. The API collects and distributes a precision kernel clock information from/to userlands programs and supports the DCD pin connected to a 8250 UART. The DCD pin is sensed using a new serial line discipline named PPS, which is an extension of the TTY line discipline. This sensing occurs as an interrupt time, so it provides a very precise time stamping of the DCD events. This PPS API, also known as LinuxPPS is available as a module on Linux kernel version 2.6.34 or later. For Linux systems with kernel older than 2.6.34 LinuxPPS is not yet available and a patch must be applied. In order for the NTPD to synchronize the system clock with the GPS receiver and using the pps signal for precision timing the following two lines must be added to the `"/etc/ntp.conf"` file:

Server 127.127.20.0 mode 1 minpoll 4 prefer

Fudge 127.127.20.0 flag3 1 flag2 0 time1 0.0

Where

Mode = 1, means that only the GPMRC messages of the NMEA protocol will be analyzed.

Flag3 = 1, tells ntpd to use the PPS line discipline of the kernel

Flag2 = 0, tells the driver to use the rising edge of the DCD signal to indicate the start of each second.

time1 time: Specifies the PPS time offset calibration factor, in seconds and fraction, with default 0.0

To activate the PPS line discipline on the serial port connected to the GPS, it is necessary to run the “ldattach” utility, which is part of the “util-linux-ng” package v2.14 and up and will run in the background to keep the serial port open and the discipline active. Ldattach was provided with the Linux distribution used in our test, however for any system where this tool is not provided it will be necessary to build the tool from its sources which can be found by referring to kernel.org (<http://www.kernel.org/pub/linux/utils/util-linux-ng/>) for the source code and your distro provider for an updated util-linux package.

2.4.3 SHM REFERENCE CLOCK

The Shared Memory (SHM) driver accepts delayed timing information from a System-V IPC (Inter Process Communication) shared memory and this timing information is observed in the ntpd logs. The timing information is written there by some process; this process would

read the information from the GPS and write it to the shared memory so that ntpd can process it. One user space utility that performs this task is gpsd, which is a general-purpose daemon designed to talk to most types of GPS modules and according to the texts it is also capable of processing the PPS signals and sending timing information to ntpd via a shared memory device. Gpsd feeds two devices to ntpd, one with the absolute timestamp parsed from the NMEA messages, and another feeding the PPS. Ntpd sees both devices as two different SHM devices so the "ntp.conf" file must include these lines:

```
Server 127.127.28.0 minpoll 4
```

```
fudge 127.127.28.0 refid GPS
```

```
Server 127.127.28.1 minpoll 4 prefer
```

```
Server 127.127.28.1 refid PPS
```

Where

Refid is just a string that specifies the driver reference identifier.

3 METHODOLOGY

In our approach to build this new Son-O-MERMAID prototype, Figure 1-1 was modified, as described in Figure 3-1 below. Building Son-O-MERMAID could be divided in two major components: one component is the mechanical part which includes building the instrumentation housing, array assembly and suspension cable. This task is out of scope of this work and has been finalized. The other component consists on the hardware integration and processing which constitute the focus of the author's work.

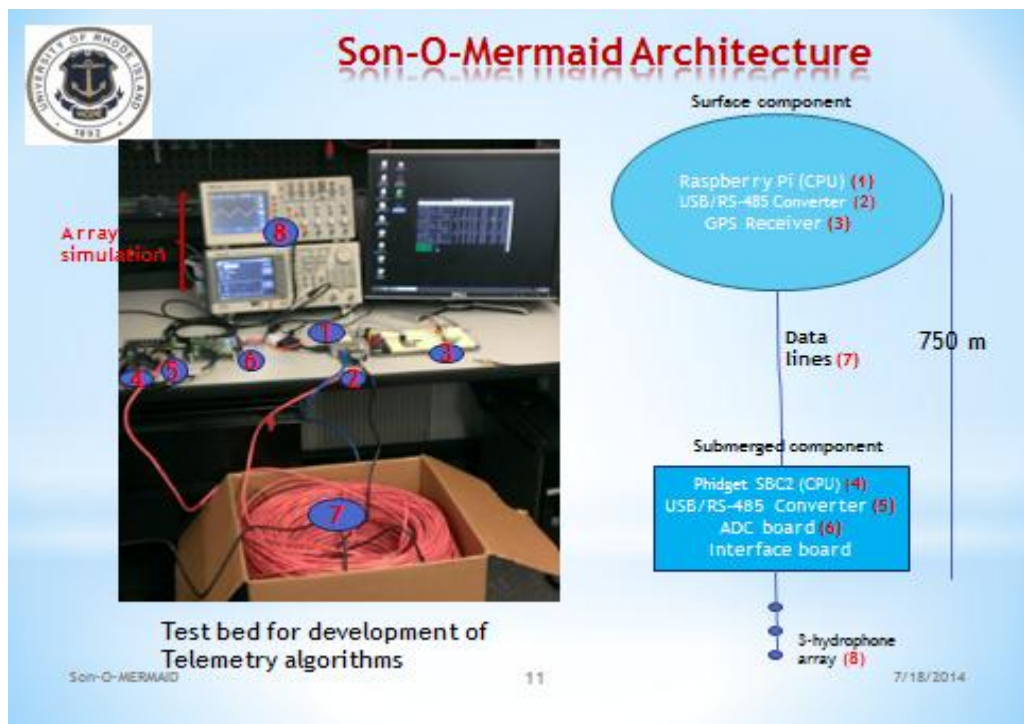


Figure 3-1: Architecture of Son-O-MERMAID

3.1 Son-O-MERMAID SYSTEM DESIGN

Figure 3-1 describes a high level architecture of Son-O-Mermaid on the right, and a test bed for development of the telemetry algorithms on the left. As this figure shows, Son-O-MERMAID can be divided in three subcomponents: First, a surface component which

receives and stores the acoustic data sent from the submerged unit for further analysis. This unit runs a very accurate system time synchronized to a GPS receiver and used to time stamp the acoustic data. Second a submerged component immersed at a depth of ~750 meters which collects, digitize and samples acoustic data. Third, the interface that connects the submerged and surface components, which in this figure consists in the data lines.

3.1.1 THE SUBMERGED COMPONENT

The submerged component includes the following parts: a three-hydrophone array, an analog to digital converter (ADC), a central processing unit (CPU) and one RS-485 adapter. All these parts with the exception of the hydrophones are placed inside a pressure vessel. Figure 3-2 shows this component and a description and integration of its parts follows.

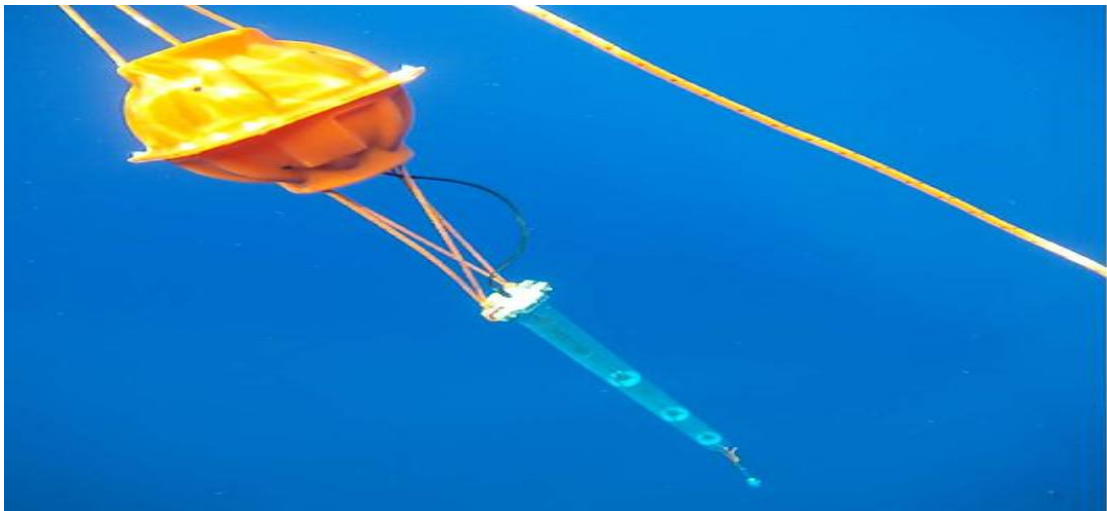


Figure 3-2: Submerged component as deployed in 2012, on the first implementation of Son-O-MERMAID

3.1.1.1 3-HYDROPHONE ARRAY

The hydrophones arranged in this array are manufactured by High Tech Inc. and their technical specifications are listed in Table 1 in the Appendix, section A.2. This 3-hydrophone

array was built at the Equipment Development Laboratory (EDL) at URI's Narragansett Bay Campus by Catherine Cipolla and Gary Savoie, whose combined scientific instrument design and fabrication experience exceeds some 50 years. The array is displayed in Figure 3-2 above and Figure 3-3 below. This array is expected to be used in the next deployment of Son-O-MERMAID (version 2).



Figure 3-3: Hydrophone array for Son-O-MERMAID.

During the prototype design of Son-O-Mermaid version 2 rather than using the array, the majority of the time the array's acoustic data were simulated by a function generator as shown in figures 3-4a and 3-4b below. The wave form from the function generator was split into three channels to simulate the three inputs from the array into the ADC board. At the ADC board, the analog data were digitized and read in by the Phidget SBC in the submerged unit. The data were then sent to the surface unit as a complete file or as sample by sample depending on the approach selected, as it will be described later.

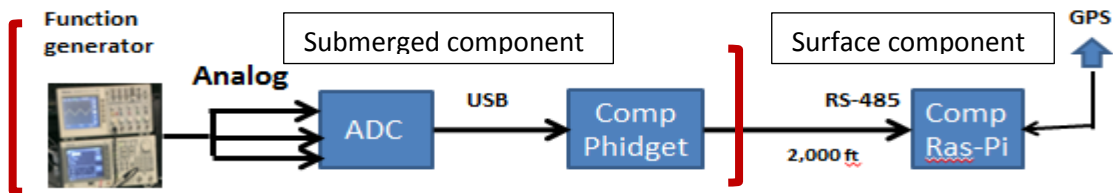


Figure 3-4a: Graphic description of array and telemetry simulation

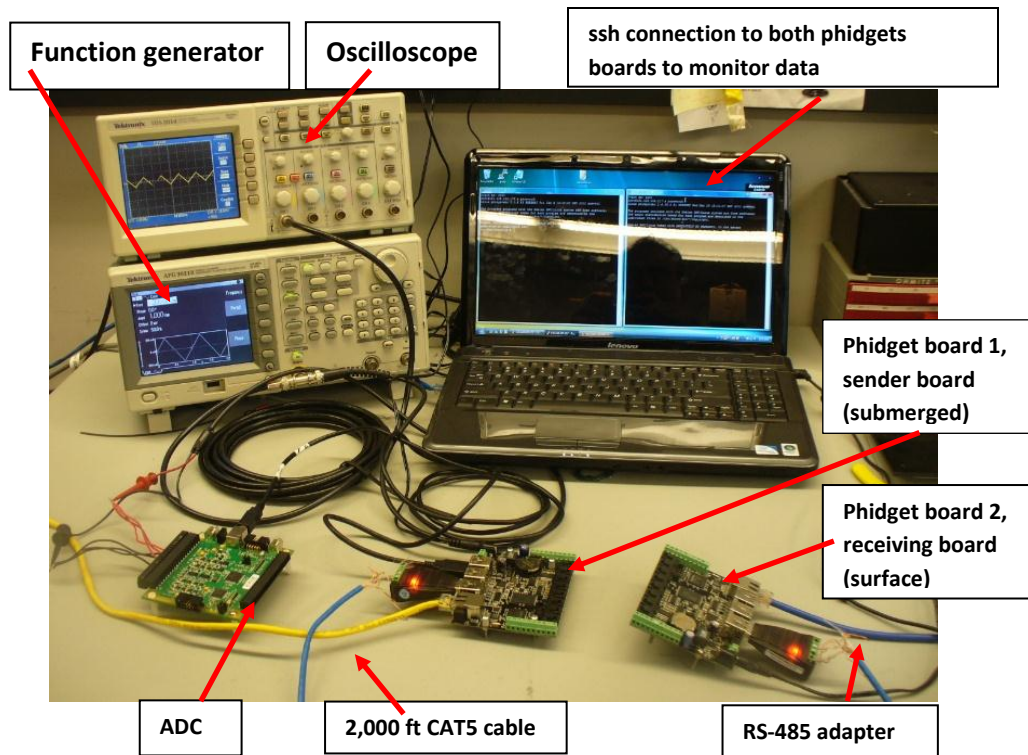


Figure 3-4b: Test bed development of Son-O-Mermaid with Array Simulation.

3.1.1.2 HYDROPHONES AND ADC INTERFACE BOARD

During the development of prototype version 1, a circuit board was designed and used as the interface between the hydrophone array and the ADC board. This interface board receives power from the source (battery) and distributes it to the hydrophones. In addition, it functions as a bridge to canalize the acoustic data from the hydrophones through three different channels, one per hydrophone, and drives this data as input to the ADC. This circuit interface board is shown on Figure 3-5a and 3-5b below. Figure 3-5b depicts the internal wiring of the board shown in figure 3.5a.

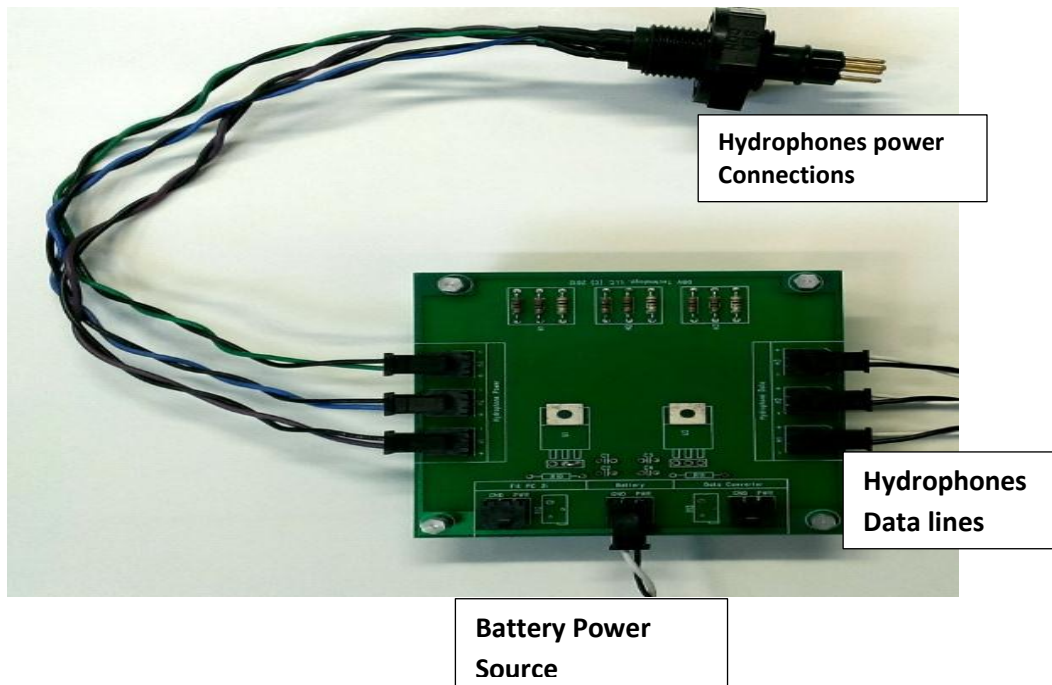


Figure 3-5a: Hydrophones & ADC Interface board

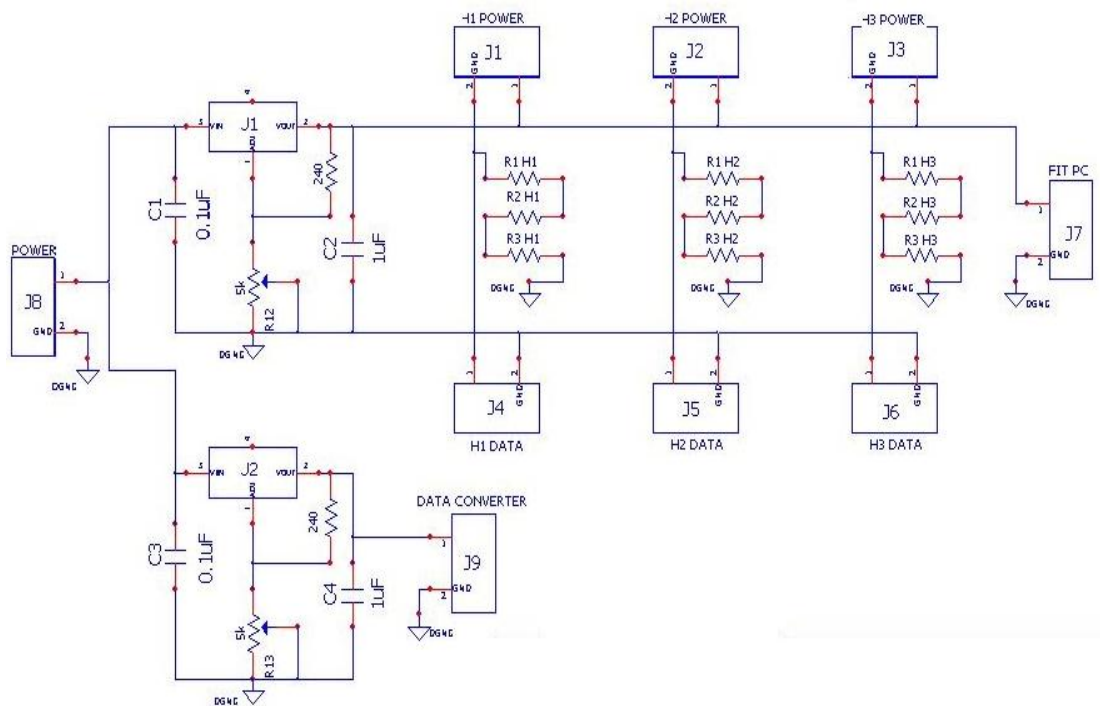


Figure 3-5b: Hydrophones & ADC Interface circuit.

3.1.1.3 ANALOG TO DIGITAL CONVERTER (ADC)

The USB-7202 ADC board was used to perform the analog to digital conversion in the Son-O-MERMAID prototype. This board receives analog data from the hydrophones and digitizes it and is programmed to sample it at a rate of 100 Hz. This acoustic data is then pulled by the Phidget board 100 samples each second and sent to the upper unit. The chosen ADC board operates by implementing the DAQFlex framework, which consists of combining a driver with a message-based command protocol. The DAQFlex framework consists of a software API, DAQFlex device driver, and a DAQ device message engine. A DAQFlex program sends DAQFlex messages to the driver. The driver sends the encapsulated messages to the data acquisition device. The device interprets the message using the message engine, and sets its corresponding attributes using the DAQ engine. The data acquisition device then returns the requested data to the DAQFlex driver, which returns the data in an array (ScanData) to the program. The DAQFlex framework is described in Figure 3-6 below and is implemented in the data acquisition script that runs in the submerged component described in sections 3.2.3 and 3.3 below.

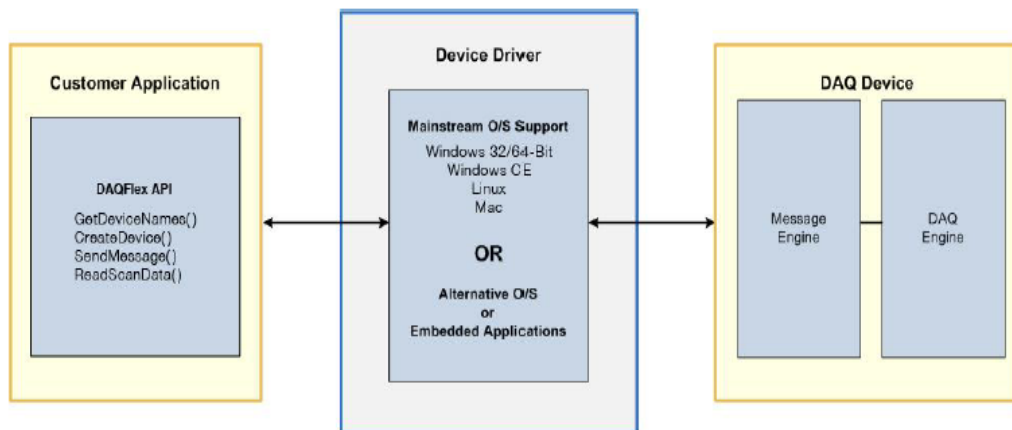


Figure 3-6: DAQFlex Framework

To use the DAQFlex framework under a Linux platform which was the approach taken, the following software requirements needed to be met: Linux 32-bit Operating System, Linux kernel 2.4 or later, Mono Framework 2.0 or later and the “libusb” user-mode driver version 1.0.0.0. This software was installed in the computer running in the submerged component.

Since Mono is an open source, cross-platform implementation of C# and CLR that is compatible with Microsoft.NET, and “IronPython” is also an open source implementation of the Python programming language integrated with the .NET Framework. This gave us an additional option of using IronPython to program the ADC. Both C# and IronPython were tested and both worked when reading digitized acoustic data into the submerged unit; however, to transmit the acoustic data from the lower unit to the upper one via the serial port, only the IronPython approach was tested with a hope that a ready to use module (pyserial) would make things simple. We sustained our decision of using IronPython but our hope for simplicity wasn’t met as it will be explained later. Figure 3-7 shows the ADC described in this section and Table 2 in Appendix, section A.2 documents technical specifications of the board.

The DAQFlex permits the programming of DAQ devices such as the USB-7202 ADC by using a simple message-based interface. Installing the universal library is described in the Phidget configuration section, and the DAQFlex’s command selection used in this prototype is coded into the “mermaid_rec.py” script used to set up the acquisition board to sample the hydrophone data at a rate of 100Hz. This sampled data is then sent via RS-485 to the upper unit. The “mermaid_rec.py” script is located in Appendix, section A.1.3.1.



Figure 3-7: USB-7202 16-bit Analog to Digital Converter

3.1.1.4 USB TO RS-485 ADAPTER (Model: xs885)

The USB to RS-485/RS-422 converter shown below, automatically senses if RS-485 or RS-422 is connected to the serial interface. This is a plug and play device and all that is needed is to connect the wires. Power for the converter is provided by the USB port. This adapter uses the FT232 serial processor chip from FTDI and does not require any extra driver to be installed in the Linux system. It works by default and the overall features are provided in Table 6 in the Appendix, section A.2. Two configurations were tested on the device: RS-422 and RS-485 and both proved to work; however, RS-485 was used for two reasons: it uses three wires as opposed to 5, and secondly, data only flow in one direction (half duplex) from the submerged component to the surface, as opposed to full duplex which unnecessarily would result in greater expense in terms of power usage. Figure 3-8 below displays this device.



Figure 3-8: USB to RS-485/RS-422 converter

3.1.1.5 THE CENTRAL PROCESSING UNIT (CPU)

The CPU in the lower unit is a computer expected to receive digitized acoustic data from the ADC board sampled at a rate of 100 Hz. This data is then sent via RS-485 to the upper unit where it will be stored and manipulated later. It is required that this computer includes at least two USB ports, one to connect to the ADC board and the other to host the RS-485 adapter. It must run on Linux Operating System and the unit must be low in power consumption. In an attempt to select the most power efficient unit available in the market two boards were tested: the first computer tested was the “Fit PC2i” and the second was the “Phidget SBC2”. The fit PC2i is a miniature (10 x 11 x 2.5 cm) full featured computer running Linux mint, with an Intel Atom 1.6GHz CPU and up to 2GB RAM that draws 8 to 10 Watts of power. More detailed product specifications are provided in the Appendix, section A.2.3. The Fit PC2i was the first CPU used while developing the submerged component. It fulfilled two of the three basic requirements: it has two USB ports and run on Linux OS; however, it has a high power consumption. As a full operating computer it had too many unnecessary features and a high power consumption and, hence this CPU, wasn’t an optimal solution for Son-O-MERMAID. The second computer tested was the Phidget SBC2 version 1072_0. This is

a single board computer (SBC) running Debian 6.0 with 64 MiB SDRAM, 512 MiB Flash and 6 USB 2.0 full speed ports. This unit was a better fit for Son-O-MERMAID. It supported the two USB ports required, runs Linux OS, and it was much more power efficient than the Fit PC2i. This computer also offers more features than necessary, but is a much better fit than the Fit PC2i due to the lower power consumption. This was the best choice available that matched our design criteria at the time. By the time the Son-O-MERMAID prototype was designed and tested, the Phidget SBC2 board had been discontinued and replaced by the Phidget SBC3 (1073) board. More detailed product specifications are provided in Table 4 in the Appendix, section A.2. In our search for a power efficient computer, the PC-104 products were also researched and a possible option using two PC-104 boards, a CPU and an ADC, was identified; however, the cost and power consumption offered by the Phidget board were lower. Figure 3-9 shows the Phidget SBC2 board.

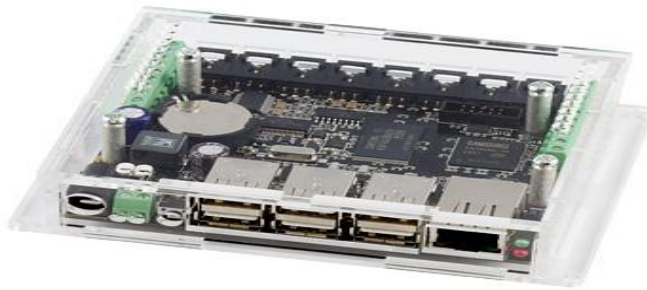


Figure 3-9: Phidget sbc2 version 1072_0

3.1.1.6 CONFIGURATION OF THE SUBMERGED COMPONENT

Figure 3-10 below depicts the hardware configuration of the submerged component. The interface board gets power from the battery source and distributes to the rest of the equipment. The hydrophones connect to the interface board and receive power from it, and

in return acoustic data is canalized from the hydrophones to the interface board. This acoustic data is then received by the ADC board where it gets digitized and sampled at a rate of 100 Hz. The data is then collected by the Phidget SBC2 board to be stored into files or to be sent to the upper unit via RS-485 based on the algorithm selected. This unit was tested during a pre-launch event prior to deployment of first prototype of Son-O-Mermaid in 2012. The results of this test are provided in Figure 3-11.

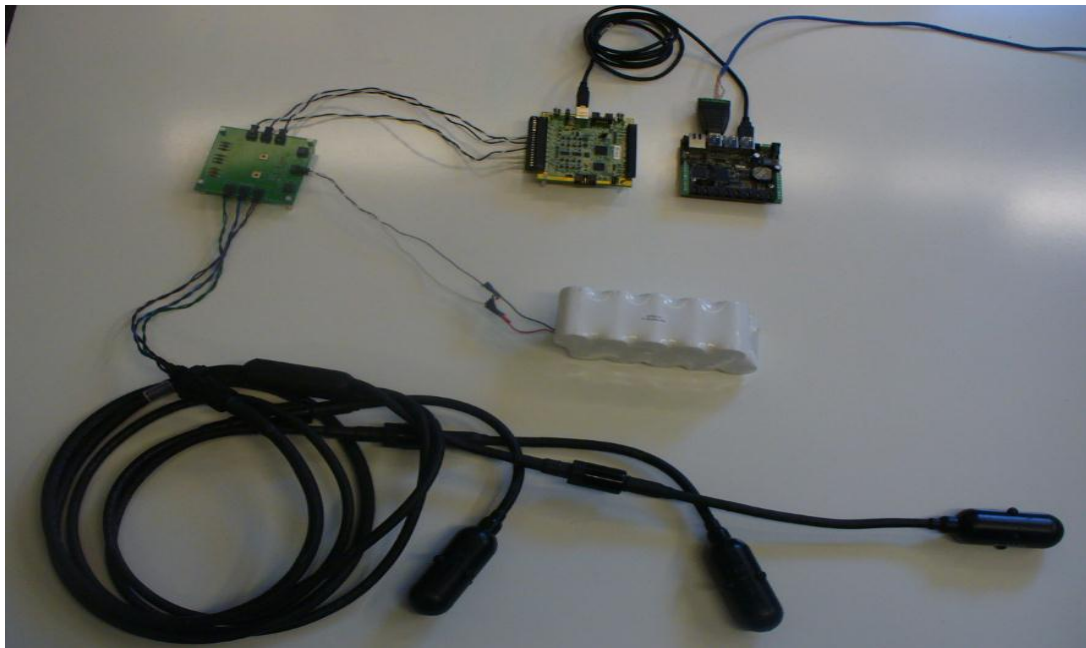


Figure 3-10: Configuration of Son-O-MERMAID submerged component

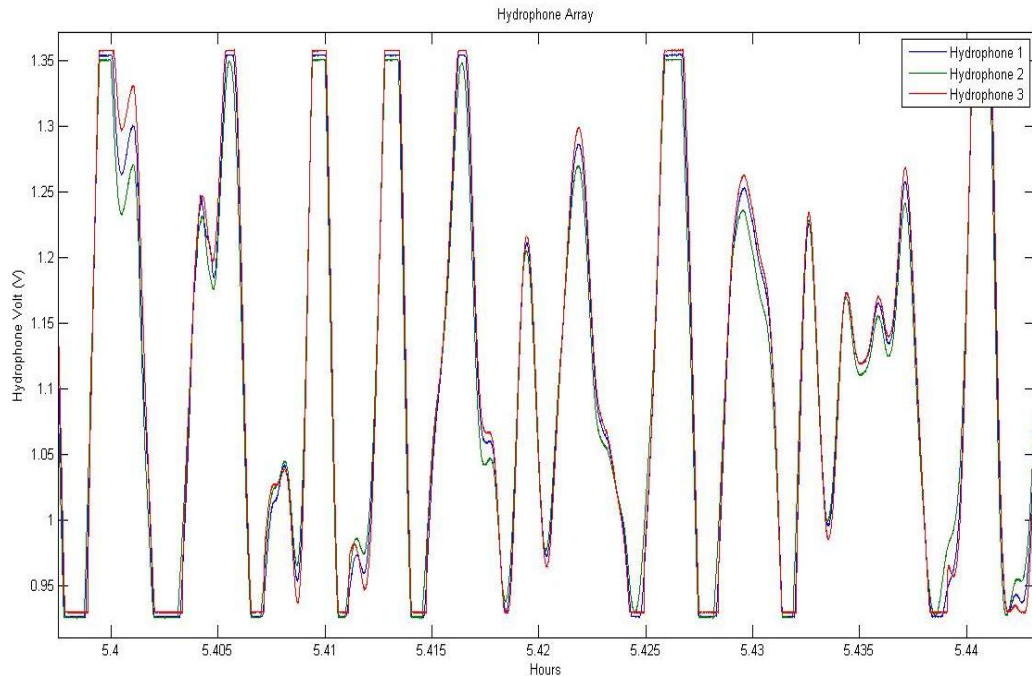


Figure 3-11: Data collected at pre-launch test of Son-O-Mermaid prototype one in October, 2012.

3.1.1.6.1 PHIDGET SBC2 SOFTWARE CONFIGURATION

The system comes with factory default settings. After the first boot it is necessary to run a series of configuration steps to enable the basic features on the board. All the configuration instructions are found online in the Phidget’s User Guide page [10]. The basic configuration steps entailed in updating the applications software in the board by installing “ssh” to be able to work remotely with the device, and installing Mono. The Mono application is necessary to run “ironPython”, the language selected to communicate with the Analog to Digital Converter board. Most of the testing for Son-O-MERAID was conducted in the laboratory by simulating the hydrophones acoustics. To verify correct operation of the submerged unit the development test bed was set up as described in Figure 3-12 below and the following sequence of events were performed: a triangular wave was injected by the function generator into the ADC board, the ADC digitized and sampled the data which was

then read in by the Phidget SBC and saved into one-minute files, the data stored at the Phidget computer were then plotted to verify that the signal going into the system was equal to the received signal output. The results of these events are shown in Figures 3-13 to 3-15 below.

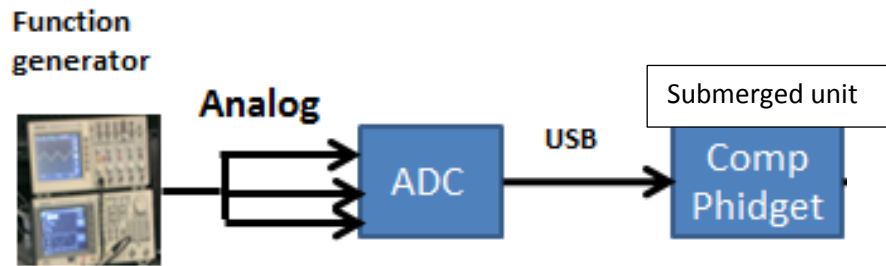


Figure 3-12: Son-O-MERMAID submerged unit simulation

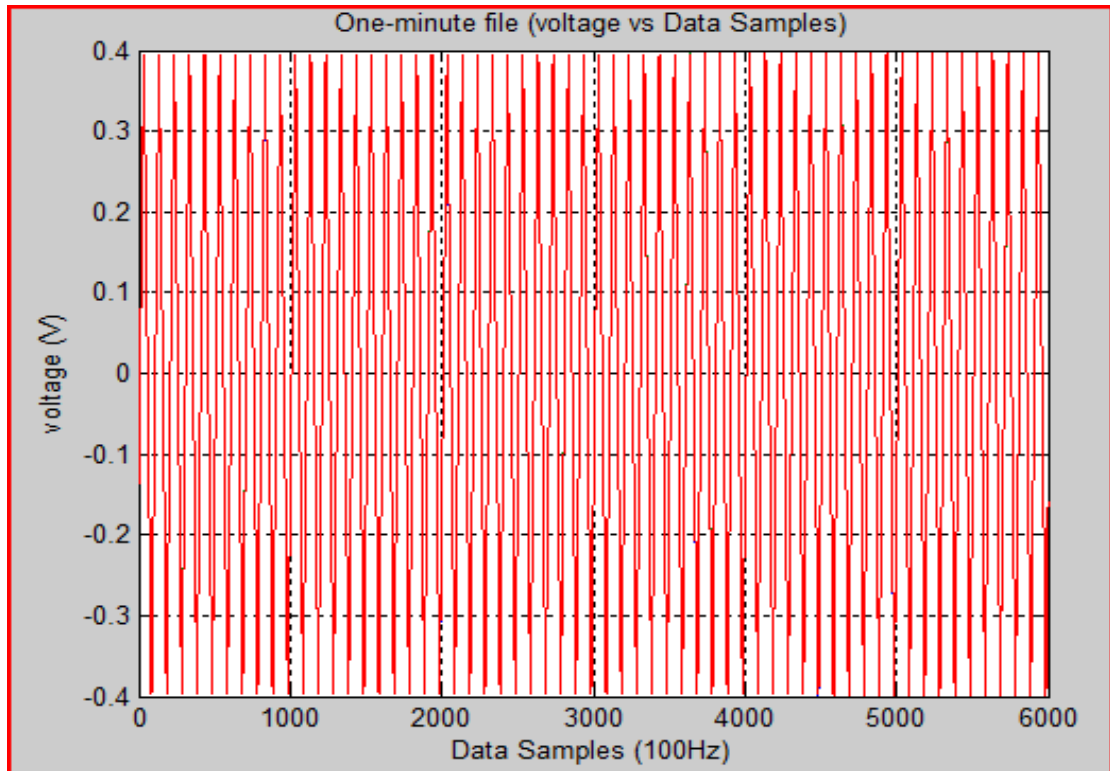


Figure 3-13: A triangular wave sampled by the ADC board, and stored in the submerged unit into one-minute files.

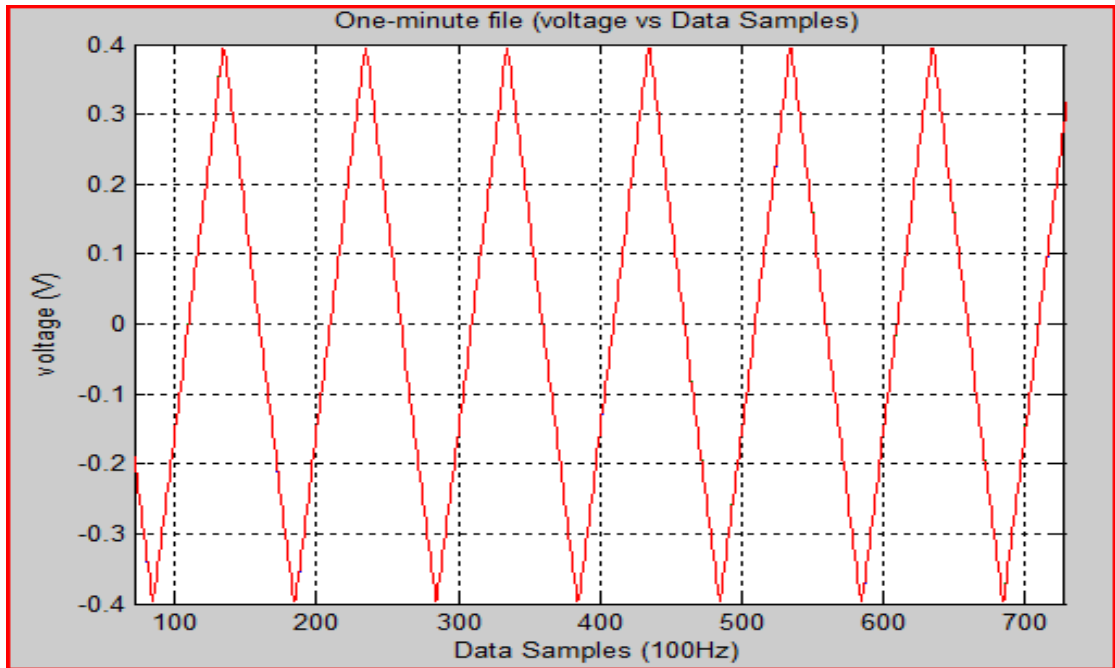


Figure 3-14: Zooming into Figure 3-13 to verify a triangular wave was received.

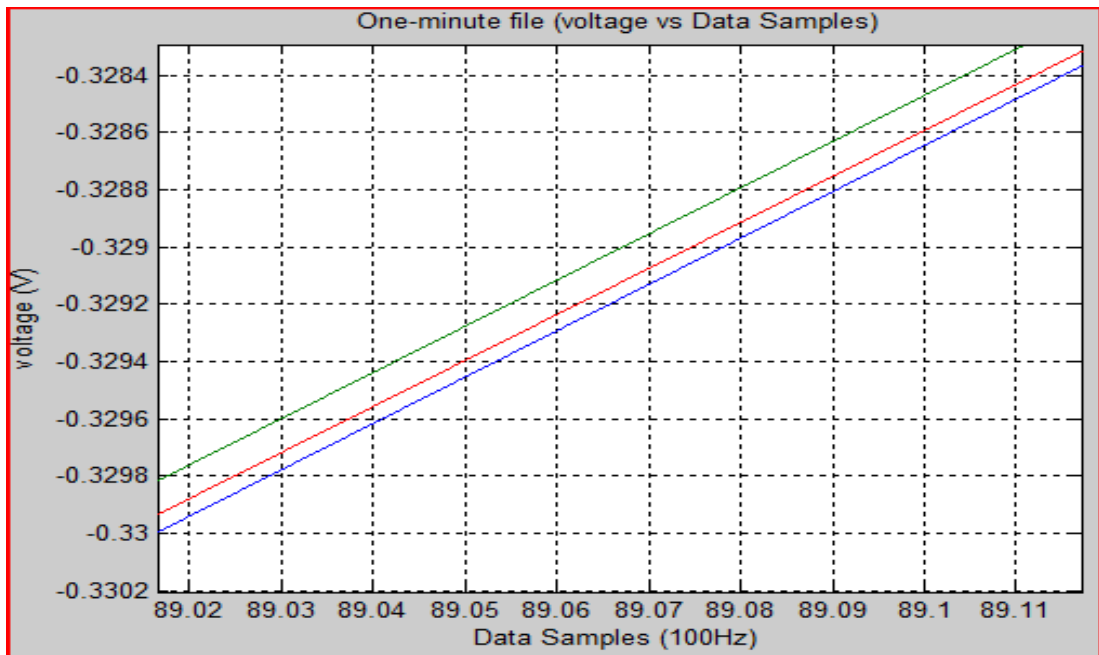


Figure 3-15: Zooming into the figure 3-14 to verify that the three inputs (channels) are received.

3.1.2 PART 2 –SURFACE COMPONENT

The computer used in the surface component must fulfill as a minimum the following requirements:

- a. Linux operating system
- b. data receiving and storing capability
- c. support IRIDIUM communication capability
- d. networking capability (needed for remote monitoring and for Internet access during development)
- e. A minimum of 2 USB ports (one used for the USB/RS-485 connection and one for USB flash drive for storage during development).
- f. serial port for GPS time synchronization

The first five requirements listed above did not represent any impediment in searching the open market for a Single Board Computer (SBC) that could support them. An SBC is a computer built on a single circuit board with microprocessor, memory, input/output and peripheral connections. At the time of this research there were many SBC products available in the market that could fulfill these requirements. Examples of these SBCs are: PC/104 form factor boards, Phidget SBC2, and Raspberry Pi. All of these boards are advertised as low power consumers; however in modern computers, USB is displacing RS-232 from most of its peripheral interface roles, so finding an SBC that supports the serial port communications in conjunction with low power consumption was the decisive factor.

3.1.2.1 SINGLE BOARD COMPUTER (SBC) SELECTION AND GPS TIME

SYNCHRONIZATION OF Son-O-MERMAID SURFACE COMPONENT

Temporal accuracy is part of the foundation of any signal processing data analysis.

Every computer motherboard, with some exceptions, includes an internal clock that

continues to run on battery even when the computer is switched off. On any operating system that is installed this clock is the default time source. Unfortunately, motherboard clocks are anything but accurate and may drift by as much as several seconds or minutes in a day. An efficient solution to this problem is to synchronize the computer to one or more reliable time sources. The most common way to accomplish this is synchronizing the computer time to a few of the many available Network Time Protocol (NTP) servers available in the Internet, but problems will occur if the Internet link becomes unavailable for any significant length of time. A second and more reliable option is to use a local time source such as radio receivers for time signals, which are inexpensive but not too accurate, and atomic clocks which are extremely accurate, but complex and expensive. None of these two options are suitable for Son-O-MERMAID. The first one is unsuitable because this system will be deployed to the open ocean where no Internet connection will be available. The second one is unsuitable because radio receivers would not provide the desired accuracy and are not feasible for the open ocean either. Furthermore an atomic clock is out of the scope of a simple and low cost system. A third option was to use a GPS receiver with pulse-per-second (PPS) output. These receivers are inexpensive, relatively simple, yet capable of providing near microsecond accuracy. This was, by far, the best option for Son-O-MERMAID.

Based on its connection type, two types of GPS receivers could have been used: one that connects via the USB port, or another that connects via the serial port. USB cannot transfer a PULSE which will translate in poor time accuracy, while the serial port can process pulses with handshake inputs via the Universal Asynchronous Receiver/Transmitter (UART). For this reason, the approach followed on Son-O-MERDAID was to use a GPS receiver that connects to the computer via serial port.

3.1.2.1.1 SYNCHRONIZATION OF A COMPUTER’S TIME TO A GPS RECEIVER –A PROOF-OF-CONCEPT

As a proof-of-concept and to learn how to synchronize a computer’s time to a GPS receiver a Garmin 18x LVC GPS receiver shown in Figure 3-16 below and a Desktop computer (PC) Running Linux “Debian 7.0” were used. As this figure shows, the wires from the GPS receiver (TX, RX, Ground and PPS signals) were split and welded to a DB9 female serial connector, and the power wire was attached to the power line of a USB connector which drove 5 Volts from USB port of the PC to power the GPS receiver.



Wire	DB9 female serial	USB	Function
Red	-	Red	+5 Volt power
Black (thick)	-	Black	Power ground
Black (thin)	5	-	Signal ground
Black (loose in cable)	-	-	not used
Yellow	1	-	PPS pulse
White	2	-	TxD from GPS
Green	3	-	RxD to GPS
Shield	-	Shield	Shield

Figure 3-16: Wiring of a Garmin 18x LVC GPS

The process of synchronizing the computer’s time to a GPS receiver will be described in details in section 3.1.2.3, synchronization of Raspberry Pi Model B to Adafruit Ultimate

GPS Breakout board rev 3. However, as a proof-of-concept, a general approach to synchronize a computer's time to a GPS receiver is summarized below:

- a. Customize the hardware as described in figure 3-16 above.
- b. If current Linux kernel is older than version 2.6.34 it must be recompiled to add PPS support. This can be enabled under "Device Drivers" section while running "xconfig" during the Linux kernel recompilation process.
- c. Install "pps-tools" package.
 - `cd /usr/src`
 - `apt-get install git-core`
 - `git clone git://www.linuxpps.org/git/pps-tools pps-tools`
 - `cd /usr/include`
 - `cp /usr/src/pps-tools /timepps.h timepps.h`
 - `cd /usr/src/pps-tools`
 - `make`
- d. Test "pps" by running the following commands:
 - `modprobe 8250`
 - `ldattach 18 /dev/ttyS0`
 - `./ppstest /dev/pps0` [should see some output scrolling down on the screen]
- e. Install "gpsd" (gps daemon).
 - `Apt-get install gpsd` [latest version available will get installed]
- f. Install ntp.
 - `Apt-get install ntp` [latest version available will get installed]
- g. Modify configuration of gpsd.

- Run command: “dpkg-reconfigure gpsd” [with no quotes]
- Change flags to:
 - i. START_DAEMON = true
 - ii. GPSD_OPTIONS= -n
 - iii. DEVICES= /dev/ttyS0
 - iv. USBAUTO=true
- h. Modify “/etc/modules” by adding to the end of the file:
 - pps_ldisc
 - pps_core
- i. Edit “/etc/ntp.conf” file to add PPS support. This file is shown in section 3.1.2.3.2, Synchronization of NTP server with GPSD and PPS.

3.1.2.1.2 COMPUTER SELECTION FOR THE SURFACE UNIT

The configuration steps shown above successfully demonstrated the synchronization of a computer’s time to a GPS receiver. The next step was to find an SBC to replace the PC used in the above configuration to be used in the surface unit. The different computer models investigated are described below:

a. Phidget SBC2

This SBC board used in the submerged component was not an option for the surface component of Son-O-MERMAID since it does not provide the required serial port.

b. fitPC2i

This is a miniature fan less PC based on Atom CPU. It is not exactly an SBC but is extremely small and it had been used on previous tests during the Son-O-MERMAID development. Since we had a few units on hand it was worthwhile attempting to synchronize its system time to the GPS receiver. This computer is advertised providing serial support (Full UART). After testing, it was verified that absolute date and time was received, however the 1-Hz clock signal (PPS) was not received. Further investigation revealed that this computer does not support serial port pin 1, Data Carrier Detect (DCD). This is the pin where the PPS signal is expected. This was confirmed by e-mailing the manufacturer of this product (CompuLab). For reference to RS-232 DB9 pin-out, see figure 3-16 above.

c. Raspberry Pi model B

The Raspberry Pi was the system chosen to handle the computations of Son-O-MERMAID at the surface. These computations include receiving and storing acoustic data for later processing, synchronization of system time to a GPS receiver for time accuracy, and supporting IRIDIUM communications for the transmission of data to a land-based station. The Raspberry Pi is a credit card-sized computer powered by the Broadcom BCM2835 system-on-a-chip (SoC). This SoC includes a 32-bit ARM1176JZFS processor, clocked at 700MHz, and a Videocore IV GPU. This board fulfils the requirements listed for the computer to be used on the surface component of Son-O-MERMAID. Table 5 in the Appendix, section A.2 lists additional Raspberry Pi specifications and figure 3-17 below provides a full view of this SBC.

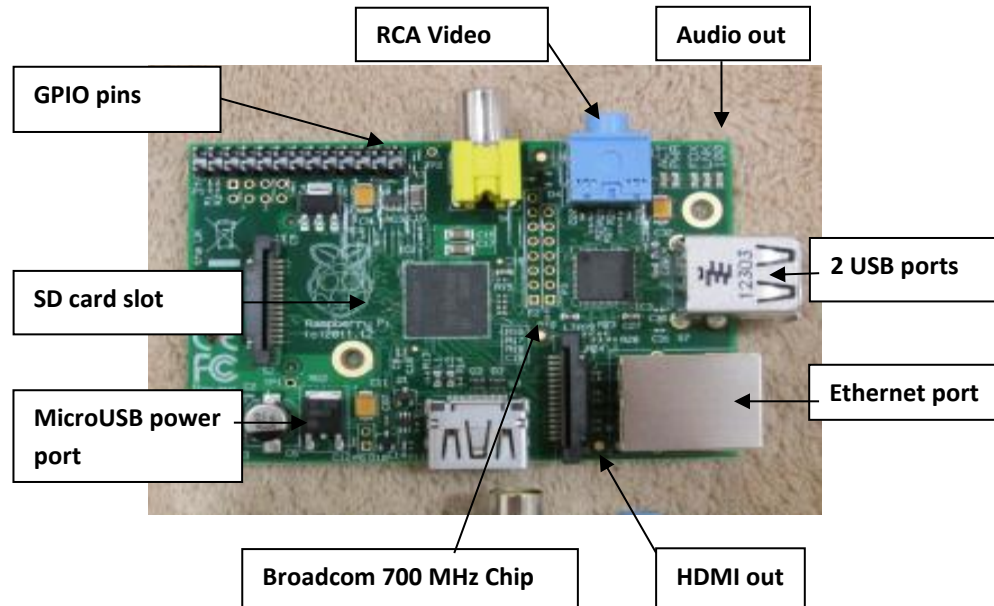


Figure 3-17: Raspberry Pi model B

3.1.2.2 SELECTION OF A GPS RECEIVER TO BE USED AS TIME SOURCE FOR GPS TIME SYNCHRONIZATION

a. Garmin 18x LVC GPS

The GPS 18X shown in figure 3-18 below includes an embedded receiver and antenna. It tracks multiple satellites at a time while providing fast time-to-first-fix, and precise navigating updates once per second. The GPS 18X interfaces to a serial port. The unit accepts TIA-232-F (RS232) level inputs and transmit voltage levels that swing from ground to the positive supply voltage TIA-232-F (RS232) polarity. It implements reverse polarity protection. When tested on a desktop PC with a fully working serial port, this GPS unit worked as expected providing the NMEA serial data and the 1 PPS signal updates each second. However, it would not work on the Raspberry Pi without adding a circuit to modify the output voltage. By design, the Raspberry Pi can only handle voltages on the 0V – 3.3V

range on the UART signals connected to the GPIO pins. The output signals of the Garmin 18x LVC GPS were measured and the following values obtained:

- TX signal = -5.4V, and carries the coarse data (date, time and position information)
- PPS signal = 5.0V

Both of these values were out of processing range of the Raspberry. The Raspberry Pi cannot process negative voltage and anything greater than 3.5V would fry the device. A circuit could have been designed but instead it was decided to search for a GPS receiver available in the market and compatible with the Raspberry Pi that provided the signal with the required voltage.



Figure 3-18: Garmin 18x LVC GPS with USB and RS232 connector

b. Garmin 19x HVS GPS

Another GPS receiver tested was the Garmin 19x HVS GPS. This model works fine with a PC where a serial port is fully implemented but would not work on the raspberry Pi without adding circuitry to modify the minimum output voltage of the GPS. When measured in the lab the following voltage values were observed:

- Coarse data (date, time and position) = 1.9V.

- PPS = 3.0V.

As these values suggest, the PPS signal was received, however the coarse data was never observed on the Raspberry Pi. Apparently the 1.9 V measured on the coarse data would have been interpreted as low (zero) and it never went high (one). The minimum voltage of a high (one) signal on the GPIO pins of the Raspberry Pi must be 2.3V. The Garmin 19x HVS GPS is shown on Figure 3-19 below.



Figure 3-19: Garmin 19x HVS GPS

c. Adafruit Ultimate GPS Breakout board rev 3

The Adafruit Ultimate GPS Breakout board rev 3 was chosen as the GPS receiver for Son-O-MERMAID time synchronization. This board was designed specifically to accomplish this task, namely, to synchronize a Raspberry Pi system time. This GPS receiver is manufactured by the Adafruit Industries and its technical specifications are found in Table 7 in the Appendix, section A.2. Figure 3-20 shows this GPS.

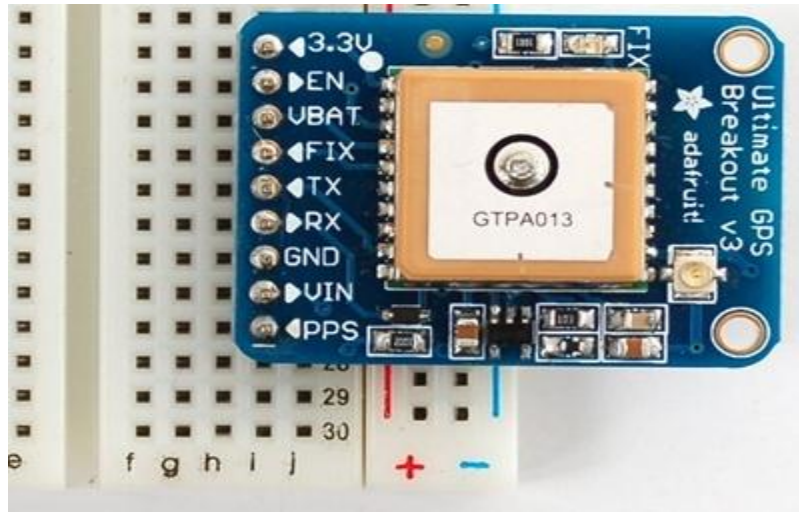


Figure 3-20: Ultimate GPS Breakout Version 3

3.1.1.2.3 TIME SYNCHRONIZATION OF RASPBERRY PI MODEL B TO ADAFRUIT ULTIMATE GPS BREAKOUT BOARD Rev 3

Two approaches were followed to synchronize the system time of the Raspberry Pi to the GPS receiver: the first one uses NTP and kernel drivers, and the second one uses NTP and GPSD applications. Below is a list of the items necessary to build from scratch a Raspberry Pi system with time synchronized to Adafruit Ultimate GPS Breakout board rev 3, and a description of the two synchronization approaches is described in detail below.

Parts list to build a Raspberry Pi system synchronized to GPS

- a. Raspberry Pi Type B.
- b. Adafruit Ultimate GPS Breakout v3: a GPS unit with 1PPS output. The GPIO pins on a Raspberry Pi are only 3.3 volts tolerant with no over-voltage protection which means that a bad connection could fry a pin or worse the device.
- c. wires: five female/male jumpers.
- d. Ethernet Cable: Needed to configure the raspberry pi, once the device is configured Ethernet connection is not needed.

- e. SD Card: Needed for loading the distribution (Operating System) and storing data if desired.
- f. power adapter: the power adapter terminates to a micro USB cable that provides current rating of 850 mA and 5 volts input.
- g. breadboard: used to wire the connections between the Raspberry Pi and the GPS receiver unit.
- h. Raspberry Pi box: an enclosure that offers protection to the Raspberry Pi.
- i. two different distributions were tested: “Occidentalis” and “Raspbian”. The Occidentalis distribution was used in the synchronization of NTP using kernel drivers, and the Raspbian distribution, on the other hand, was used in the synchronization of the NTP server with GPST and PPS. These two distributions are described below in approach one and two respectively.

3.1.2.3.1 APPROACH ONE: SYNCHRONIZATION OF NTP USING KERNEL DRIVERS

Build steps:

A. Creating an SD card with the Operating System:

- a. Find a Windows OS computer with SD card drive.
- b. Download “Occidentalis”, a fork of Raspbian, an OS image for the SD card and follow instructions to install [11].
- c. Unzip the contents of the Zip archive to a .IMG file.
- d. Download the SD card writer program: “Win32DiskImager” from the following site: “<http://sourceforge.net/projects/win32diskimager/>” or google a different site on the web.

- e. Use the Disk Imager to write the OS image to the SD card.
- f. Plug in the SD card to the Raspberry Pi, add peripherals: monitor, mouse, keyboard and Ethernet connection and apply power.

B. Initial Setup

The first time the Raspberry Pi boots up it automatically runs a tool called “raspi-config”. This tool starts before the windowing system and so we have to use the cursor keys and return key to navigate the menu system. The following are the options to select:

- a. Expand rootfs: by default, the Raspberry Pi only uses as much of the SD card as the operating system requires. To fix this so that all the space on the SD card can be used, use the up / down arrow keys to select “expand_rootfs” menu option and hit return. Click return again at the confirmation window
- b. Overscan: this option refers to using the whole screen or monitor. Hit enter to access the overscan option and select “Disable” then hit enter.
- c. Configure_keyboard: select this option and press enter, then select the default option “Generic 105-key (Intl) PC. You will also need to select the keyboard layout: select “Others”, then “English (US)”. When asked about “modifier keys” choose the default option, then choose “No compose key”, the default next option.
- d. Change_password: select “change_pass” and hit enter. After a confirmation screen, you will be prompted to choose a new UNIX user password. Used “**urioce**” as **password**, obviously anything can be used as password.

- e. Change_locale: select “change_locale” and using the space bar key de-select the default selected option, then select “en_US.UTF-8”. The next dialogue window will ask you to choose a default locale; select again “en_US.UTF-8”.
- f. Change_timezone to use UTC time: set the time zone to “Etc”, then select “UTC”.
- g. Select ssh: select the option to enable ssh server. Ignore the remaining options and reboot. Once the device boots completely if necessary log in as (**user: pi, password: urioce**).

C. Free up the UART so that the NMEA “serial” data can be read on the Raspberry Pi

- a. Edit “/boot/cmdline.txt”, along with “/etc/inittab” :

```
$ sudo nano /boot/cmdline.txt
```

```
# Remove these in cmdline.txt:
```

```
# “console=ttyAMA0,115200”
```

```
# and “kgdboc=ttyAMA0,115200”.
```

```
# So it should looks like this:
```

```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4
```

```
elevator=deadline rootwait
```

- b. Edit “etc/inittab” and comment out the last line in this file

```
$ sudo nano /etc/inittab
```

```
#Spawn a getty on Raspberry Pi serial line
```

```
#T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

- c. Power off the Raspberry Pi by running:

\$ sudo poweroff

When only the PWR LED remains lit and the screen goes black, remove the power connector.

D. Wire the Raspberry Pi (RPI) to the GPS receiver (depicted on Figure 3-21 below)

GPS		RPI

RX	→	TXD (pin 8)
TX	→	RXD (pin 10)
PPS	→	GPIO #23 (pin 16) [different from approach two]
GND	→	GND (ping 6)
VIN	→	5V0 (pin 2)

NOTE: If the GPS receiver is being used indoor, make sure the antenna is placed outside the building and wired to the GPS receiver.

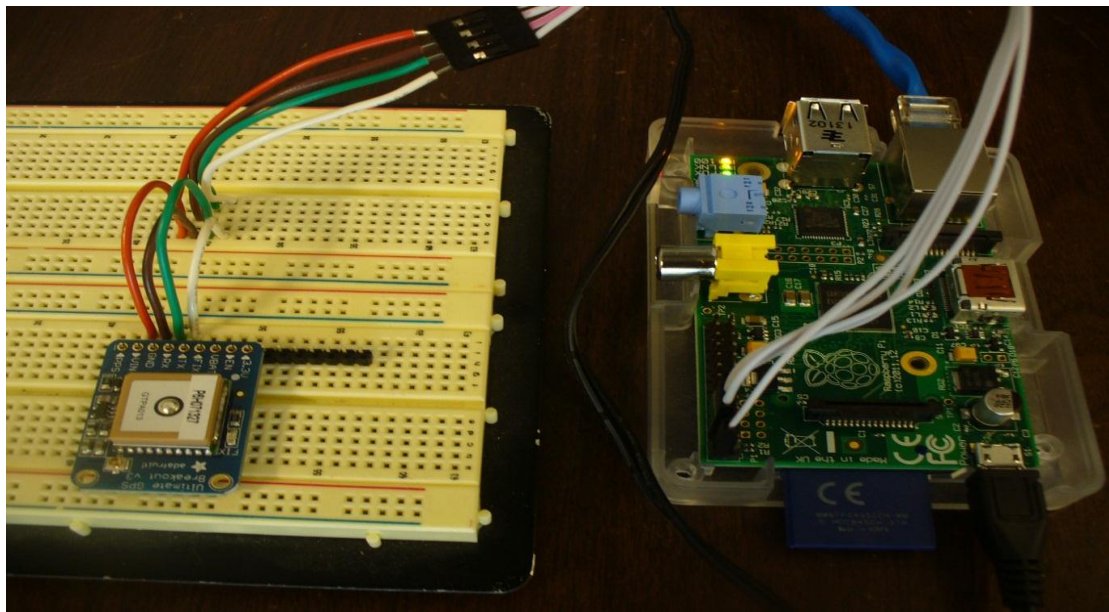


Figure 3-21: Wiring of GPS antenna (left) to a Raspberry Pi for GPS time synchronization

E. Switching the Kernel. Download a precompiled kernel and modules by following the steps below:

a. Download a precompiled kernel and modules:

```
$ sudo apt-get install git
```

```
$ git clone https://github.com/davidk/adafruit-raspberrypi-linux-pps.git
```

b. Back up and copy the kernel image:

```
$ cd adafruit-raspberrypi-linux-pps
```

```
$ sudo mv /boot/kernel.img /boot/kernel.img.orig
```

```
$ sudo cp kernel.img /boot
```

c. Move the modules over:

```
$ sudo mv modules/* /lib/modules
```

d. Add the pps-gpio module to /etc/modules

```
# Run this command in a sub-shell so appending works (quotes are part of  
the command)
```

```
$ sudo sh -c "echo 'pps-gpio' >> /etc/modules"
```

F. Automatically Making Links In “/etc/udev”. By default, our setup isn’t usable by the NTP server since it expects data to be present at specific locations. By adding a few rules to udev, we can automatically make symbolic links (aliases, shortcuts, etc) from our NMEA serial and 1PPS data, to a place where NTP can read and interpret it. Without these rules

the NTP driver would get no data to process. These rules are added by typing the text that appears in Figure 3-22 below.

```
1 $ sudo nano /etc/udev/rules.d/80-gps-to-ntp.rules
2 # Change MODE of ttyAMA0 so it is readable by NTP and provide a symlink to
3 # /dev/gps0
4 KERNEL=="ttyAMA0", SUBSYSTEM=="tty", DRIVER=="", SYMLINK+="gps0", MODE="0666"
5 # Symlink /dev/ppp0 to /dev/gpspps0
6 KERNEL=="ppp0", SUBSYSTEM=="ppp", DRIVER=="", SYMLINK+="gpspps0", MODE="0666"
```

Figure 3-22: Adding rules to “udev” so NTP driver would process the correct data.

- G. Installing and Configuring NTP. The repository version of NTP does not pick up on 1PPS so we need to modify the default install process. This step will take about 45 minutes since we are recompiling from scratch and using special flags or options. To accomplish this, type the text shown in Figure 3-23 below.

```
1 $ sudo apt-get install ntp
2 $ sudo apt-get remove ntp
3 $ sudo apt-get update
4 $ sudo apt-get install libcap-dev
5 $ sudo apt-get install pps-tools
6 $ wget http://www.eecis.udel.edu/~ntp/ntp_spool/ntp4/ntp-dev/ntp-dev-4.2.7p319.tar.gz
7 $ tar -xvf ntp-dev-4.2.7p319.tar.gz
8 $ cd ntp-dev-4.2.7p319
9
10 $ ./configure --prefix=/usr --enable-all-clocks --enable-parse-clocks \
11 --enable-SHM --enable-debugging --sysconfdir=/var/lib/ntp --with-sntp=no \
12 --with-lineditlibs=edit --without-ntpsnmpd --disable-local-libopts \
13 --disable-dependency-tracking && make
14 $ sudo make install
```

Figure 3-23: Step by step instructions to install NTP version 4.2.7p319 with specific set of flags.

H. Edit the “/etc/initd/ntp” file to change the DAEMON line:

```
$ sudo nano /etc/init.d/ntp
```

```
DAEMON=/usr/bin/ntpd
```

```
#DAEMON=/usr/sbin/ntpd
```

I. Edit the “/etc/ntp.conf” file to include the following two lines:

```
server 127.127.20.0 mode 17 minpoll 3 iburst true prefer
```

```
fudge 127.127.20.0 flag1 1 time2 0.496
```

DONE! The process of synchronizing the system time of a Raspberry Pi with a GPS receiver is complete. Reboot the RPI.

3.1.2.3.2 APPROACH TWO: SYNCHRONIZATION OF NTP WERVER WITH GPSD AND PPS

Building Steps

A. Repeat steps (a) to (f) from section (A) in approach one above to create SD card with Operating System with one exception: on step (b) download the “2012-09-18-wheezy-raspbian.zip” Operating System image for the SD card from link [12]. Additionally update the OS by running the following commands:

a. Sudo apt-get update

b. Sudo apt-get dist-upgrade

c. Sudo reboot

B. Repeat steps (B) from approach one above.

C. Repeat steps (B) from approach one above.

D. Wire the Raspberry Pi (RPI) to the GPS receiver (depicted on Figure 3-21 above)

GPS		RPI
RX	→	TXD (pin 8)
TX	→	RXD (pin 10)
PPS	→	GPIO #24 (pin 18)
GND	→	GND (pin 6)
VIN	→	5V0 (pin 2)

E. Configuring PPS for improved precision

- a. At the terminal prompt \$ type without the quotes: `uname -a`. This command will provide the current kernel version of your system similar to the following output: `Linux raspberrypi 3.6.27+ #250 PREEMPT Thu Oct 18 19:03:02 BST 2013 armv61 GNU/Linux`. Record this system information as it will be used to easily differentiate between this kernel and the new kernel and modules that will get installed for PPS support.

- b. On a different system, other than the Raspberry Pi download the kernel image called: `kernel-pps-gpio24.zip` which could be searched on google.com or downloaded from the following link:

https://docs.google.com/file/d/0BznvtPCGqrd3ZEIKZHEtUDRpUEU/edit?usp=drive_web&urp=http://www.satsignal.eu/ntp/Raspberry-Pi-NTP.html&pli=1.

- c. On the same system used on previous step download the kernel modules to enable PPS support. The folder is named: `3.2.27-pps-g965b922-dirty.zip` and can be found at the following link:

https://docs.google.com/file/d/0BznvtPCGqrd3VTZ2TmxFTktYM0E/edit?usp=drive_web. In order to properly download all the modules contained in this folder it

is necessary to click on “File” and then scroll down to “Download”. Kernel and modules come down as zip files and were extracted on the system used for download and then placed on a USB flash drive.

- d. Plug the USB flash drive containing the kernel image and modules to the Raspberry Pi”.
- e. On the Raspberry Pi create a new folder or directory on a desired location called “pps” by executing the command: “*mkdir pps*”.
- f. Type: “*cd pps*” to change directory to the directory created on previous step.
- g. Copy the two directories (kernel-pps-gpio24 and 3.2.27-pps-g965b922-dirty) from the USB flash drive into this “pps” directory.
- h. In the pps/kernel-pps-gpio24 directory there is a file “kernel-pps-gpio24.img. This file must be renamed and moved to the “/boot/” directory, but before we do that a safety copy of the original kernel image should be secured. To accomplish this task this two actions must be taken:
 - i. “*sudo mv /boot/kernel.img /boot/kernel.img.orig*”
 - ii. “*sudo cp kernel-pps-gpio24.img /boot/kernel.img*”
- i. Move the module files into the area where the new kernel expects to find them. We need to see in the “/lib” directory a structure similar to the one below:

```
/lib/modules/3.2.27+  
/lib/modules/3.2.27+/kernel
```

```
/lib/modules/3.2.27+/modules.*
```

```
/lib/modules/3.2.27-cutdown+  
/lib/modules/3.2.27-cutdown+/kernel  
/lib/modules/3.2.27-cutdown+/modules.*
```

```
/lib/modules/3.2.27-pps-g965b922-dirty  
/lib/modules/3.2.27-pps-g965b922-dirty/kernel/  
/lib/modules/3.2.27-pps-g965b922-dirty/modules.*
```

- j. In the unzipped 3.2.27-pps-g965b922-dirty directory we find both the kernel directory and the modules files, so assuming we are now in the pps directory we need to move the required files to the “/lib/modules” directory, and add the pps-gpio module to the module list:

i. “`sudo mv 3.2.27-pps-g965b922-dirty /lib/modules/3.2.27-pps-g965b922-dirty`”

ii. Edit the “/etc/modules” file and add “`pps-gpio`”, without the quotes at the end of the file

iii. “`sudo reboot`”. Verify the changed kernel name at the next login

iv. “`uname -a`”. Verify the command output matches the line below:

```
“Linux raspberrypi 3.2.27- pps-g965b922-dirty    31
```

```
PREEMPT Sat Sep 22 16:30:50 EDT 2012 armv61 GNU/Linux”
```

F. Installing Additional Resources

a. Installing pps-tools

b. Installing GPSD:

i. `sudo apt-get install gpsd gpsd-clients python-gps`

ii. `sudo gpsd /dev/ttyAMA0 -n -F /var/run/gpsd.sock`

c. Installing and verifying PPS is working by running the following commands:

i. “uname -a” gives output: “Linux raspberrypi 3.2.27-pps-g965b922-dirty

#1 PREEMPT Sat Sep 22 16:30:50 EDT 2012 armv6l GNU/Linux”

ii. “dmesg | grep pps” gives entries similar to the following:

Linux version 3.2.27-pps-g965b922-dirty

pps pps0

pps_ldisc

pps pps0: source “/dev/ttyAMA0” added

iii. Sudo aptitude install pps-tools # may take some time

d. Installing NTP

The version of NTP supplied with the Raspberry Pi Linux does not support PPS, therefore it is necessary to re- install NTP from scratch:

- On raspberry Pi create a directory called ntp: “mkdir ntp”
- cd ntp
- sudo apt-get install libcap-dev
- # Get the desired tarball, current or development:
- wget <http://archive.ntp.org/ntp4/ntp-4.2.6p5.tar.gz> # release
- wget <http://archive.ntp.org/ntp4/ntp-dev/ntp-dev-4.2.7p397.tar.gz>
development
- tar xvfz ntp-dev-4.2.7p397.tar.gz
- cd ntp-dev-4.2.7p397
- ./configure --enable-linuxcaps # takes about 15 minutes.
- make # takes about 20 minutes
- sudo make install # puts ntp in /usr/local/bin/ntp; takes about 30-60 seconds.
- Stop and re-start NTP:
 - sudo /etc/init.d/ntp stop
 - sudo /etc/init.d/ntp start
- Updating the NTP configuration file

To get NTP to use the PPS data which is now available to it, we need to add another "refclock" (server) line to the ntp.conf file. The server we use is a type 22 server called the ATOM refclock. It needs a fudge line with flag3 set to 1 to use the kernel-mode PPS data, and we can give it a reference ID of "PPS". We also changed the reference ID of the serial data to "GPS". Note that with a type 22 clock one *must* have another server marked as "prefer". The text below show an updated NTP file:

```
# /etc/ntp.conf, configuration for ntpd; see ntp.conf(5) for help

# Drift file to remember clock rate across restarts
driftfile /var/lib/ntp/ntp.drift

# coarse time ref-clock
server 127.127.28.0 minpoll 4 maxpoll 4

fudge 127.127.28.0 time1 +0.350 refid GPS stratum 15
# time1 time: Specifies the PPS time offset calibration factor, in seconds and
# fraction, with default 0.0.

# Kernel-mode PPS ref-clock for the precise seconds
server 127.127.22.0 minpoll 4 maxpoll 4
fudge 127.127.22.0 flag3 1 refid PPS

# Flag3=1 tells ntp to use the PPS line discipline of the kernel
# refid string: Specifies the driver reference identifier, an ASCII string from
# one to four characters, with default GPS.

# LAN servers
server 192.168.0.3 minpoll 5 maxpoll 5 iburst prefer
server 192.168.0.2 minpoll 5 maxpoll 5 iburst
server 192.168.0.7 minpoll 5 maxpoll 5 iburst
```

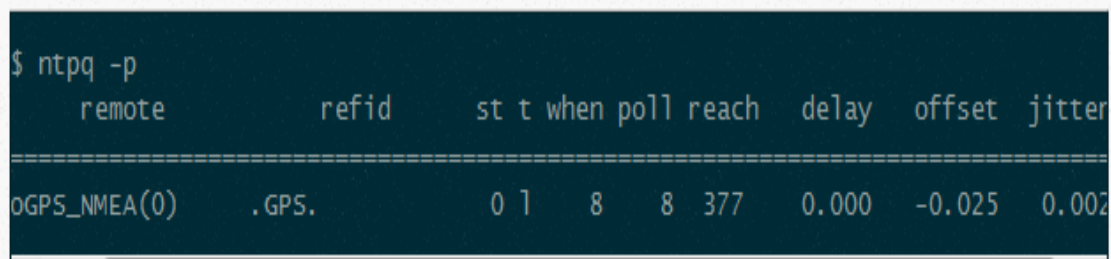
Configuration is Complete!

3.1.2.3.3 VERIFY PERFORMANCE OF TIME SYNCHRONIZATION OPERATIONS

a. Checking the Time on the Raspberry Pi

Once the RPI is backed up, one must wait for the GPS to lock. If we are testing the RPI indoor, we need to place the GPS antenna outside the building with clear view of the sky, when the GPS is locked to a sufficient number of satellites the GPS will blink intermittently. If the GPS unit was turned up for the first time or was turned on after being down for a while, one must wait about 20 minutes for the time synchronization to start taking place. The Raspberry Pi does not have a hardware clock and by default, when the RPI is turned down, it stores its current time before it turns off. When the device is turned back on with no internet connection available its system clock will be set to the time that was last saved on the RPI before it was turned off. However if the RPI is turned on while connected to the internet it will read the current time provided by the network server during the boot process and set its system time to it instead of the last saved on.

Verify that the RPI is connected to the internet, open a terminal and type: “*sudo ntpq -p*” and something similar to the output displayed in Figure 3-24 should appear.



```
$ ntpq -p
      remote           refid      st t when poll reach  delay  offset  jitter
=====
oGPS_NMEA(0)   .GPS.         0 1  8  8 377  0.000 -0.025  0.002
```

Figure 3-24: Partial output of the “*ntpq -p*” command to monitor NTP time synchronization status.

b. System Operation with No Internet connectivity

Both implementations, approaches one and two were fine and the system time was synchronized to the GPS receiver as expected when the Raspberry Pi was connected to the Internet.

Once the connection to the Internet was removed, the configuration described on approach one, synchronization of NTP using kernel drivers, continued receiving GPS time updates and the system maintained its synchronization. However, the configuration described on approach two, Synchronization of NTP server with GPSD and PPS, did not work. It was assumed that GPS receiver was still sending data to the Raspberry Pi, but the configuration of NTP drivers and GPS daemon depended on Internet connectivity and would not work without it. It was verified that when the Internet connection was removed from the board, the PPS signal and coarse data (date, time and position) did not show as being received on the Raspberry Pi.

Based on these results, the configuration of approach two proved to be inefficient and ineffective for our design specification of Son-O-MERMAID (expected to be deployed to the open ocean with no internet connection). Recall from chapter 2 that by design, this was the way NTP was meant to work: to synchronize the clock of computers in a network to a reference time via Internet. The configuration on approach one, on the other hand, was tested to be effective and compliant with our device requirements, and was used in the final implementation of Son-O-MERMAID.

c. Performance comparison between GPS receiver and Internet Time servers

The following illustrates the output of the “ntpq -p” command collected after the Raspberry Pi has been running for 12 minutes. The output shows the performance of four network time servers and a GPS receiver. A comparison between each of the time sources can be established by inspecting the output.

GPS is the source selected.

System synchronized to GPS with 2 microseconds accuracy.

remote	refid	st	t	when	poll	reach	delay	offset	jitter
oGPS_NMEA(0)	.GPS.	0	l	6	8	377	0.000	0.002	0.004
+206.51.211.152	206.51.211.153	4	u	-	64	377	46.224	-2.605	0.719
*clock.xmission.	.GPS.	1	u	18	64	177	91.136	-7.687	1.935
-mail.honeycomb.	204.246.88.88	3	u	9	64	377	41.222	1.565	0.455
+name1.glorb.com	128.174.38.133	2	u	19	64	377	48.687	-10.025	0.916
remote	refid	st	t	when	poll	reach	delay	offset	jitter
oGPS_NMEA(0)	.GPS.	0	l	4	8	377	0.000	0.003	0.004
+206.51.211.152	206.51.211.153	4	u	6	64	377	46.224	-2.605	0.719
*clock.xmission.	.GPS.	1	u	24	64	177	91.136	-7.687	1.935
-mail.honeycomb.	204.246.88.88	3	u	15	64	377	41.222	1.565	0.455
+name1.glorb.com	128.174.38.133	2	u	26	64	377	48.687	-10.025	0.916
remote	refid	st	t	when	poll	reach	delay	offset	jitter
oGPS_NMEA(0)	.GPS.	0	l	3	8	377	0.000	0.002	0.004
+206.51.211.152	206.51.211.153	4	u	13	64	377	46.224	-2.605	0.719
*clock.xmission.	.GPS.	1	u	31	64	177	91.136	-7.687	1.935
-mail.honeycomb.	204.246.88.88	3	u	22	64	377	41.222	1.565	0.455
+name1.glorb.com	128.174.38.133	2	u	32	64	377	48.687	-10.025	0.916
remote	refid	st	t	when	poll	reach	delay	offset	jitter
oGPS_NMEA(0)	.GPS.	0	l	1	8	377	0.000	0.003	0.004
+206.51.211.152	206.51.211.153	4	u	19	64	377	46.224	-2.605	0.719
*clock.xmission.	.GPS.	1	u	37	64	177	91.136	-7.687	1.935
-mail.honeycomb.	204.246.88.88	3	u	28	64	377	41.222	1.565	0.455
+name1.glorb.com	128.174.38.133	2	u	38	64	377	48.687	-10.025	0.916

Figure 3-25: Output of the “ntpq -p” command to monitor NTP time synchronization status.

In summary, the output above indicates that the current time source used to synchronize the system time of Raspberry Pi is the GPS and that the Pulse Per Second is also used. It also indicates that the second and fifth peers in the “remote” column are included in the trustable peer list. Peer number 4th (-mail.honeycomb) on the other hand has been discarded by the cluster algorithm. A complete description of the output follows below:

Columns Definition:

remote: peers specified in the ntp.conf file.

* = current time source.

o = source selected, Pulse Per Second (PPS) used.

+ = source selected, included in final set.

- = source discarded by cluster algorithm.

refid: remote source's synchronization source.

st (stratum): stratum level of the source.

GPS NMEA: stratum 0.

GPS clock transmission: stratum 1.

Other sources: stratum 2, 3 and 4.

t: types available

l = local (such as a GPS).

u = unicast (most common).

when: number of seconds passed since last response.

poll: polling interval, in seconds, for source.

reach: indicates success/failure to reach source; 377 all attempts successful.

delay: indicates the roundtrip time, in milliseconds, to receive a reply.

offset: indicates the time difference, in milliseconds, between the client server and source.

Jitter: indicates the difference, in milliseconds, between two samples.

Additional tests were performed to verify performance accuracy of the Raspberry Pi time synchronization:

With Internet Connection:

- Let system run for couple of hours and verified the delta between GPS source and Raspberry Pi decreases with time as system synchronizes. The delta value is greater at the beginning when the system boots but the offset value decreases as the NTP server starts synchronizing the system time to a time source.

Without Internet Connection:

- Removed Internet connection keeping GPS connection only.
- Changed system date a month behind, a day behind, hours behind, and reboot after each change.
- Verified after each change that date and time updated correctly to current GPS provided values after each reboot within about 3 seconds.

3.1.3 INTERFACE BETWEEN PART 1 AND PART 2

The only parts of this interface that we were concerned with were the wires needed for data transmission. For test bed testing of our prototype, a 2,000 feet CAT5 Ethernet cable was used. Out of the 8 wires available in this cable, only 3 were needed to carry the data via RS-485 (D+, D- and GRD).

3.2 PROTOCOLS AND MEDIA FOR DATA TRANSMISSION IMPLEMENTATION

3.2.1 10BASE-T (Twisted Pair Ethernet)

10Base-T is the IEEE 802.3 clause 21 standard for 10 megabits per second (Mbps) baseband Ethernet over twisted pair cables terminated with RJ-45, with a maximum length of 100 meters per segment. We knew we were limited to a maximum of 100 meters from the start; however we also knew this limitation applied to a standard network in which several computers may be connected and exposed to network congestions, and where a fast data transmission (10 Mbps) is expected by definition. In Son-O-MERMAID only two computers are connected, a baud rate of 38400 bps is sufficient, and data traffic flows in one direction only. Hence no network congestion is anticipated.

To verify whether 10Base-T was an option for Son-O-MERMAID the following operations were performed:

- a. Connected two computers at both ends of a 1000-foot twisted pair Ethernet cable (CAT5).

- b. Ran the “ping” command to see if both computers could reach each other over the network. The following connection settings were modified:

- a. Adjusted the time-out to be the double of the default value. Default is 1 second.
- b. Minimized the size of the ping packet down to 16 bytes. The default is 32.

Without a few exceptions this approach failed. A few times the ping commands were successful in getting a reply from the other computer connected at the other end of the cable, but the majority of the trials failed. The same approach was tested with 2000-feet of CAT5 cable, but as was expected based on previous results, this also failed.

As a result, 10Base-T Ethernet cannot be used in Son-O-MERMAID for data transfer between the lower and upper units.

3.2.2 100BASE-FX ETHERNET OVER FIBER

100Base-FX is the IEEE 802.3 clause 24 & 26 standard, a version of Fast Ethernet over optical fiber. It uses a 1300 nm near-infrared (NIR) light wavelength transmitted via two strands of optical fiber, one for receive (RX) and the other for transmit (TX). The maximum length is 412 meters for half duplex and 2,000 meters for full duplex connections over multi-mode optical fiber.

In order to test this approach the following operations were performed:

- a. Networked two Data Converters model AT-MC101XL via multimode fiber optic cable.
- b. Connected each Data Converter to a computer via CAT5 Ethernet cable.
- c. Executed the “ping” command and both computers successfully reached each other over the network.

As a result, 100Base-FX Ethernet over Fiber could potentially be used in Son-O-MERMAID for data transfer between the lower and upper units. However, no further test was performed with this approach and it was judged not desired for Son-O-MERMAID for two reasons: first, it adds two Data Converters into the system which brings up the cost of the system but more importantly adds power consumption (8.64 Kilo Watts hours (KWh) to the system in one month. This is a significant increase in power consumption. Second, the cost of the system will increase dramatically by adding fiber optic cable, approximately \$15,000 for the cable and termination at both ends. The cable will also need a special coat for protection in the ocean environment. For these reasons, this approach was discarded.

3.2.3 RS-485 PROTOCOL –A SUCCESSFUL AND FINAL APPROACH

The basics of the RS-485 standard indicate that devices can communicate half-duplex on a single pair of wires, plus a ground wire at distances up to 1200 meters which fulfils the Son-O-MERMAID data transfer requirement. To implement this approach two USB to RS-485 converters were acquired. These adapters are powered by the USB port and work reliably with the FT232RL processor chip from FTDI, so it is fully compatible with Linux. To test this approach the following operations were performed:

- a. Wired two “USB to RS-485” converters using three straight connections: D+, D- and GRD on one converter to its corresponding ones on the other converter at the other end of the cable. Three wires of a 2,000 feet CAT5 Ethernet cable were used for this test, and the wiring is depicted in Appendix, section A.3.1.

- b. Plugged the two “USB to RS-485” converters to the USB port of two computes placed at both ends of the cable and executed a basic Python code to successfully send data across the serial ports. Details are provided section 3.3. Figure 3-26 displays the hardware configuration of this approach.

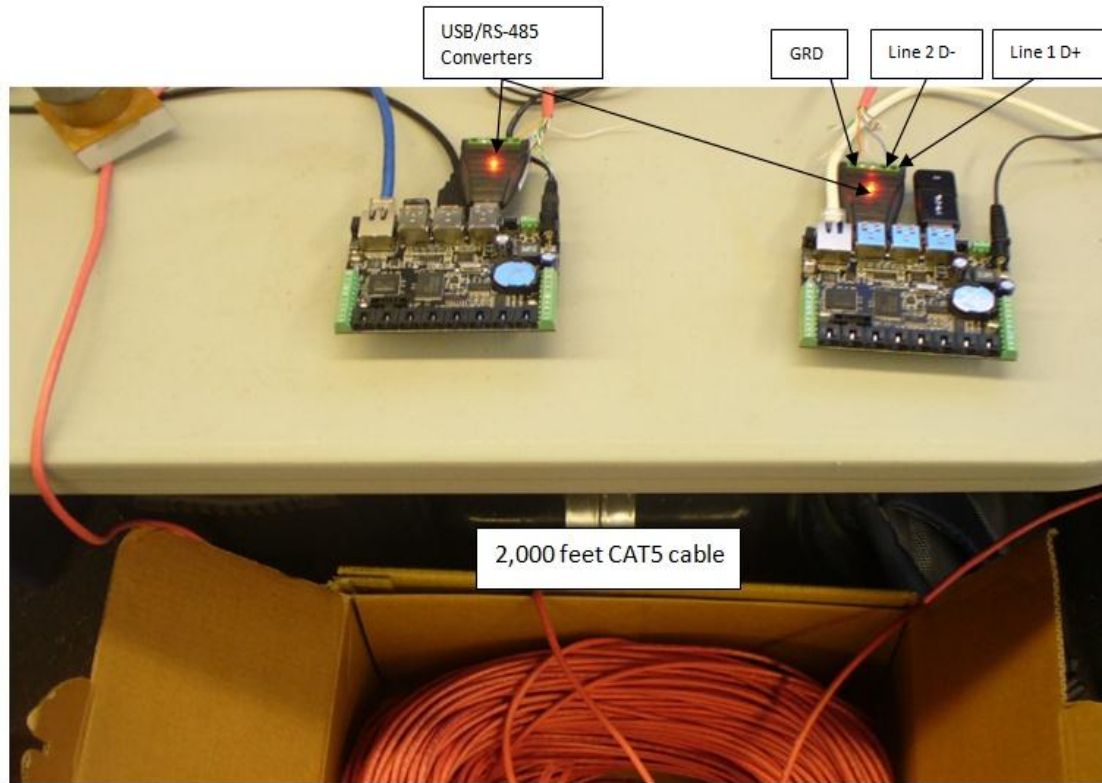


Figure 3-26: Two SBC boards connected via “USB to RS-485” converters at both ends of a 2,000 feet long CAT5 Ethernet cable.

3.3 DATA TELEMETRY

Two approaches were tested for data transmission from the submerged unit to the one on the surface. The first approach was sending complete files, and the second one was a sample by sample transmission. Figure 3-27 below, shows a test bed set up for both approaches.

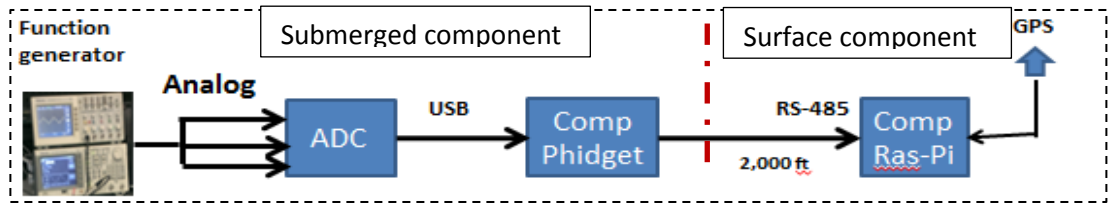


Figure 3-27: Test bed set up for telemetry approaches development

3.3.1 APPROACH ONE: COMPLETE FILE TRANSMISSION ALGORITHM

This algorithm was tested on two Phidget SBC boards running as the sending and receiving units respectively as Figure 3-26 shows above. This algorithm consists of collecting acoustic samples and storing them in one-minute files in the lower unit. Each time one file is completed and closed it gets sent to the surface unit via RS-485 and stored there for future analysis.

To accomplish this task, a solution was found on the “lrzsz” protocol, a cosmetically modified zmodem/ymodem/xmodem package built from the public domain version of Chuck Forsberg’s rzs package. These programs use error correcting protocols ($\{z,x,y\}$ modem) to send (sz, sx, sb) and receive (rz, rx, rb) files over a serial port from a variety of programs running under various operating systems.

In summary to accomplish this file transfer two processes are needed: the “send” process running in the submerged unit, and the “receive” process that runs on the surface unit, and the following steps were necessary:

1. Install “lrzsz” application on both Phidget SBC boards for submerged and for surface units:
 - Connect to the Internet.
 - Open terminal and type: apt-get install lrzsz
2. Set baud rate on both units:

- On terminal type: `stty -F /dev/ttyUSB0 115200`
3. On the “sending” unit run the following shell command:
 - On terminal type: `sz -vv -b fileName.bin > /dev/ttyUSB0 < /dev/ttyUSB0`
 4. On the “receiving” unit run the following shell command:
 - On terminal type: `rz -vv </dev/ttyUSB0 > /dev/ttyUSB0`

To have this file transmission process running automatically at system boot, two scripts were implemented on each unit to send and receive data files respectively. The scripts running in the sending unit are: “sendZ.py” and “sz.sh”, while “recZ.py” and “rz.sh” run in the receiving unit. These scripts can be located in the Appendix, section A.1.2.

Once this algorithm was implemented it was realized that these data were not time-stamped and therefore no meaningful analysis could have been performed on it. The Phidget SBC2 board does not provide a reliable time either, in case the file creation time was to be used for some reference. Therefore, a different algorithm needed to be designed as the next section describes.

To verify that no data drop occurred on file transmission and that the flow and reception of data by the SBC board wasn’t interrupted by the file opening, saving and closing operations three tests were run: first, 2 consecutive one-minute files were appended and plotted using Matlab to verify continuity of data at the boundary of the files. Second, a similar test was run appending and plotting two-minute files; and third, the same test was run with five-minute files. The results were successful for every test. Figure 3-28 – 3-30 below show the results for the first test.

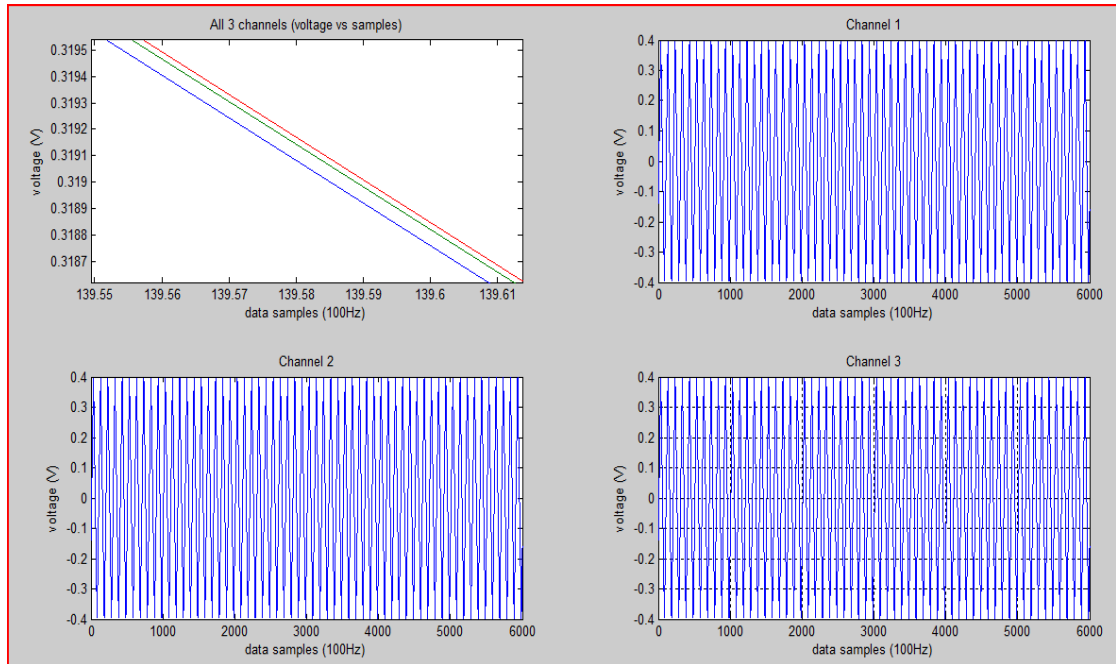


Figure 3-28: Three channels of data contained on each data file

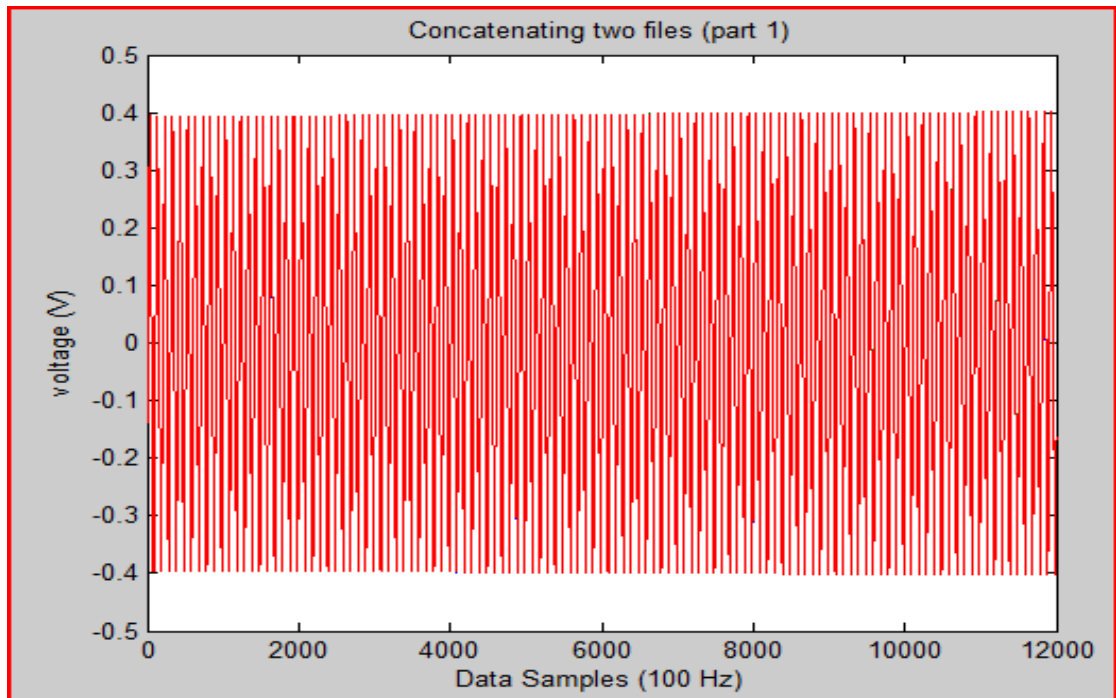


Figure 3-29: Two one-minute files concatenated to demonstrate data continuity between files.

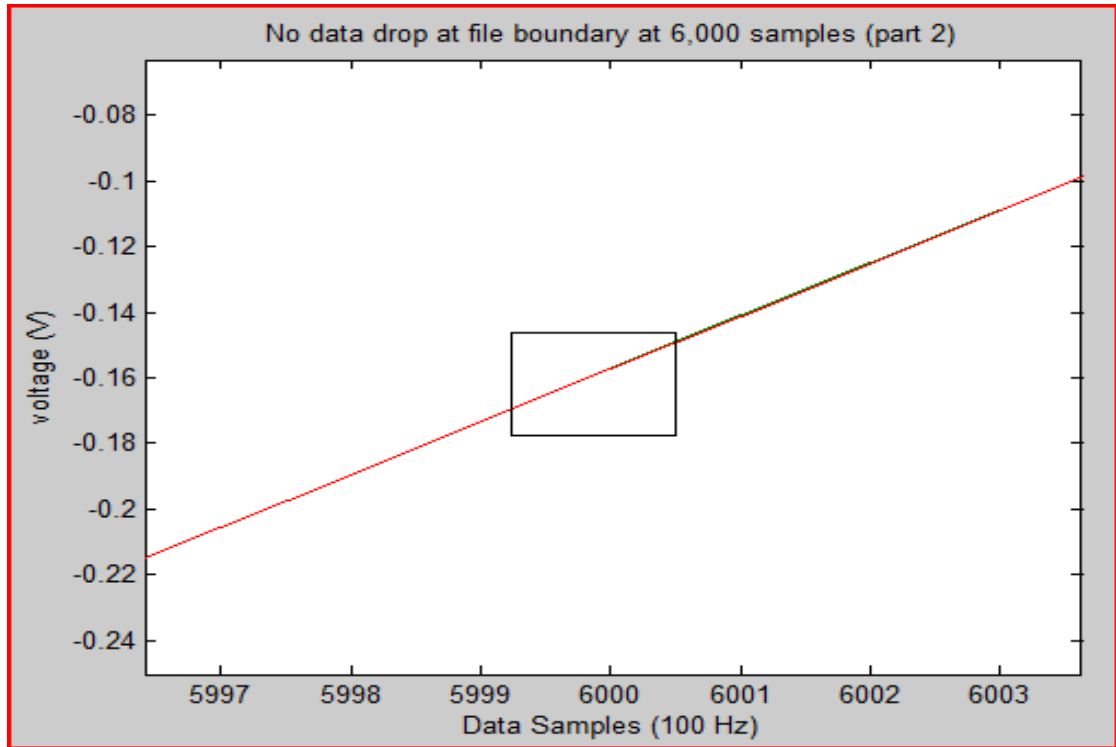


Figure 3-30: Zooming into Figure 3-29 shows no data between files boundary.

3.3.2 APPROACH TWO: SAMPLE BY SAMPLE TRANSMISSION ALGORITHM

This algorithm consists on the submerged SBC collecting acoustic data samples and sending one second of data at a time to the surface unit where data is saved into one-minute files for further analysis. No data is saved on the submerged unit at any time; a variable is used as a place holder to momentarily hold one second of data. This data is then sent to the upper unit and the task repeats over and over until the system runs out of power. This task is also described by the sending algorithm flowchart shown below. The “ironPython” code which performs this task runs on the lower unit computer and is found in the Appendix, section A.1.3.

Since the surface component has synchronized its system time with a GPS receiver, a time stamp is applied to each data file. The time stamp is applied in the following way: first we considered the transmission time of each data sample from the lower to the upper unit as negligible since data travels at speed of light a distance of 2,000 feet. Second, the file creation time is used to timestamp the first data sample on a file, and third a time vector was built to timestamp each data sample on each file in reference to file creation time. This sample by sample transmission approach was adopted in the final implementation of the Son-O-MERMAID prototype being described. This data telemetry solution implemented in this prototype design is described in the flowcharts below summarized in two parts: the sending algorithm which runs in the lower unit implemented in “ironPython” programming language, and the receiving algorithm which runs in the upper unit and is implemented in C programming. The code for both algorithms is included in the Appendix, section A.1.3.

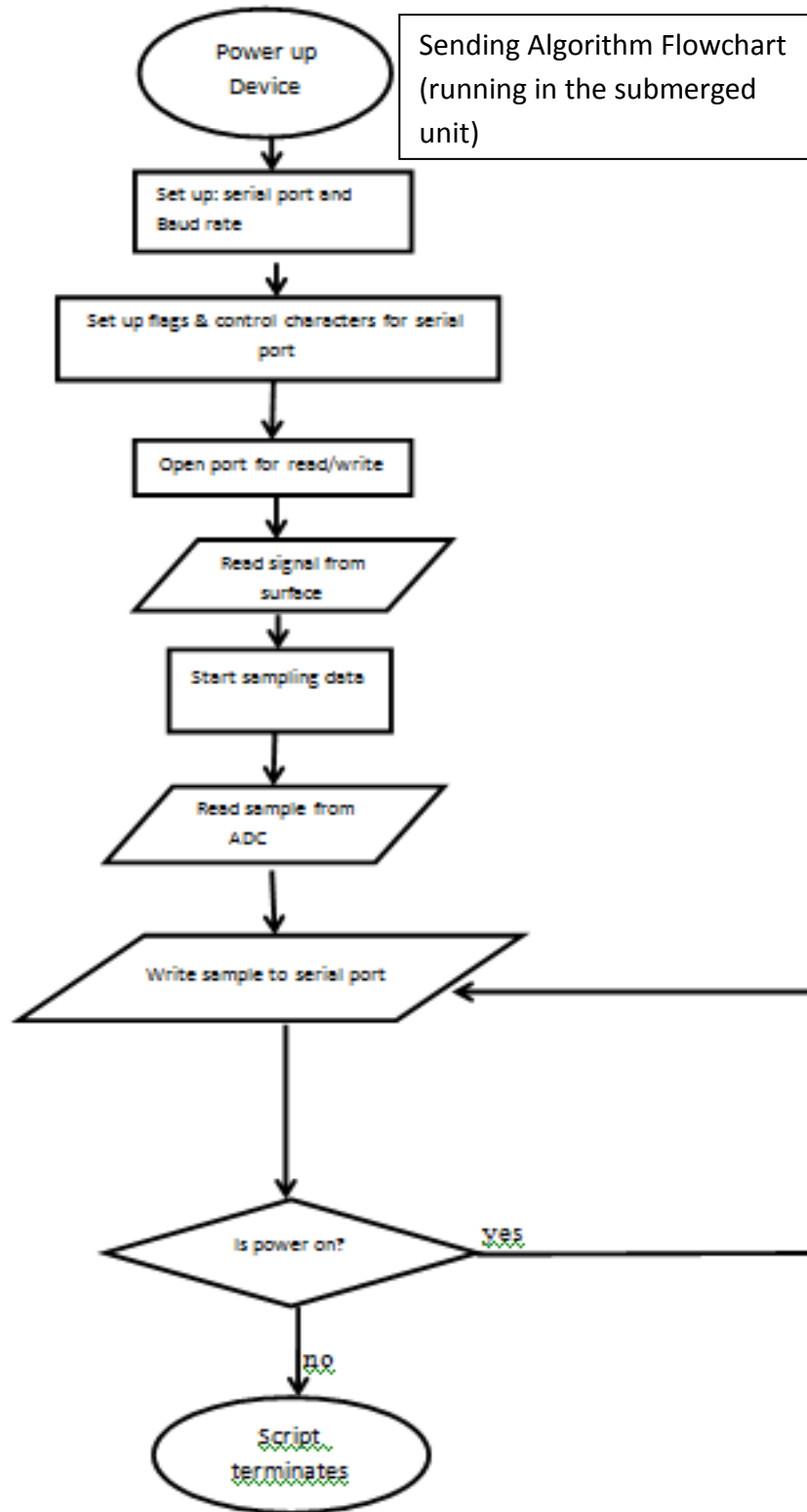
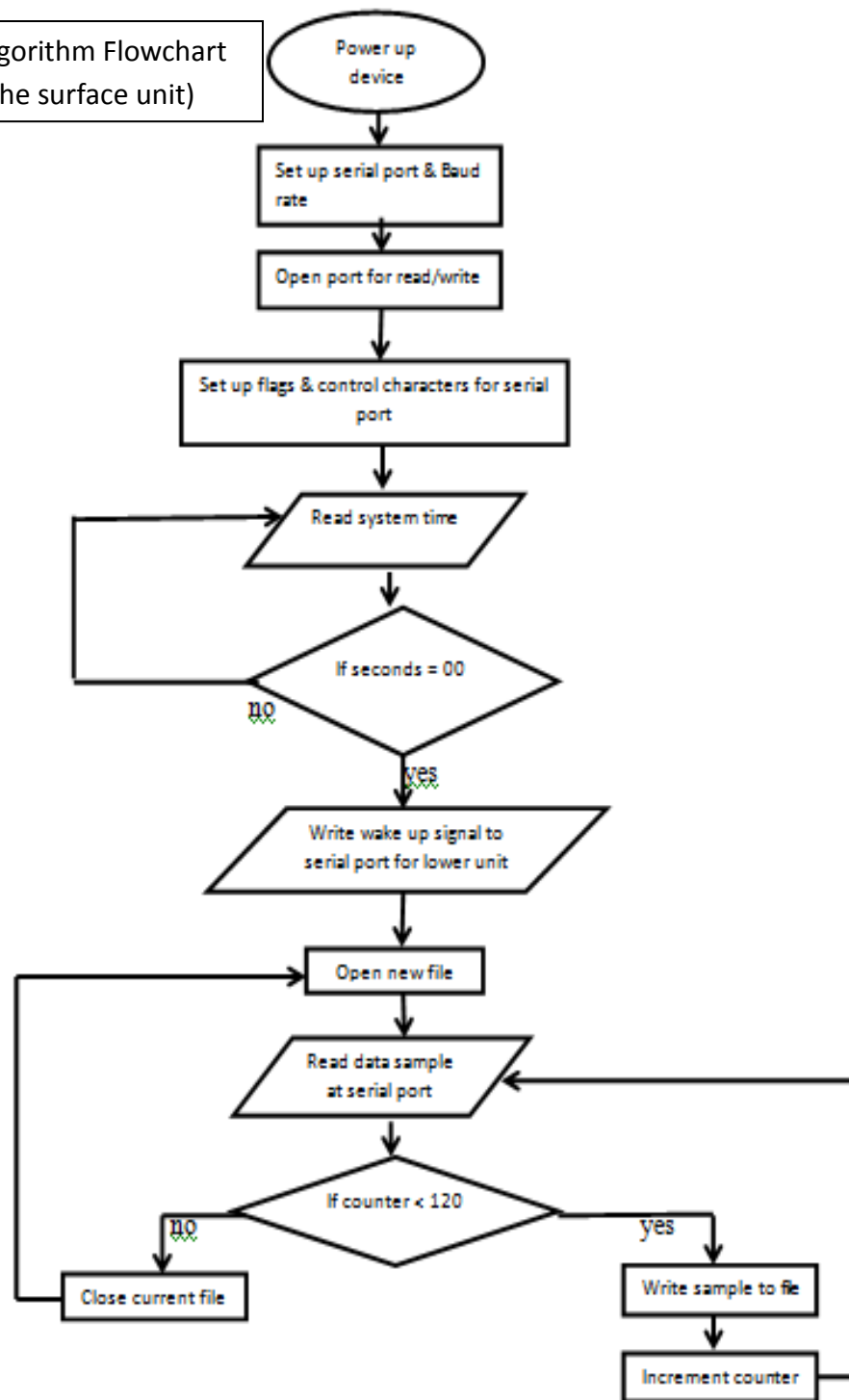


Figure 3-31: Sending Algorithm Flowchart

Receiving Algorithm Flowchart
(running in the surface unit)



NOTE: Program runs forever if power is not interrupted.

Figure 3-32: Receiving Algorithm Flowchart

3.4 POWER CONSUMPTION OF Son-O-MERMAID AND STORAGE SPACE REQUIRED

3.4.1 POWER CONSUMPTION BY THE SUBMERGED UNIT

Once the Son-O-MERMAID prototype is complete, the next planned event is to deploy the prototype in the open ocean for one month test. For that reason, the estimated power consumed in a month was measured. The hydrophones used in this design can operate from 10V to 36V and its output is independent of the input, and the Phidget SBC board can operate from 6V to 15V. In order to measure power consumption of the submerged component, all the hardware (*Circuit interface board, Acquisition board (A/D), Hydrophones and SBC Phidget board*) were connected to the power/data interface board and measured the input power being drawn from the source. Power consumption was measured in two ways: first, a voltage regulator was used within the circuit interface board; and secondly, the voltage regulator was removed. The results are as follow:

Power used by all electronics in the submerged unit (voltage regulator was used within the circuit interface board)

15.1 V input ; 220 mA → 3.322 W

3.322 W * 24 Hrs = 79.728 Wh/day

79.728 * 30 = 2391.84 Wh or 2.39184 KWh (30 days)

Power used by all electronics in the submerged unit (no voltage regulator within the circuit interface board)

15.1 V input ; 170 mA → 2.567 W

2.567 W * 24 Hrs = 61.608 Wh/day

61.608 * 30 = 1848.24 Wh or 1.84824 KWh (30 days)

3.4.2 POWER CONSUMPTION BY THE SURFACE UNIT

The power consumed by all electronics (Raspberry Pi, USB/RS-485 and Ultimate GPS Breakout board) in the surface unit when the device operates in normal conditions is as follows:

5V input; 400mA → 2 Watts (W)

2W * 24 hrs = 48 Wh/day

48Wh/day * 30 days = 1,440Wh or 1.44KWh (one month)

3.4.3 STORAGE SPACE NEEDED

3.4.3.1 COMPLETE FILE TRANSMISSION

Since this approach used binary data (.bin) format at a sampling frequency of 100 Hz and we are using a 16-bit ADC, the necessary space for storage was computed in the following way:

[16 bits → 2 bytes * (100 samples/sec)*(3600/1 hr)*(24 hrs/day)* 30 days]

720,000 bytes/hr (703.125 KB) / hr

17,280,000 bytes/day (16.5 MB) / day

518,400,000 bytes/month 506,250 MB / month (~0.5 GB) / channel

(~1.5 GB) / 3 channels)

In Summary, 2 GB flash drive is sufficient for 30 days of storage with this approach.

3.4.3.2 SAMPLE BY SAMPLE TRANSMISSION

This approach uses ASCII data (.txt) format at a sampling frequency of 100 Hz and we are using a 16-bit ADC. To measure the storage space needed for this approach the size of a one-minute file was used as reference:

[One minute file = 126 KB] →

126 * 60 minutes = 7560 KB/hr →

7560 * 24 hrs = 181440 KB/day 177.2 MB/day →

181440 * 30 days = 5443200 KB/month **5315.625 MB (~5.2 GB) / 3 channels / month**

In Summary, 6 GB flash drive is sufficient for 30 days of storage with this approach.

4 FINDINGS

4.1 HARDWARE REQUIRED FOR BUILDING Son-O-MERMAID PROTOTYPE

4.1.1 SUBMERGED COMPONENT

The submerged component is the part of Son-O-MERMAID that collects the acoustic data. It reads acoustic data from a 3-hydrophone array, then digitizes and samples the data, and sends it to the surface component via RS-485 for storage and further analysis. Figure 3-10 describes this component which includes the following items: power/data interface board, 3-hydrophone array, Analog to Digital Converter, Single Board Computer and a USB/RS-485 converter. The specification characteristics of these items, with the exception of the power/data interface board, can be found in the Appendix, tables A-1, A-2, A-4 and A-6 respectively. This component is configured in the following way:

- An interface board receives power from a local power source (battery) located within the pressure vessel with the rest of the submerged components and distributes power to the 3-hydrophone array, the Analog to Digital Converter and the Single Board Computer.
- The Analog to Digital Converter receives acoustic data from hydrophones via the data lines of the power/data interface board, it digitizes and samples the data and sends 100 samples at a time to the SBC board via USB connection.
- The SBC reads in data samples from ADC, 100 samples at a time and writes them to serial port to send to surface component.

- The USB/RS-485 converter connects to the SBC via USB port and carries out the data between submerged and surface components.

4.1.2 SURFACE COMPONENT

The surface component is the part of Son-O-MERMAID that receives the acoustic data from the submerged component and stores it for further analysis. This component includes the following items: Raspberry Pi Model B, Adafruit Ultimate GPS Breakout board rev 3 and USB/RS-485 converter. The specification characteristics of these items can be found in the Appendix, tables A-5, A-7 and A-6 respectively. This component is configured in the following way:

- The GPS receiver connects to the Raspberry Pi via the GPIO pins in the following way:

GPS		RPI

RX	→	TXD (pin 8)
TX	→	RXD (pin 10)
PPS	→	GPIO #23 (pin 16)
GND	→	GND (pin 6)
VIN	→	5V0 (pin 2)

- The USB/RS-485 converter connects to the Raspberry Pi via USB port and carries out the data between submerged and surface components.

4.1.3 WIRING BETWEEN SUBMERGED AND SURFACE COMPONENTS

During prototype development and testing, 3 out of 8 wires from a 2,000 feet CAT5 Ethernet cable were used to carry the data between the submerged and surface components. These wires were: D+, D- and GRD, which were connected to pin one, pin 2 and

pin 5 respectively in the RS-485 adapter. Wiring details are described in the Appendix, section A.3.1.

4.2 SOFTWARE SOLUTION FOR DATA TELEMETRY OF Son-O-MERMAID

RS-485 Protocol was used to reliably transmit data between submerged and surface components. Two algorithms were developed and verified successfully: complete file transfer and sample by sample transmission. As discussed in section 3.3.2, the sample by sample transfer algorithm was proved to be the final telemetry approach for this Son-O-MERMAID prototype.

4.3 GPS TIME AND DATA SYNCHRONIZATION ACROSS SEVERAL Son-O-MERMAID SYSTEMS

Per design requirement, Son-O-Mermaid must be able to synchronize its time to a GPS receiver, but it also must be able to synchronize within one millisecond to multiple Son-O-Mermaid systems and to monitoring seismic stations on land. To verify this requirement, the deployment of two Son-O-Mermaid systems was simulated as depicted by Figure 4-1 below.

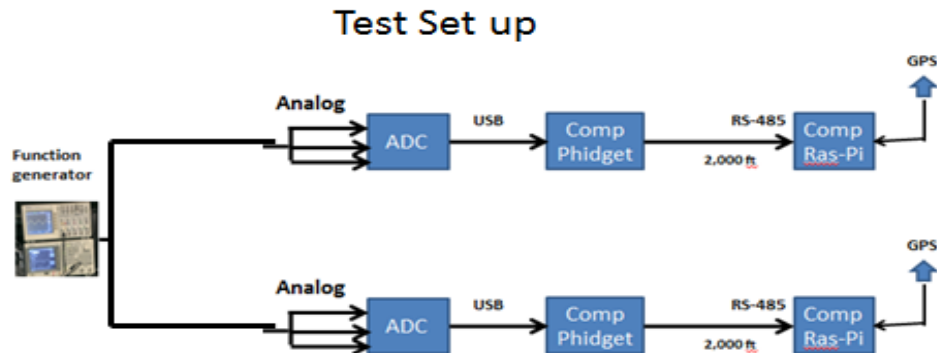


Figure 4-1: Test setup of a simulation of two Son-O-Mermaid systems.

It was expected that data from both systems would overlap since the same input was injected into both systems and the time in both systems was synchronized to a GPS receiver. The test; however, resulted in a failure due to the inability to synchronize the two Son-O-Mermaid systems within one millisecond accuracy. A time offset was observed between systems as shown in Figure 4-2 below.

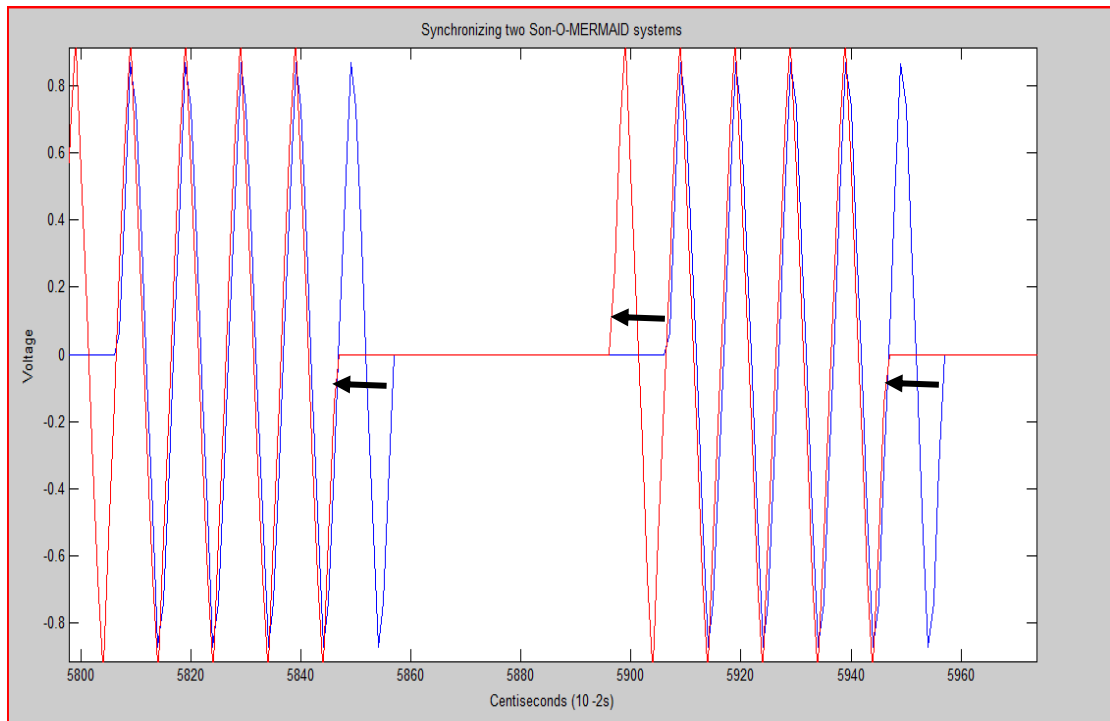


Figure 4-2: Simulation of two Son-O-Mermaid systems both with time synchronized to GPS.

The reason for the failure on synchronizing two or more systems with acceptable precision is unknown. More investigation must be conducted to find the root of the problem and resolve the issue. It is suspected that the reason for the failure is due to the use of several oscillators/clocks (3) in each system: one in the ADC board, one in the submerged computer and one at the surface, all oscillating at different speeds: 100 Hz, 400 MHz and

700MHz respectively; however, this is just an assumption and more investigation is necessary to verify the root cause of the problem.

5 CONCLUSION

The overall objective of this investigation was to build a prototype of a data acquisition and telemetry system for the Son-O-MERMAID, a floating instrument that acts as a freely drifting seismometer that captures acoustic signals caused by distant seismic activity. Prior to the start of this investigation project, some development work had been performed and an unfinished prototype had been fielded without success. As a result of this preliminary development work, two hardware components were carried over and used during this research: the hydrophone array and the ADC board. Specification details for these equipment are found in the Appendix, tables A-1 and A-2 respectively. The remaining hardware for this prototype was selected based on results obtained during testing where cost and power consumption played a major role. The result of this effort is a system that collects acoustic signals at a depth of 2,000 feet, digitizes the data and sends it via RS-485 to a surface computer where it gets stored for further analysis. This surface computer runs a very accurate time system that is synchronized to a GPS receiver and provides data timestamp.

This Son-O-MERMAID prototype is a proof-of-concept, and there are a plethora of directions in which future development could proceed. One of the requirements for this prototype was to provide data timestamp within one millisecond of accuracy. This was necessary in order to synchronize multiple Son-O-Mermaid systems in support of data analysis. This requirement failed and data timestamp was accomplished but with an inconsistent accuracy greater milliseconds. The cause for the failure in data timestamp accuracy is still unknown, but it is believed that part of the problem consists of having three different oscillators/clocks in each system: one in the ADC board, one in the submerged

computer and one in the surface computer, all oscillating at different speeds: 100 Hz, 400 MHz and 700 MHz respectively.

For future recommendations, it is suggested an investigation be conducted to find the root cause of the poor timestamp accuracy. One possible test is to connect an oscilloscope to the data line on each Son-O-Mermaid system to observe and compare the arrival times of data at different points in the system such as: reception of message from the surface unit into the submerged unit which instructs to start sampling data, the arrival of data samples from the ADC into the Phidget SBC at the submerged unit, and the arrival of data samples from Phidget SBC into the Raspberry Pi at the surface unit. This test will provide insights as to where the time synchronization starts degrading. Another recommendation is to eliminate the computer in the submerged unit and connect to the interface power/data board directly from the surface computer. This will require sending power from the surface to the submerged component and performing changes to the proven telemetry algorithms. The advantage of this approach is the reduction of power consumption and possibly improvement of timestamp accuracy by taking one oscillator/clock out of the system. Finally, for future enhancement to the prototype the following questions will need to be addressed after data is stored in the surface unit: how will data analysis be conducted, how long should data be stored for, and how much and how frequent should data be transmitted via IRIDIUM. The answers to these questions will also imply modifications to the telemetry algorithms implemented in this prototype. As a side note, it is worth mentioning that the IRIDIUM communications system was developed as a module on the first Son-O-Mermaid prototype and will be adopted on this new prototype.

APPENDIX

A.1 SOFTWARE IMPLEMENTATION OF Son-O-MERMAID

A.1.1 STORING ACOUSTIC DATA INTO ONE-NIMUTE FILES

File Name: USBDAQ4

```
#!/usr/bin/ipy
```

```
#####
```

```
# PREREQUISITS:
```

```
# In order for this script to operate correctly, it must be placed
```

```
# in a folder along with the following files:
```

```
# a) DAQFlex.dll (DAQFlex API file, version 3.1.0 from Measurement
```

```
# Computing).
```

```
# b) DAQFlex.dll (XML configuration file)
```

```
#####
```

```
# The following script sets the Analog to Digital Converter (ADC)
```

```
# board to collect digitized data at a rate of 100 Hz on three
```

```
# channels (channels 3, 4 and 5).
```

```
# A variable is declared to hold 100 samples (one second) of data
```

```
# within the ADC board, then this data is read in and get stored into
```

```
# one-minute files in the Phidget SBC board in the submerged unit of #
```

```
# Son-O-MERMAID.
```

```
#####
```

```

import os

import struct

import clr

clr.AddReferenceToFile('DAQFlex.dll')

from MeasurementComputing.DAQFlex import *

from time import gmtime, strftime

i = 0

j = 0

LoopAround = True

deviceNumber = 0

deviceNames = DaqDeviceManager.GetDeviceNames(DeviceNameFormat.NameAndSerno)

print deviceNames

deviceName = deviceNames[deviceNumber]
#####

device = DaqDeviceManager.CreateDevice(deviceName)

device.SendMessage("AISCAN:XFRMODE=BLOCKIO") # set transfer mode

                                                    # for analog input scan data

device.SendMessage("AISCAN:LOWCHAN=3") # lowest channel used

device.SendMessage("AISCAN:HIGHCHAN=5") # highest channel used

device.SendMessage("AISCAN:CAL=ENABLE") # enable calibration

device.SendMessage("AISCAN:SCALE=DISABLE")

device.SendMessage("AISCAN:RATE=100") # set the A/D data rate per

# channel for all channels

#device.SendMessage("AI{0}:RANGE=BIP5V")

#device.SendMessage("AI{1}:RANGE=BIP5V")

#device.SendMessage("AI{2}:RANGE=BIP5V")

```

```

device.SendMessage("AI{3}:RANGE=BIP5V") # set analog input range

device.SendMessage("AI{4}:RANGE=BIP5V")

device.SendMessage("AI{5}:RANGE=BIP5V")

#device.SendMessage("AI{6}:RANGE=BIP5V")

#device.SendMessage("AI{7}:RANGE=BIP5V")

#response = device.SendMessage("AI{0}:RANGE=BIP5V")

#value = response.ToValue()

Time = int(float(strftime("%S", gmtime())))

if Time == 0:

    OldTime = 59

else:

    OldTime = Time - 1

N = 1

while os.path.isfile("/media/usb0/testFolder/ipydata" + N.ToString() + ".bin") == True:

    N = N+1

file = open("/media/usb0/testFolder/ipydata" + N.ToString() + ".bin", "wb")

format = "H"

device.SendMessage("AISCAN:SAMPLES=0")

device.SendMessage("AISCAN:START")

while 1:

    scanData = device.ReadScanData(100, 0)

    for i in range (0,100):

```

```

for j in range(0,3):
    ScanData = int(scanData[j,i])
    if ScanData > 65535:
        ScanData = 65535
    elif ScanData < 0:
        ScanData = 0
    data = struct.pack(format, ScanData)#scanData[j,i]
    #file.write(scanData[j,i].ToString("F03") + " ")
    file.write(data)
    #file.write('\n')
if int(float(strftime("%S", gmtime())) == OldTime and LoopAround == True:
    LoopAround == False
file.close()
N = N+1
file = open("/media/usb0/testFolder/ipydata" + N.ToString() + ".bin", "wb")
if int(float(strftime("%S", gmtime())) == Time and LoopAround == False:
    LoopAround = True
file.close()
device.SendMessage("AISCAN:STOP")
DaqDeviceManager.ReleaseDevice(device)

```

A.1.2 TRANSMITTER AND RECEIVER CODE FOR COMPLETE FILE TRANSFER

A.1.2.1 DATA SENDING

Two scripts are used to send data: “sendZ.py” and sz.sh.

A.1.2.1.1 File name: sendZ.py

```
# File name: sendZ.py
```

```
# This script searches for file names in the directory below:
```

```
#"/media/usb0/testFolder" in the sequence 1 through n.
```

```
# Once a file name is found (example: "file2.bin") it sends
```

```
# the previous one (example: "file1.bin").
```

```
#####
```

```
#!/usr/bin/python
```

```
#import serial
```

```
import datetime
```

```
import os
```

```
import sys
```

```
from time import sleep
```

```
now = datetime.datetime.now()
```

```
##### calling system's commands (lrzsz) within python #####
```

```
N = 2
```

```
while True:
```

```
    appendString = ("ipydata" + str(N) + ".bin")
```

```
    # Searching for a "ipydata#.bin" file in
```

```
    #"/home/don/Desktop/out_files" directory
```

```
    if os.path.isfile("/media/usb0/testFolder/" + appendString) == True:
```

```
        print("found " + appendString + " file")
```

```

f_send = ("ipydata" + str(N-1) + ".bin")
print ("Sending file: " + "ipydata" + str(N-1) + ".bin")
##### calling system's commands (lrzsz) within python #####

cmd = './sz.sh '+ f_send
os.system(cmd)

N = N+1

sleep(5)

else:

print(appendString + " file does not exist yet")

sleep (10)

```

A.1.2.1.2 File name: sz.sh

```

#!/bin/bash

# This script initiates a connection with zmodem to send files

cd /media/usb0/testFolder/

stty -F /dev/ttyUSB0 115200

#sz -vv -b ipydata1.txt > /dev/ttyUSB0 < /dev/ttyUSB0

sz -vv -b $1 > /dev/ttyUSB0 < /dev/ttyUSB0

```

A.1.2.2 DATA RECEIVING

Two scripts are used to receive data: "recZ.py" and rz.sh.

A.1.2.2.1 File name: recZ.py

```
# this script receives files serially by lrzsz -ZMODEM
```

```
import os
```

```
import sys
```

```
while True:
```

```
    cmd = './rz.sh'
```

```
    os.system(cmd)
```

A.1.2.2.2 File name: rz.sh

```
#!/bin/bash
```

```
# This script opens a receive connection with zmodem to receive files
```

```
# It is intended to open connection without interruption
```

```
cd /media/usb0/junk/
```

```
stty -F /dev/ttyUSB0 115200
```

```
rz -vv -b -O < /dev/ttyUSB0 > /dev/ttyUSB0
```

A.1.3 SAMPLE BY SAMPLE TRANSFER

A.1.3.1 THE SENDER CODE (Implemented in “ironPython”)

```
#File name: mermaid_send.py
```

```
# This script runs in the submerged (phidget Single Board Computer) component of
```

```
# MERMAID and performs the following tasks:
```

```
# 1) Performs board settings on the ADC board and waits for order from the surface
```

```
# unit to start sampling data at the indicated time.
```

2) Receives order from surface component and pass it to the ADC board so it knows
when to start collecting acoustic data samples.
3) Receives samples of acoustic digitized data from the ADC board (100 samples at
a time) and transmit them via RS-485 to the surface component.

```
#!/usr/bin/ipy
```

```
import clr
```

```
clr.AddReferenceToFile('DAQFlex.dll')
```

```
from MeasurementComputing.DAQFlex import *
```

```
from time import gmtime, strftime, sleep
```

```
import os
```

```
import datetime
```

```
#
```

```
print ("sleep for 30 seconds to allow time to get USB/Serial adapter ready" + '\n')
```

```
print ("This is needed when this script starts at boot when this script" + '\n')
```

```
print ("is added to the rc.local file" + '\n')
```

```
sleep (30)
```

```
# IronPython port settings
```

```
clr.AddReference('System')
```

```
from System import *
```

```
serialPort=IO.Ports.SerialPort('/dev/ttyUSB0')
```

```
serialPort.BaudRate = 38400
```

```
serialPort.DataBits = 8
```

```
serialPort.Open()
```

```
#
```

```

# DAQFlex Analog to Digital Converter (ADC) initialization

deviceNumber = 0

deviceNames = DaqDeviceManager.GetDeviceNames(DeviceNameFormat.NameAndSerno)

print deviceNames

deviceName = deviceNames[deviceNumber]
#####

device = DaqDeviceManager.CreateDevice(deviceName)

device.SendMessage("AISCAN:XFRMODE=BLOCKIO")

device.SendMessage("AISCAN:LOWCHAN=3")

device.SendMessage("AISCAN:HIGHCHAN=5")

device.SendMessage("AISCAN:CAL=ENABLE")

device.SendMessage("AISCAN:SCALE=ENABLE") #enabled for TXT. files
                                         #disabled for BIN. files

# ADC set to 100Hz

device.SendMessage("AISCAN:RATE=100")

# Channels 3, 4, and 5 set to read acoustic data from hydrophones

device.SendMessage("AI{3}:RANGE=BIP5V")

device.SendMessage("AI{4}:RANGE=BIP5V")

device.SendMessage("AI{5}:RANGE=BIP5V")

# placed following two lines here, before serial signal is receive

# to compare time accuracy

# Read serial signal from unit at surface to start sending acoustic samples

readSerial = ""

while (len(readSerial) == 0):

```

```

readSerial = serialPort.ReadLine()

if (len(readSerial) > 0):

    print("Okay to read acoustic data" + '\n')

# Start collecting acoustic data samples in the DAQFlex's buffer

device.SendMessage("AISCAN:SAMPLES=0")

device.SendMessage("AISCAN:START")

#

# Read DAQFlex's buffer and send 100 samples each time via RS-485 to surface unit
while 1:

    dataHolder = "" # variable to hold 100 samples (one second)

    scanData = device.ReadScanData(100, 0)

    for i in range (0,100):

        for j in range(0,3):

            dataHolder = dataHolder + (scanData[j,i].ToString("F03") + " ")

            dataHolder = (dataHolder + ",")

    serialPort.WriteLine ((dataHolder)+ '\n') #writes 100 samples to serial port

    print("size of line written to serial port:" + len(dataHolder).ToString()+ '\t')

    print("TIME: " + datetime.datetime.now().time().isoformat()+ '\n')

#

serialPort.Close()

device.SendMessage("AISCAN:STOP")

DaqDeviceManager.ReleaseDevice(device)

```

A.1.3.2 THE RECEIVER CODE (Implemented in C)

#File name: mermaid_rec.c

/* File name: mermaid_rec.c

* This script runs in the surface component of MERMAID and

* performs the following tasks:

* 1) Sends a signal via the RS-485 line to the ADC board

* to order when to begin sampling acoustic data.

* 2) Receives samples of acoustic digitized data from the lower

* unit (100 samples, XXX bytes at a time.

* 3) Stores acoustic samples into one-minute file. File size

* can be changed by changing the number of iterations in

* loop (line #)

* Read and write from/to multiple binary files in a directory

*/

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <errno.h>

#include <termios.h>

#include <stdarg.h>

#include <stdlib.h>

#include <time.h>

/* baud rate defined in <asm/termbits.h>, which is included by <termios.h> */

```

#define BAUDRATE B38400

#define MODEMDEVICE "/dev/ttyUSB0"

#define _POSIX_SOURCE 1      /* POSIX compliant source */

#define FALSE 0

#define TRUE 1

/*****

main ()

{

    printf("sleeping 2 minutes to allow time to phidget to be ready\n");

    sleep(120);

    int fd, c, res;

    struct termios oldtio, newtio;

    char buf [3110];          //changed from 255

    char ofile[3110];        //changed from 255

    /* Settings to send signal via RS-485 to the Analog to Digital
       Converter (DAQFlex) board to start collecting samples to be
       transmitted via RS-485 to the surface SBC (Raspberry pi */

    time_t nowtime;

    struct tm *ptr_time;

    char buffer[10];

    int seconds = 11;

    /* open modem device for reading and writing */

    fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );

```



```

if (fd < 0) {perror (MODEMDEVICE); exit(-1);}

tcgetattr(fd, &oldtio); /* save current serial port settings */

bzero (&newtio, sizeof(newtio)); /* clear struct for new port settings */

/* set baud rate (38400), 8bit, no parity, 1 stop bit */
newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

/* Raw input */
newtio.c_oflag = 0;

/* ICANON : enable canonical input
 * disable all echo functionality, and don't send signals to calling program */
newtio.c_lflag = ICANON;

/* initializing control characters
 * default found at /usr/include/asm/termbits.h */
newtio.c_cc[VINTR]    = 0;    /* Ctrl-c */
newtio.c_cc[VQUIT]   = 0;    /* Ctrl-\ */
newtio.c_cc[VERASE]  = 0;    /* del */
newtio.c_cc[VKILL]   = 0;    /* @ */
newtio.c_cc[VEOF]    = 4;    /* Ctrl-d */
newtio.c_cc[VTIME]   = 0;    /* inter-character timer unused */
newtio.c_cc[VMIN]    = 1;    /* blocking read until 1 character arrives */
newtio.c_cc[VSWTC]   = 0;    /* '\0' */

```

```

newtio.c_cc[VSTART] = 0; /* Ctrl-q */
newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
newtio.c_cc[VEOL] = 0; /* '\0' */
newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
newtio.c_cc[VEOL2] = 0; /* '\0' */

/* clean modem line and activate the settings for the port */
tcflush(fd, TCIFLUSH);
tcsetattr(fd, TCSANOW, &newtio);

/* Waits for the star of a new minute with "00"
seconds to send signal to DAQFlex to start sampling */
while (seconds!=00) {
    time(&nowtime);
    ptr_time = localtime(&nowtime);
    strftime(buffer, 10, "%S", ptr_time);
    seconds = atoi(buffer);
    printf("%d\n", seconds);
} //end while

/* Sending "S"; a character to wake up the DAQDFlex ADC board

```

which will start reading data samples to be sent via RS-485?

to the surface unit (Raspberry pi)*/

```
res = write(fd, "S\n", 10);
```

```
printf("Sent signal to DAQFlex\n");
```

```
/* terminal settings done, now handle input
```

```
* Infinite loop. Uses a counter called "sample" to iterate twice per second;
```

```
* 120 iterations equals ONE minute file. The counter can be incremented as
```

```
* desired to get a file of a desired size. */
```

```
int fileCtr = 1;
```

```
while (1) {
```

```
    /* Opening the files to store acoustic data samples */
```

```
    sprintf(ofile, "/home/pi/Desktop/piTest/systemAtest2/out_file%d.txt",  
fileCtr++);
```

```
    FILE *fpOut = fopen(ofile, "w");
```

```
    if (!fpOut) {
```

```
        printf("unable to open: %s\n", ofile);
```

```
    }
```

```
    int sample = 0;
```

```
    while (sample < 120) {
```

```
        res = read(fd, buf, 3110);
```

```
        printf("number of bytes read: %d\n", res);
```

```
        // (res-1) to avoid printing '\n' character
```

```

// which produces an empty line

int n = fwrite(buf, 1, (res-1), fpOut);
        printf("number of bytes written: %d\n", n);

if (n != (res-1)) {

printf("output write mismatch (%d versus %d)\n", n, res);

} else {

        fflush(fpOut);

}

sample = sample + 1;

if (sample == 120) {

        printf("End of file reached\n");

        fclose (fpOut);

} //end if

}

}

/* restore the old port settings */

tcsetattr(fd, TCSANOW, &oldtio);

} // end main

```

A.1.4 NTP CONFIGURATION FILE

```
"/etc/ntp.conf"
```

```
$ sudo nano /etc/ntp.conf
```

```
# http://www.eecis.udel.edu/~mills/ntp/html/drivers/driver20.html explains
```

```
# these settings slightly modified, but credit to:
```

```
# Paul Kennedy @ (http://www.raspberrypi.org/phpBB3/viewtopic.
```

```
server 127.127.20.0 mode 17 minpoll 3 iburst true prefer
```

```
fudge 127.127.20.0 flag1 1 time2 0.496
```

For a general setup with an Internet connected Raspberry Pi, leaving these uncommented

is generally considered good practice, but were commented out in Son-O-MERMAID since

there were not needed:

```
#server 0.debian.pool.ntp.org iburst
```

```
#server 1.debian.pool.ntp.org iburst
```

```
#server 2.debian.pool.ntp.org iburst
```

```
#server 3.debian.pool.ntp.org iburst
```

```
##### End of NTP File #####
```

What do the bits in the server line above do?

- 127.127.20.0: Specifies the GPS_NMEA driver.
- mode 17: This sets the line speed (bit 4, dec: 16) to 9600 bps. Additionally, \$GPRMC is processed (bit 0, dec: 1). We get a total sum of 17 when adding the decimal parts together (hence "mode 17").
- minpoll: Minimum polling interval for NTP messages in a power of 2. Here, 3 = 8 seconds.
- iburst: If a server is unreachable, send a burst of eight packets instead of one.
- true: Let the server survive NTP's algorithmic weeding.
- prefer: If we have a choice among good hosts (post-determination, etc), use this one for syncing.

What do the bits in the fudge line above do?

- The fudge options are driver dependent.
- 127.127.20.0: Specify the GPS_NMEA driver.

- flag1 1: Activate PPS API, and process the PPS signals we get.
- time2: Compensate slightly for transmission delays. Instructions for tuning this are located # on the “[driver home page](#)” on the site:

(<http://www.eecis.udel.edu/~mills/ntp/html/drivers/driver20.html>). Specifically, look for 7 # (bit) / 128 (decimal) in the mode section. This web site also presents a complete explanation of the flags used.

A.2 HARDWARE TECHNICAL SPECIFICATION TABLES

Table 1: Hydrophones Specifications

Sensitivity with preamp	Max -165 dB re: 1V μ Pa (562 V/Bar) Min -240 dB re: 1V μ Pa (0.1 V/Bar)
Frequency Response	2 Hz to 30 KHz
Equivalent Input Self Noise	RMS from 1 Hz to 1000 Hz: 78 dB re: 1 μ Pa 0.08 μ Bar
Spectral	54 dB re: 1 μ Pa/ \sqrt Hz @ 10Hz 42 dB re: 1 μ Pa/ \sqrt Hz @ 100Hz 42 dB re: 1 μ Pa/ \sqrt Hz @ 1000Hz
Maximum Operating Depth	10,000 feet (3,048 meters)
Size	2.50 “ length X 0.75” dia.

Table 2: ADC specifications

Analog Input characteristics	
A/D converter type	16-bit A/D converter
Number of channels	8 single-ended
Input configuration	Individual A/D per channel
Sampling method	Simultaneous
Max input voltage	\pm 15 V max (IN to GRD)
Input impedance	100 M Ω
Input ranges	\pm 10V, \pm 5V, \pm 2V, \pm 1V

Sampling rate	0.6 S/s to 50 kS/s, software selectable. Son-O-MERMAID used 100Hz
Throughput	Scan to system memory [(100 kS/s) / (# of channels)]; max of 50 kS/s for any channel]
Digital Input / Output	
Configuration	Independently configured for input or output
Output high voltage	3.8 V min
Output low voltage	0.7 V max
Counter	
Counter type	Event counter
Input type	TTL, rising edge triggered
Resolution	32 bits
Input frequency	1 MHz max
Pulse width	500 ns min
MEMORY	
Data FIFO	32,768 samples, 65,536 bytes
EEPROM	1,024 bytes
POWER	
Supply current	Up to 500 mA
Input power to board from USB source	5 V
Output current	350 mA max (total amount of current that can be sourced from the +5V input power and digital outputs.
ENVIRONMENTAL	
Operating temperature range	0 °C to 70 °C
Storage temperature range	-40 °C to 70 °C
Humidity	0% to 90% non-condensing

Table 3: Fit PC2i specifications

Feature	Specification
CPU	Intel Atom Z530
CPU speed	1.6GHz
Memory	1GB / 2GB DDR2-533 on board
Networking	2 x 1000 BaseT Ethernet
Operating System	Linux Mint
Power	12V single supply, 8-15V tolerant
Power consumption	6W at low CPU load, 8W at full CPU load
Operating temperature	0° to + 45°C with hard disk, 0° to + 70°C with SSD
Serial	RS-232 Full UART
Price per unit	£250

Table 4: Phidget SBC2 product specifications

Feature	Specification
CPU	Samsung S3C2440
CPU speed	400 MHz
NAND Memory Size	512 MiB
SDRAM Size	64 MiB
Boot time	30 s
Networking	10/100Base-T Ethernet
Operating System	Debian Linux
Supply Voltage min/max	6 / 15 V DC
Available External Current	500 mA
Operating Temperature min/max	0°C / 70 °C
Power Consumption Base (w/ Ethernet)	1.2 W
Number of Digital Inputs	8
Digital Input Update Rate	125 samples/s
Price per unit	\$150.00

Table 5: Raspberry Pi Model B specifications

Chip	Broadcom BCM2835 SoC full HD multimedia applications processor
CPU	700 MHz Low Power ARM1176JZ-F Applications Processor
GPU	Dual Core VideoCore IV® Multimedia Co-Processor
Memory	512MB SDRAM
Ethernet	onboard 10/100 Ethernet RJ45 jack
USB 2.0	Dual USB Connector (2 ports)
Video Output	HDMI (rev 1.3 & 1.4) Composite RCA (PAL and NTSC)
Audio Output	3.5mm jack, HDMI
Onboard Storage	SD, MMC, SDIO card slot
Operating System	Linux
Dimensions	8.6cm x 5.4cm x 1.7cm

Table 6: USB to RS-485 Adapter specifications

Feature	Specification
Communication supported	RS-485 and RS-422 capabilities
Processor	FTDI FT232RL
Baud rate	300-921.600bps, auto detection
Power	Port powered from USB
Output voltage	Output voltage: 3.60VCD (between D+ and GND)
Data bits	7, 8
Stop bits	1, 2
Buffer size	128/385 bytes
Serial signals	RS-422: TX-, TX+, RX-, RX+, GND (full duplex) RS-485: D-, D+, GND (half duplex)
Flow control	Automatic. No IRQ conflicts, no IRQs, IO or DMA required
Current draw	Less than 100mA
Operating humidity	5% to 95% - No condensation
Operating Temp	-40°C to 85 °C
Operating System	Operating System supported: Linux, Windows and Mac

Table 7: Ultimate GPS Breakout version 3 features

<p>Satellites: 22 tracking, 66 searching Patch Antenna Size: 15mm x 15mm x 4mm Update rate: 1 to 10 Hz Position Accuracy: 1.8 meters Velocity Accuracy: 0.1 meters/s Warm/cold start: 34 seconds Acquisition sensitivity: -145 dBm Tracking sensitivity: -165 dBm Maximum Altitude for MTK3329: 18,000 meters Maximum Altitude for MTK3339: no limit Maximum Velocity: 515m/s Vin range: 3.0-5.5VDC MTK3329 Operating current: 48mA tracking, 37 mA current draw during navigation MTK3339 Operating current: 25mA tracking, 20 mA current draw during navigation Output: NMEA 0183, 9600 baud default DGPS/WAAS/EGNOS supported FCC E911 compliance and AGPS support (Offline mode : EPO valid up to 14 days) Up to 210 PRN channels Jammer detection and reduction</p>

Multi-path detection and compensation
 Breakout board details:
 Weight (not including coin cell or holder): 8.5g
 Dimensions (not including coin cell or holder): 23mm x 35mm x 8mm / 0.9" x 1.35" x 0.3"
 Includes headers and a CR1220 coin cell holder (soldering required.)

A.3 MISCELLANEOUS

Table 8: RS-485 Wiring

Screw Terminal	RS-485 (1)	RS-485 (2)	Screw Terminal
1	D+	D+	1
2	D-	D-	2
3	Not used	Not used	3
4	Not used	Not used	4
5	GRD	GRD	5

BIBLIOGRAPHY

- [1] F. J. Simons, G. Nolet, J. M. Babcock, R. E. Davis, J. A. Orcutt, A future for drifting seismic networks, *Eos Trans. AGU* 87 (31) (2006) 305 & 307, doi: 10.1029/2006EO310002.
- [2] Urick, Robert J., *Principles of Underwater Sound*. Los Angeles: Peninsula Publishing, 3rd edition, 1983 page 262
- [3] Mills, David L. “Network Time Protocol (NTP) Daemon.” <http://www.eecis.udel.edu/~mills/ntp/html/ntpd.html>. Last updated on 29 November, 2012. Accessed on 15 April, 2014.
- [4] Mills, David L. “Adaptive Hybrid Clock Discipline Algorithm for the Network Time Protocol.” <http://www.eecis.udel.edu/~mills/database/papers/allan.pdf>, page 1
- [5] Mills, David L. University of Delaware. “How NTP Works.” <http://www.eecis.udel.edu/~mills/ntp/html/warp.html> (Last update: 18-Aug-2012 19:04 UTC)
- [6] Mills, David L. “Adaptive Hybrid Clock Discipline Algorithm for the Network Time Protocol.” <http://www.eecis.udel.edu/~mills/database/papers/allan.pdf> , page 3.
- [7] Mills, David L. “Cluster Algorithm.” <http://www.eecis.udel.edu/~mills/ntp/html/cluster.html> (13-Apr-2012 16:35 UTC)
- [8] Mills, D.L. Modelling and analysis of computer network clocks. Electrical engineering Department Report 92-5-2, University of Delaware, May 1992, 29 pp.
- [9] Mills, D.L. Clock discipline algorithms for the Network Time Protocol Version 4. Electrical Engineering Department Report 97-3-3, University of Delaware, March 1997
- [10] Phidget SBC2 User Guide Page. http://www.phidgets.com/docs/1072_User_Guide

- [11] Raspberry Pi Distribution: Occidentalis v0.2.
<https://learn.adafruit.com/adafruit-raspberry-pi-educational-linux-distro/>
- [12] Raspberry Pi Distribution: weezy-Raspbian.
<http://ftp.gnome.org/mirror/raspberrypi/images/raspbian/2012-09-18-wheezy-raspbian/>