INTEGRATING ACOUSTIC COMMUNICATIONS
WITH UNDERWATER AUTONOMY

by

CONNOR LENOIR HARPER WEBB
AIJUN SONG, COMMITTEE CHAIR
TODD FREEBORN
XIAOYAN HONG

A THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Electrical and Computer Engineering in the Graduate School of The University of Alabama

TUSCALOOSA, ALABAMA

2022
ABSTRACT

Due to the properties of the underwater environment, the most reliable form of long-distance communication utilized by autonomous underwater vehicles is acoustic communications. This thesis explores the integration of acoustic communications with the autonomy software of an autonomous underwater vehicle. The integration of these two subjects transforms the autonomous underwater vehicle into a mobile platform in an underwater acoustic communications network. A software defined acoustic modem was installed as a payload to an autonomous underwater vehicle, which runs MOOS-IvP as the autonomy software. A group of MOOS applications was created to integrate the acoustic modem with the user computer. First, the Unetstack API, used on the modem as an interface, was used to inform the modem of its location based upon sensor information obtained from the vehicle main computer. Second, additional applications were created to send and receive acoustic commands to control the movement of the vehicle, such as aborting the current mission or switching between different mission tracks. Similarly, the Unetstack API was used send and receive commands that controlled the signal power level of an acoustic transmission remotely. Finally, the first steps were made in acoustic networking by creating a MOOS application that forwards information between different vehicles. The applications were used in multiple field experiments at Lake Tuscaloosa. One experiment involved two autonomous underwater vehicles to form a three-node underwater acoustic network.
DEDICATION

I dedicate this thesis to my parents.
# LIST OF ABBREVIATIONS AND SYMBOLS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UWSN</td>
<td>Underwater Wireless Sensor Network</td>
</tr>
<tr>
<td>ROV</td>
<td>Remote Operated Vehicle</td>
</tr>
<tr>
<td>FSK</td>
<td>Frequency Shift Keying</td>
</tr>
<tr>
<td>QAM</td>
<td>Quadrature Amplitude Modulation</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency-Division Multiplexing</td>
</tr>
<tr>
<td>DFE</td>
<td>Decision Feedback Equalizer</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase Locked Loop</td>
</tr>
<tr>
<td>ISI</td>
<td>Intersymbol Interference</td>
</tr>
<tr>
<td>TRM</td>
<td>Time Reversal Mirror</td>
</tr>
<tr>
<td>MCM</td>
<td>Multicarrier Modulation</td>
</tr>
<tr>
<td>AUV</td>
<td>Autonomous Underwater Vehicle</td>
</tr>
<tr>
<td>MOOS</td>
<td>Mission Oriented Operating Suite</td>
</tr>
<tr>
<td>IvP</td>
<td>Interval Programming</td>
</tr>
<tr>
<td>UAN</td>
<td>Underwater Acoustic Networks</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>BER</td>
<td>Bit Error Rate</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Position System</td>
</tr>
<tr>
<td>MACA-MCP</td>
<td>Multiple Access with Collision Avoidance – Multiple Channels and Positioning</td>
</tr>
<tr>
<td>UANET</td>
<td>Underwater Ad-hoc Network</td>
</tr>
<tr>
<td>UWSN</td>
<td>Underwater Sensor Network</td>
</tr>
</tbody>
</table>

iv
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOF</td>
<td>Degree of Freedom</td>
</tr>
<tr>
<td>MBFSK</td>
<td>Modified Binary Frequency Shift Keying</td>
</tr>
<tr>
<td>CCL</td>
<td>Compact Control Language</td>
</tr>
<tr>
<td>WHOI</td>
<td>Woods Hole Oceanographic Institute</td>
</tr>
<tr>
<td>REMUS</td>
<td>Remote Environmental Monitoring Units</td>
</tr>
<tr>
<td>MLO</td>
<td>Mine-like Objects</td>
</tr>
<tr>
<td>ASV</td>
<td>Autonomous Surface Vehicle</td>
</tr>
<tr>
<td>CCU</td>
<td>Command and Control Units</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>C2</td>
<td>Command and Control</td>
</tr>
<tr>
<td>DCCL</td>
<td>Dynamic Command and Control Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>UVC</td>
<td>Underwater Vehicle Console</td>
</tr>
<tr>
<td>NMEA</td>
<td>National Marine Electronics Association</td>
</tr>
<tr>
<td>MOOSDB</td>
<td>Mission Oriented Operating Suite Database</td>
</tr>
<tr>
<td>DVL</td>
<td>Doppler Velocity Logger</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>dB</td>
<td>Decibels</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>AL</td>
<td>Alabama</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

First and foremost, I would like to acknowledge my parents, my grandparents, and my late great-grandparents. Due to your unending love and support, you have given me the strength I needed to complete work I have done. To those who passed before I could finish this, I thank you and miss you. Without all of you, none of what I have done would be possible.

I would also like to acknowledge Dr. Song, my advisor and committee chair, who has been an excellent supervisor of the research work. His kindness and willingness to teach have provided the necessary tools to complete this work. The other committee members, Dr. Hong and Dr. Freeborn, have provided excellent advice and feedback throughout the process, for which I greatly appreciate.

Third, I would like to acknowledge all of the other students in our research group. To all the students, I cannot repay you for the grueling hot and humid hours and the physical exertion required for field work at Lake Tuscaloosa or Lake Palmer, so I can really only give my thanks and appreciation. To the other graduate students, Joshua, Qiang, Aswanth, Murtaza, and Daniel thank you for helping navigate graduate school, the work involved in research, and the writing skills involved with writing a thesis. To the undergraduate students, Anika, Brodie, and Cassidy, thank you for providing the necessary manpower for doing this research work.

Fourth, I would like to thank my friends, Johnathan, Jacob, Will, Matt, and Trey, who have each given me a person to talk to and provide a needed respite from my work. Also, I would like to thank my friends from tennis and Bible study, Coach Bill, Scott, and Landon, for their years and spiritual and emotional development. Coach Bill, I wish you were still here to see
this today. To all those who have helped along the way, but I have neglected to mention, thank you.
CONTENTS

ABSTRACT ................................................................................................................... ii

DEDICATION ............................................................................................................... iii

LIST OF ABBREVIATIONS AND SYMBOLS .............................................................. iv

ACKNOWLEDGMENTS ............................................................................................... vi

LIST OF TABLES ......................................................................................................... x

LIST OF FIGURES ....................................................................................................... xi

CHAPTER 1 INTRODUCTION AND LITERATURE REVIEW ......................................... 1
  Section 1.1: The Importance, Challenges, and State-of-the-Art of Underwater Acoustic
  Communications ....................................................................................................... 1
  Section 1.2: Contributions of this Thesis .................................................................... 4
  Section 1.3: Literature Review ................................................................................... 5
  Section 1.4: Introduction to Main Efforts ................................................................... 10
  Section 1.5: Organization ......................................................................................... 13

CHAPTER 2 INTEGRATION OF THE FRONTSEAT AND BACKSEAT
  COMPUTERS OF THE IVER3 AUV ............................................................................ 15

CHAPTER 3 INTEGRATION OF THE IVER3 WITH AN ACOUSTIC MODEM .............. 24

CHAPTER 4 ACOUSTIC CONTROL OF THE IVER3 AUV .......................................... 36
  Section 4.1: Introduction ........................................................................................... 36
  Section 4.2: Using the Unetstack API to Remotely Transmit Datagram Messages ....... 37
  Section 4.3: Using the Unetstack API to Remotely Receive Datagram Messages ......... 38
  Section 4.4: Acoustic Control of the AUV to Abort Current Mission ......................... 39
Section 4.5: Acoustic Control of the Modem Signal Power Level..........................42
Section 4.6: Acoustic Control of the AUV to Change Current Mission..................46
Section 4.7: Acoustic Inquires for Information from the AUV.............................50
Section 4.8: Forwarding Information through Underwater Acoustic Networks........53

CHAPTER 5 CONCLUSION....................................................................................63
REFERENCES ......................................................................................................66
APPENDIX A FILES FOR THE INTRODUCTION TO MOOS-IvP.........................69
APPENDIX B FILES FOR iModemComms ............................................................74
APPENDIX C FILES FOR iModemAbort ..............................................................95
APPENDIX D FILES FOR iModemPower ............................................................102
APPENDIX E FILES FOR iModemDeploy .........................................................117
APPENDIX F FILES FOR iModemInfo ..............................................................128
APPENDIX G FILES FOR iModemForward .......................................................137
LIST OF TABLES

2.1. NMEA sentence format for an $OMS sentence .............................................16

3.1. Structure of the $C NMEA sentence ..................................................................33

4.1. List of the IP addresses and node addresses in the laboratory and Lake Palmer field experiment .................................................................59

4.2 Variables and final values changed by $PokeDB .................................................60
LIST OF FIGURES

2.1. The Iver3 vehicle ........................................................................................................15
2.2. Architecture of the MOOS community ......................................................................17
2.3. Frontseat and backseat connected with serial port ..................................................18
2.4. Path of the frontseat mission ....................................................................................20
2.5. Path of the backseat mission ....................................................................................21
2.6. Comparison of the AUV tracks in missions 3 and 4 ...................................................21
2.7. Comparison of the AUV tracks in missions 3 and 6 ...................................................22
2.8. Annotated GPS track of Lake Palmer mission 6 .......................................................23
3.1. Acoustic transmitter installed on topside of AUV ......................................................24
3.2. Interaction of Unetstack agents with ignored script request ....................................28
3.3. Representation of agent interactions with new shell agent .....................................29
3.4. Diagram of Ecomapper-Subnero system integration ................................................31
3.5. Recorded vehicle depth and transmission location with depth issue ......................32
3.6. Recorded vehicle depth and transmission location after correction .......................34
4.1. Experimental setup for testing acoustic control .......................................................36
4.2. Flowchart for iModemAbort ....................................................................................40
4.3. Comparison of original and aborted mission tracks ...............................................41
4.4. Flowchart for iModemPower ..................................................................................43
4.5. Signal power level of the modem during experiment ..............................................45
4.6. Flowchart for iModemDeploy ................................................................................47
4.7. Comparison of original frontseat mission and mission modified by iModemDeploy ..........48
4.8. Annotated track of the AUV modified by iModemDeploy ........................................49
4.9. Flowchart for iModemInfo .........................................................................................50
4.10. Recorded mission track of the AUV during experiment ...........................................52
4.11. Output and response from the interrogation program .................................................53
4.12. Method for forwarding information across the network ...........................................53
4.13. Receiving flowchart for iModemForward ..................................................................56
4.14. Forwarding flowchart for iModemForward .................................................................57
4.15. Laboratory experiment setup in water tank .................................................................58
4.16. Results of the laboratory experiment .........................................................................60
4.17. Output from iModemForward field test ....................................................................61
CHAPTER 1
INTRODUCTION AND LITERATURE REVIEW

Section 1.1: The Importance, Challenges, and State-of-the-Art of Underwater Acoustic Communications

Underwater acoustic communications is an increasingly important field of research because of a “growing interest in monitoring aqueous environments for scientific exploration, commercial exploitation, and protection from attack” [1]. The growing interest is partially fueled by the size of the ocean, which “[covers] about two-thirds of … the Earth” [1]. In its study of the use of Underwater Wireless Sensor Networks (UWSNs), [1] details multiple examples of different applications that involve UWSNs that use acoustic communications. One such application is submarine detection. Due to the high level of technology involved in the dampening of the acoustic signals of military submarines, the submarine can only be detected if the acoustic sensor is in close proximity to the submarine. Second, UWSNs with acoustic transmitters are useful in ocean monitoring. Typically, remote operated vehicles (ROVs) are tethered to a controller at the surface, which can easily be damaged causing a loss in the control of the ROV. By utilizing acoustic communications, the UWSN can operate without the constraint of the tether cable. According to [1], there are many advantages of underwater acoustic communications, especially as it relates to mobile UWSNs. Most importantly, acoustic communications is most widely used due to the simple fact that “electromagnetic waves cannot propagate over long distance in underwater environments.” The same issue is prevalent in optical communication as well due to optical signals being
“strongly scattered and absorbed underwater” [2]. There are two primary benefits to mobile UWSNs. First, if a large number of mobile sensors are used in the UWSN that is deployed in the current of a body of water, the combined information obtained from the UWSN can provide environmental sampling in not only the spatial dimension but the time dimension as well. Second, the sensors in the UWSN float can allow for what is referred to as adaptive sampling, where changes are made in real-time by the sensors, like moving vertically in the water column. Also, the floating sensors can increase reusability by allowing easy recovery. Finally, [1] provides a list of constraints caused by the underwater environment that UWSNs, facilitated by acoustic communications, is able to solve. First, the underwater environment is not suitable for humans, which gives rise for unmanned craft and sensors. Closely related to this constraint is the need for tetherless networking, which acoustic communications greatly relieves. Finally, UWSNs, using underwater acoustic communications in a network, allow for large-scale monitoring of the underwater environment.

The use of underwater acoustic communications is not without its challenges. In [3], the challenges related to high-speed acoustic communications is listed. Acoustic communications experience a limited bandwidth of the channel, multipath due to reflection, and Doppler shifts that are considered fairly large. Shallow water environments exasperate these issues, which are the main area of focus in modern research. However, research in acoustic communications has provided methods of trying to alleviate these issues.

According to [1], there are two fundamental advancements in the state-of-art of acoustic communications. First, the movement from analog to digital communication schemes has greatly increased performance. Research has been done in both noncoherent modulation, including frequency shift keying (FSK) and coherent modulation, such as and
quadrature amplitude modulation (QAM). Coherent modulation has increased the performance of acoustic communications greatly. Second, the introduction of orthogonal frequency-division multiplexing (OFDM), which is a multicarrier modulation scheme, has been of great importance to the increased performance of acoustic communications.

In [3] and [4], the state-of-art of acoustic communications is provided. First, various channel equalization methods were investigated in the literature. Channel equalization includes the use of adaptive decision feedback equalizers (DFEs) to mitigate the issues in the underwater acoustic channel. One notable example is the use of DFEs with phase locked loops (PLLs) to alleviate the effects of intersymbol interference (ISI) prevalent in underwater acoustic environments [4]. Another advancements are phase conjugation, which uses time reversal mirrors (TRMs) [4]. TRMs allow for a reduction in the delay spread caused by multipath and is explained as improving the performance of communication with combined with DFEs for channel equalization. Third, significant improvement has been made in channel modeling, particularly in shallow water environments. In line with [1], the next area of current research put forth by [3] is the development of multi-carrier modulation (MCM), such as OFDM, which performs equalization in the frequency domain. According to [4], MCM divides the bandwidth into separate subchannels. Other methods of MCM, such as multiple FSK, have shown promise in the improvement of acoustic communications. Finally, spatial modulation, such as multi-input multiple-output schemes, have been explored to increase the bandwidth efficiency of the acoustic channel. Due to the importance of underwater acoustic communications, much more advancements in the state-of-art of acoustic communications is expected in the future.
Section 1.2: Contributions of this Thesis

The efforts covered in this thesis are the first steps in the creation of a mobile platform for acoustic communications and networking using the Iver3 Autonomous Underwater Vehicle (AUV), which is a “low-cost … single-man portable” AUV developed by L3Harris Oceanserver [5]. The implementation involves the integration of the two computers onboard the AUV with a Subnero acoustic modem. The main audience of this thesis are those in the field of underwater acoustic communications and even underwater robotics in general. For the underwater acoustic communications community, the integration of the acoustic modem with the AUV provides a way of scheduling acoustic transmissions based on certain vehicle parameters, such as location and depth. Previously, only time was considered in scheduling, but the integration now opens the door for autonomous control of acoustic transmissions to decrease the bit error rate. On the other hand, underwater robotics researchers and users now have the ability to communicate with and control the Iver3 AUV while it is deployed underwater using MOOS-IvP applications. The work details a method to acoustically control the movement of the vehicle, such as commanding the vehicle to return to a specific location.

There are mainly three contributions that the research provides. First, MOOS-IvP with iOceanServerComms was installed on the backseat computer of the AUV. These software packages extend the possibilities of future research involving the Iver3 vehicle for being a mobile acoustic communications platform. Second, the Subnero modem was integrated with the MOOS-IvP autonomy software onboard the backseat computer of the AUV. Based on that integration, a MOOS application was created that sends location information to the modem in the form of GPS coordinates and depth in meters. Using the location information, the modem can make an informed decision about when to transmit acoustic transmissions. Also, because the
acoustic transceiver must be underwater during use, the depth information protects the acoustic transceiver from damage. Finally, the Subnero modem provides a route to communicate between a computer on the shore and the AUV. Several MOOS applications were created that use the modem to listen for a datagram message, which represents commands and inquires for the AUV. One such pair of commands is to either abort the current mission running on the AUV and return to a specific location, or to change the current mission to another preplanned mission. If the modem receives these messages, the modem will forward the command using the application programming interface (API) to the backseat computer running a MOOS-IvP mission, which will send the AUV to a designated location. Another MOOS application was created that would remotely change the signal power level of the Subnero modem based on received datagram messages. Two other applications were created to handle inquires for information. The first MOOS application responds with the depth value of the AUV. The other application uses multi-hop communication to provide the location information for each node in an underwater acoustic network. Specifically, a node receives a collated list of the location from the previous node, appends its location information to the list, and then forwards the information to the next node in the network list. With these research contributions, the first steps toward a mobile platform for acoustic communications are completed.

Section 1.3: Literature Review

The study of underwater acoustic communications and networking is a fast, active, and growing field of research across the globe. In [6] when describing the development of basic Underwater Acoustic Networks (UANs), UANs are defined as being “formed by establishing two-way acoustic links between autonomous underwater vehicles and sensors.” UANs are necessary for multiple different applications. In [2], multiple different applications are listed
including communication networks of submarines and AUVs, surveillance, environmental monitoring, mine detection, etc. When AUVs, sensors, and a surface station are connected in a network, the resulting “configuration creates an interactive environment where scientists can extract real-time data from multiple distinct underwater instruments” [6]. In [7], one such network configuration was created as a testbed for researchers to use in studying UANs.

However, UANs are not without their challenges. Some of these challenges relate to the inherent differences between acoustic communications and the more well-established field of radio communication. First, according to [2], the bandwidth in a UAN is more limited than in traditional radio communication because the “attenuation of acoustic signals increases with frequency and range,” where the typical system, operating over tens of meters, had a bandwidth of 100 kHz while that same system has an available bandwidth of 10 kHz when operating over several kilometers. According to Second, the propagation delay, due to the speed of sound, is longer than the radio communication operating at the speed of light. Third, the impulse response not only varies spatially in a UAN, but it also varies as a function of time as well. Finally, the bit error rate (BER) of a UAN is much higher than in radio communication links. Other issues in the research of UANs involve the actual design of the acoustic network itself, such as the physical layer design, the medium access control (MAC) layer design, and the network layer design. For network layer design, multi-hop delivery is better suited in an UAN than normal single-hop delivery due to the energy efficiency of multi-hop delivery. [2] When performing multi-hop delivery, the nodes in the UAN “can either have pre-established routes … or they can do this in an ad-hoc manner” [8]. The two methods, called virtual circuit routing and packet-switch routing, respectively, are the common form of routing methods implored in UANs, but virtual circuit routing is considered the better option [2].
To deal with the issues related to UANs, multiple possible solutions are being researched. In [2], the issues related to network design are proposed to be solved by a cross-layer approach. Also, self-adaptability of the network topology and protocol are proposed to combat the sudden changes in the underwater environment. In [8], there is agreement on the self-adaptability of the network, and includes error correction, usually through redundancy of the transmitted message, as an important solution. Also, in [8], a solution to the issues related to routing of data is “integrated underwater acoustic communication and localization.” When describing the Seaweb system, [9] goes further by saying that “GPS … when combined with the onboard acoustic modem, make these AUVs effective mobile gateway nodes without the attendant vulnerability of moored gateways.” Other research in UANs includes [10], where the usage of security and cryptography, as specifically related to UANs, was discussed, and in [11], where a new MAC protocol, called MACA-MCP, was created that increased the efficiency and data rate of underwater acoustic communications. When exploring the use of Underwater Ad-hoc Networks (UANETs) and Underwater Sensor Networks (UWSNs) and the challenges associated with their development, [12] concludes that “UANET and UWSN are inter-disciplinary challenges requiring integration of acoustic communications, signal processing, and mobile network design.” The mobile network design involves the usage of AUVs, which are a common aspect of UANs. Accordingly, the development of underwater acoustic communications systems for AUVs is a necessary step towards the completion of an UAN.

The advancement of underwater communication systems for AUVs is a necessary step in the development of UANs. According to [13], the advancements made in the research of AUV communication include radio communications, acoustic communications, and optical communications. However, acoustic communications is the “most commonly used underwater
communication method employed nowadays because of the lower attenuation in seawater” [13].

One research topic that has come about in recent years is the cooperation of multiple AUVs in the field, which involves real-time acoustic communications between the AUVs in the network to properly perform the cooperation. The cooperation of these AUVs is a complex task due to the vehicles operating in a “low bandwidth acoustic channel” [14]. In [14], the implementation of current cooperation schemes is described as typically using single-hop delivery. An application of the communication system of an AUV is the acoustic control of the vehicle, which is a well-researched topic in the development of AUVs.

The development of control mechanisms using acoustic communications is a field that encompasses both simple and complex systems. An example of the simpler system is in [15] where a 6 Degree-of-freedom (DOF) underwater robot was created that received acoustic transmissions in the form of pulses. Each pulse, operating at different frequencies, represented a command for moving the vehicle. A bandpass filter was used to separate the signals into different frequency bands to decode the desired command message. Another example of the simpler systems is in [16], which created a robotic fish that would give the user better ability to research marine life without alarm. The robotic fish contained an acoustic modem and a transducer and transmitted Modified Binary Frequency Shift Keying (MBFSK) messages. The MBFSK messages were 16-bit words that encoded the thrust, yaw, and video recordings. A detection algorithm was used on a microcomputer to decode the MBFSK messages.

In commercial modems, the communication systems for controlling the vehicle become more complex. In [17], a communication protocol called the Compact Control Language (CCL) is described. The CCL language, created by the Woods Hole Oceanographic Institute (WHOI), is used in multiple WHOI systems including the WHOI Micro-Modem and the REMUS control
system, and is designed to allow communication between multiple AUVs either through a central node or directly with each other. To implement the CCL language, WHOI researchers used 32-byte packets containing a 1-byte indicator and 31 bytes of message content. An example of a CCL message is the bathymetry message, which consists of 2 bytes for the depth value and 3 bytes each for the latitude and longitude. With these data lengths, the resolution of the depth reading is around 10 cm and the latitude and longitude have resolutions “to several meters.” To control the vehicle, a redirect message can be sent to command the vehicle to create a new survey mission. The redirect message contains the information necessary to create a new lawn-mower pattern mission on the vehicle and execute the mission. Also, the CCL language allows the transmission of data files. In a field test for detecting mine-like objects (MLOs), the CCL language “was used to send compressed snippets of MLO sidescan to a base station in real time.” Other acoustic modems, such as one based on the WHOI Micro-Modem when developing the Bluefin AUV in [18], have used the CCL language to control vehicle movement.

Systems have been created that are similar to the work presented in this thesis, but these systems do not involve the integration of a software defined acoustic modem with the open-source MOOS-IvP autonomy software running on the backseat computer of an Iver3 vehicle. One such system is found in [19], which created a network of AUVs, autonomous surface vehicles (ASVs), buoys, a sensor network, and command and control units (CCU). The CCU were human-operated, and the buoys housed transponders that the AUVs could use for localization. The modem used on the AUVs and ASVs was the Teledyne Benthos ATM-855, which uses a publish/subscribe Application Programming Interface (API) called InterModule Communication API. The control language for the AUVs contained a 1-byte tag that identified the type of message being transmitted. The short and “compact definition has the aim of reducing
transmission time and errors.” The messages available were to enable or abort a preloaded AUV mission, ask for the mission status including position and velocity, and ask for vehicle status such as current battery power.

In [20], the Unified Command and Control (C2) architecture was developed. The Unified C2 used MOOS-IvP on a backseat computer to control the vehicle and used the Dynamic CCL (DCCL) language to communicate with the AUV. The DCCL messages are easily reconfigurable due configuration being performed in XML. The acoustic communications are handled by a MOOS application called $pAcommsHandler$. The $pAcommsHandler$ application encodes and decodes the messages, queues the messages based on a priority criterion, and interacts with the firmware of the modem. A command processing application, called $iCommander$, uses an Ncurses graphical user interface directly from the terminal window. There are about a dozen different DCCL messages, including messages for deploying missions and checking vehicle status.

**Section 1.4: Introduction to Main Efforts**

Underwater acoustic communication networks utilize various platforms both mobile and stationary [10]. One of the more common mobile platforms for acoustic communications is the autonomous underwater vehicle (AUV), which can be installed with an acoustic modem as a payload. In this paper, an Iver3 vehicle, which is an AUV developed by L3Harris, was chosen to become a mobile platform for acoustic communications. The development of the mobile platform was completed in three stages. First, autonomy software was installed to provide a more robust autonomy for the customized needs of users. Second, an acoustic modem was installed and integrated into the AUV system to develop the acoustic communications capabilities of the mobile platform. Finally, the Subnero was further integrated with the autonomy software to
allow for remote receiving and receiving of acoustic messages, which allows for simple AUV acoustic networks to be created.

The Iver3 AUV has two computers onboard the AUV that control the vehicle. The main computer, also known as the front-seat computer, uses the proprietary Underwater Vehicle Console (UVC) to interface with the sensors on the AUV and control the movement of the AUV through the water. The main computer cannot be changed by the user without voiding the warranty of the vehicle. Instead, the secondary computer, called the backseat computer, can be changed by the user to run any software, such as autonomy software. The backseat computer of the Iver3 vehicle used in the current research was installed with the MOOS-IvP autonomy software. As an interface between the MOOS-IvP software and the UVC on the main computer, the MOOS application *iOceanServerComms* was installed that uses the serial port connections between main and backseat computers to send and receive information. *iOceanServerComms* allows the MOOS-IvP software to send commands, including commands to move the vehicle, to the main computer for execution. Once the MOOS applications were installed and configured to the Iver3 vehicle, a field experiment was performed that showcased the autonomy of the vehicle. After completing the integration of main and backseat computers, an acoustic modem can be installed to form the acoustic communications system of the mobile platform.

To develop the mobile acoustic communications capability, the Iver3 vehicle was installed with a Subnero acoustic modem. The Subnero device is a software defined acoustic modem, which means that the Subnero modem is a “reprogrammable modem where the whole transmitter and receiver-side processing chains are programmed in software” [21]. Reprogrammable modems allow the user to change various parameters on the modem by simply giving commands to the modem using some interface [21]. The Subnero modem uses the Unetstack API as an
interface to configure the various device parameters. One such set of parameters was location of the modem, referred to as the node location. Because the acoustic transceiver on the AUV must be underwater before transmitting, the modem knowing the node location is imperative. To update the node location, a MOOS application called $iModemComms$ was created that used the Unetstack API to change the node location parameters from the backseat computer. Once the MOOS application was completed, the $iModemComms$ application was used in a major field experiment that used the Iver3 AUV as mobile platform for underwater acoustic research. The setup and results of the experiment show that the $iModemComms$ application was functioning successfully. Once the AUV was shown to be functional as a mobile platform for acoustic communications, software could be developed that allowed for acoustic networking capability as well.

Acoustic networking requires the uses of acoustic modems and transceivers to send and receive messages. Because of its software-defined nature, the modem allows for remote controlling of the transmitting and receiving of datagram messages, which are a sequence of bytes. A method exists that can be used to command the modem to transmit a given datagram message from a C program running on a computer connected to the modem by an Ethernet cable. Another method allows a C program to access a datagram message received by the modem. The two methods can then be used to form many different functionalities. First, the remote receiving method was integrated into the $iModemAbort$ MOOS application that, depending on the received datagram message, would abort an AUV mission running on either the main or backseat computer. Therefore, the user at the shore can command, using the remote transmitting method, the AUV to abort the current mission and return to a specified location for retrieval. The application was then used as the basis for another application, called $iModemDeploy$, that allows
the user to command the AUV to quickly change between missions given input transmission from the shore. Second, another MOOS application, called \textit{iModemPower}, was created that used the Unetstack API to change the signal power level parameter of the modem based upon remotely received datagram message. Thus, the user at the shore can adjust the signal power level by remotely transmitting the requested value. Third, a MOOS application, called \textit{iModemInfo}, was created that allowed the AUV to respond with the value of a requested parameter. Finally, the \textit{iModemInfo} application was extended to allow information to be forwarded through a multiple AUV acoustic network. The new application was called \textit{iModemForward}, which forwarded the current location of a node along the network until received back at the shore. Each AUV would append its own node location to the transmission so that the user at the shore would have a list of the current location of each node in the network.

Experiments were performed that proved the functionality of each new MOOS application. The applications were then used in a field experiment involving two AUVs, which showed the efficacy of the MOOS applications in creating a network of multiple acoustically-linked AUVs.

\textbf{Section 1.5: Organization}

The work described to be described in this thesis has been organized into three separate chapters. In Chapter 2, the integration of the frontseat and backseat computers of the Iver3 AUV is described. The description of this integration includes an introduction into MOOS-IvP and the \textit{iOceanServerComms} application. A description of the issues solved in integrating the two computers, the setup of an experiment of test the integration, and the results of the experiment are also described. In Chapter 3, a description of the integration of the Subnero acoustic modem as a payload of the Iver3 vehicle is given. The chapter details the creation of the \textit{iModemComms} application and the deployment of the application in two different experiments. The results from
these experiments are provided. Chapter 4 is separated into eight sections. Section 4.1 provides a
brief introduction to the chapter. Section 4.2 describes the method of using the Unetstack API to
remotely transmit datagram messages from the Subnero acoustic modem, while Section 4.3
describes the method for remotely receiving the acoustic datagram messages using the Unetstack
API. In Section 4.4, a description of the \textit{iModemAbort} application, and its associated shoreside
transmission program, is given. An experiment is detailed that demonstrated the efficacy of the
program in the field. Section 4.5 gives a detailed description of the \textit{iModemPower} application
and the experimental results for verification. Section 4.6 details the \textit{iModemDeploy} application
and the experimental efforts in testing the application in the field. In Section 4.7, the \textit{iModemInfo}
application is described and verified with experimental results. Finally, Section 4.8 details the
\textit{iModemForward} application and the experimental results obtained in a field experiment. Chapter
5 forms the conclusion of this work.

For the appendices, Appendix A contains the programs and other files used in the
integration of the frontseat and backseat computers described in Chapter 2. Appendix B holds the
programs and other files related to the work described in Chapter 3 for integrating the Subnero
modem into the Iver3 vehicle system using iModemComms. In Appendix C, the programs and
files pertaining to the iModemAbort application are provided. Appendix D contains the files
associated with the iModemPower application, while Appendix E contains the files associated
with the iModemDeploy application. Appendix F includes the programs involved with the
iModemInfo application. Finally, the programs and results pertaining to the iModemForward
application are provided in Appendix G.
CHAPTER 2

INTEGRATION OF THE FRONTSEAT AND BACKSEAT COMPUTERS
OF THE IVER3 AUV

In order to create a lake testbed for underwater acoustic communications, a suitable vehicle must be chosen. The Iver family of vehicles developed by L3Harris is the “first commercially developed family of low-cost Autonomous Underwater Vehicles” [5]. The Iver3 vehicle is further described in [5] as “single man portable” due to its length of 74-85 in., diameter of 5.8 in., and weight of 59-85 lbs. The Iver3 vehicle also advertises an endurance of between 8 to 14 hours. Based on the factors of size, low-cost, and endurance, the Iver3 Autonomous Underwater Vehicle (AUV) was chosen. Two Iver3 vehicles are pictured in the water in Fig. 2.1.

![Fig. 2.1. The Iver3 vehicle.](image)

The Iver3 AUV contains two important sections called the payload section and the main section. The payload section will be discussed in a subsequent chapter. Within the main section
of the AUV are two computers called the frontseat and backseat computers. The frontseat computer contains all the hardware and software necessary to operate the vehicle. The primary user interface of the AUV is Underwater Vehicle Console (UVC). The UVC allows the programmer to either manually control the vehicle or run a mission that is preplanned. However, these functionalities are limited, which raises the need for the backseat computer.

The backseat computer onboard the AUV is given specifically for the user to have complete control. The backseat and frontseat computers are connected via a serial connection and communicate using NMEA sentences. The UVC listens for these NMEA sentences from the backseat computer and promptly executes a response. A program running on the backseat computer can send a request for information, such as GPS or compass data, and the UVC will form a response message with the requested data included and send back to the backseat computer. Furthermore, the backseat can send commands to the UVC that control vehicle behavior. An important example of an NMEA sentence used for controlling the vehicle behavior is the $OMS command. The $OMS command is used to specify commands for the servos, which are the motors and the fins. The sentence structure for the $OMS sentence is shown in Table 2.1, which is based on the table found in the Remote Helm Manual for the Iver3 AUV [22].

<table>
<thead>
<tr>
<th>Format</th>
<th>$OMS, &lt;Heading&gt;, &lt;Depth&gt;, &lt;Max Angle&gt;, &lt;Speed&gt;, &lt;TO&gt;&lt;*cc&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Autopilot set points for servo control</td>
</tr>
<tr>
<td>$OMS</td>
<td>Message type: Servo Control</td>
</tr>
<tr>
<td>&lt;Heading&gt;</td>
<td>Desired heading in degrees relative to true north</td>
</tr>
<tr>
<td>&lt;Depth&gt;</td>
<td>Desired depth in meters of the AUV</td>
</tr>
<tr>
<td>&lt;Max Angle&gt;</td>
<td>Maximum pitch angle of the vehicle in degrees</td>
</tr>
<tr>
<td>&lt;Speed&gt;</td>
<td>Desired speed of AUV in knots</td>
</tr>
<tr>
<td>&lt;TO&gt;</td>
<td>Timeout of the command</td>
</tr>
<tr>
<td>&lt;*cc&gt;</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

Table 2.1. NMEA sentence format for an $OMS sentence.
As can be seen in Table 2.1, the $OMS sentence specifies the desired behavior in the form of heading, depth, and speed. Then, the $OMS sentence is sent to the UVC on the frontseat computer, which will execute the desired command. With the two functionalities of requesting information and sending commands, it is possible for a program to exist on the backseat computer that can run full missions that control the AUV.

One of the most common programs for underwater autonomy is MOOS-IvP. Mission Oriented Operating Suite (MOOS) is a group of C++ applications that work together to perform autonomous underwater robotics. The group of applications that is running during a mission is called the MOOS Community. In order to specify which MOOS applications are desired in the MOOS Community, the MOOS file (.moos) initializes the MOOS application based on certain parameters. The MOOS Community follows a publisher-subscriber architecture, which means that each MOOS application subscribes and publishes to a central database as opposed to direct communication between MOOS applications [23]. The central database of the MOOS community is called the MOOSDB. A diagram representing the architecture is shown in Fig. 2.2.

Fig. 2.2. Architecture of the MOOS community.
As can be seen in Fig. 2.2, one such application of the MOOS-IvP is the IvP-Helm. IvP-Helm, where IvP is an acronym for interval programming, is the application that determines autonomous movement of the vehicle [23]. All behaviors for the vehicle are specified in a behavior file (.bhv), which the IvP-Helm then mathematically optimizes into a specific mission that the vehicle can then carry out. The MOOS-IvP software was installed on the backseat computer to form the main autonomy software for the Iver3 vehicle. Next, a MOOS application was needed that would provide a way for the MOOS-IvP to take control of the vehicle.

The MOOS application that is used for sending and receiving between the frontseat and backseat computers is called iOceanServerComms. This module takes the commands from the IvP-Helm and converts them into the necessary NMEA sentences. Then, the NMEA sentences are sent to the frontseat computer for execution using the serial connection between them, as shown in Fig. 2.3.

Once iOceanServerComms was installed onto the backseat computer, there were some issues noticed when trying to run the missions. Because iOceanServerComms was developed for use on the older model Iver2 AUV, some changes needed to be made when initializing the MOOS file. First, the sample mission, which is the alpha mission given with iOceanServerComms, used /dev/ttyS12 as the serial port for communicating with the frontseat computer. According to the [24], which is the guide designed for the Iver2 AUV, if the AUV
uses two Intel Atom Motherboards, the two motherboards use COM13 as the serial port connection, which corresponds to `/dev/ttyS12` on a computer with a Linux OS. However, the serial port for the Iver3 AUV was reverted to using COM1, which corresponds to `/dev/ttyS0`. Therefore, the serial port needs to be changed to `/dev/ttyS0` in the `iOceanServerComms` configuration block of the MOOS file. Second, the Doppler Velocity Logger (DVL) was changed between the Iver2 and Iver3 models. Thus, the NMEA sentence containing DVL information is different between models. When the response from the main computer for DVL information was received by the backseat computer, the parsing function would not perform properly and crash the mission. Because the DVL information is not mission critical information, instead of rewriting the parsing function for the DVL information, the DVL Boolean parameter is set to false in the configuration file. The code for initializing the iOceanServerComms application, which is called `last.moos`, is found in Appendix A. Once the MOOS-IvP and `iOceanServerComms` software packages were installed and any issues with the code were resolved, a simple test of the software on the Iver3 was performed.

In order to test the backseat and frontseat computer integration, two missions needed to be created. First, a simple two-point mission needs to be created on the frontseat computer. The frontseat mission needs to be running for the backseat computer to take control of the vehicle. Second, a backseat mission needs to be created that is different enough to tell the difference between the two missions. Therefore, the two missions were chosen to be nearly perpendicular to each other. The test was performed at a small pond called Lake Palmer situated on the campus of the University of Alabama. The pond is roughly rectangular in shape with a length measured to be 150 meters and a width of 50 meters. Due to the proximity of Lake Palmer to the research laboratories, the lake provides an excellent place to perform small missions and experiments of
the AUV. On the other hand, the shallow depth of the lake, which is around 3 meters, does not allow the vehicle to dive, which greatly restricts the design of missions. Therefore, all missions run at Lake Palmer must run at the surface. Fig. 2.4 shows the frontseat mission that was designed for the experiment at Lake Palmer.

The mission starts at the dock, proceeds to the center of the lake, and then will return to the dock for retrieval. Because the program for creating the frontseat missions requires a third point for returning to the dock, the two-point mission becomes a three-point mission with the third point being slightly offset from the starting point. The backseat mission, shown in Fig. 2.5 starts a specific GPS location which is defined in the MOOS file. The behavior file, found in Appendix A called current.bhv, specifies that the AUV will travel due east for 25 meters and then return to the starting position, which can be seen in Fig. 2.5.
After the backseat mission captures both points, the backseat mission will terminate and the frontseat mission will restart and complete the mission. During the field experiment, a total of six different runs of the missions were performed. During the third mission, which will be referred to as Mission 3, the frontseat mission ran unimpeded. In the two subsequent missions, the backseat mission was enabled. A comparison of two missions, Mission 3 and Mission 4, is shown in Fig. 2.6.

Fig. 2.5. Path of the backseat mission.

Fig. 2.6. Comparison of the AUV tracks in missions 3 and 4.
In Fig. 2.6, the blue track represents the GPS coordinate data saved in a log file by the UVC during Mission 3, where the frontseat mission was run unimpeded. On the other hand, the red track represents the GPS coordinate data saved during Mission 4, where the backseat mission took control of the vehicle. Though there is a slight change in the vehicle motion between the two missions, there needed to be a mission that showed a more drastic change, which would provide more solid verification of the backseat control. Accordingly, the backseat mission was changed. Keeping the same behavior as described previously for the backseat mission, the repeat parameter was given the value of 5, which informs the IvP-Helm to repeat the behavior five times. After the behavior is repeated the fifth time, the backseat mission will terminate and return control to the frontseat computer, which will resume its original mission. The repeated behavior was tested during Mission 6. In Fig. 2.7, the results of the test are shown as a comparison of the 3-dimensional tracks of the movement of the AUV based on GPS log data on board the AUV.

Fig. 2.7. Comparison of the AUV tracks in missions 3 and 6.

In the figure, two separate tracks are being compared. The track in blue represents the GPS latitude and longitude values throughout Mission 3, where the frontseat mission ran unimpeded by the backseat. The track in red shows the backseat mission taking control of the
movement of the vehicle, performing the backseat behavior, and repeating the movement. The aberrant behavior causing the AUV to collide with the dock was due to a piece of debris getting caught in the motor and fins. The collision forced the experiment to be ended prematurely before the fifth repetition of the backseat behavior. An annotated version of the GPS track obtained during Mission 6 is pictured in Fig. 2.8. The figure captures GPS location of the vehicle as well as the water depth of Lake Palmer at the current location, which is noted by the color of the track. The water depth of the track shows that the depth of Lake Palmer is never much greater than three feet, which makes Lake Palmer not an ideal spot for testing the submersion of the vehicle. Also, the approximate location of the backseat mission waypoints, both the starting point and the 25-meter east objective point, are labeled in the figure.

![Annotated GPS track of Lake Palmer mission 6.](image)

Regardless of the aberrant behavior, the results of the experiment show that the backseat computer did control the movement of the vehicle, meaning the integration of the backseat and frontseat computers for autonomous control of the Iver3 AUV was successful.
CHAPTER 3
INTEGRATION OF THE IVER3 AUV WITH AN ACOUSTIC MODEM

As stated in the previous chapter, the Iver3 AUV contains two sections: the main section and the payload section. The payload section is an empty cavity in the front portion of the AUV where the user can install various sensors and other equipment. For the purposes of acoustic communications, a WNC-M25MSE3 embedded acoustic modem by Subnero was installed in the payload section. Various ports are found that allow the installation of an acoustic transducer on the hull of the AUV. Specifically, the AUV was configured with the transducer on the topside of the AUV, as can be seen in Fig. 3.1.

![Acoustic transmitter installed on topside of AUV.](image)

Acoustic transducers, due to its ability to produce heat during transmission, must be underwater during transmission. When running a mission that involves an extended series of
acoustic transmissions simultaneous with the movement of AUV, there is the possibility that the
vehicle may surface, and the acoustic transmission continues. Therefore, it is important to create
a safety feature that informs the modem of its depth under the water before deciding to transmit.

To create the safety feature, a MOOS application was created that would form an interface
between the MOOS community running on the backseat computer and the Subnero acoustic
modem. The MOOSDB contains several variables of interest, including the GPS coordinates of
the vehicle and the depth of the vehicle under the water. The MOOSDB obtains this information
using the $OceanServerComs application, which interfaces with the frontseat computer. The
$OceanServerComs application sends an $OSD sentence to request that data be sent to the
backseat, which includes values from both the compass and GPS sensors. The data is then
received by the backseat computer, parsed using the functions found in the parse.cpp file of
$OceanServerComs, and then placed into the MOOSDB. The three MOOSDB variables of
interest are the NAV_X, NAV_Y, and NAV_DEPTH values, which represent the latitude,
longitude, and depth of the vehicle, respectively. Next, the three location parameters need to be
shared with the Subnero modem.

A MOOS application called iModemComms was created with the intention of interfacing
the Subnero modem with the backseat computer to inform the modem of the location of the
vehicle. The iModemComms application was generated using the MOOS application generation
script called GenMOOSApp_AppCasting.sh. When running the generator script, the user must
choose which type of application to create based on the naming convention used by MOOS.
Since the desired application is an interface between the modem and the MOOSDB, the name
convention requires a prefix of “i” [25]. Hence, the name of the application is iModemComms.
After creating the application, a directory containing all the necessary C++ header and code files
is generated. The user only needs to change the `ModemComms::Iterate()` function in the `ModemComms.cpp` file to obtain the desired functionality of the program. A basic pseudocode of the necessary changes was found in [26]. The completed version of `ModemComms.cpp` can be found in Appendix B. To communicate with the modem using `iModemComms`, there needs to be a user interface between the modem and the backseat computer.

The Subnero acoustic modem operates using the Unetstack architecture, which is an *fjage* agent-based architecture rather than a traditional layer-based network architecture. In an agent-based architecture, there is no enforced hierarchy. Rather, each agent application interacts with any other agent to fulfill its own purposes. The typical interaction between two agents is called a message. Messages are typically either requests or responses. Each agent provides services, which are functionalities that define how the agent responds to messages. For instance, this work relies heavily upon the datagram service. The `DatagramReq` message is sent to an agent to ask the agent to send some form of data. If the agent also provides a physical service, then the agent can use the `DatagramReq` message to control the modem driver to transmit the desired datagram sequence acoustically. [27] According to [28], Unetstack “provides a Gateway interface to interact with *fjage* agents”. The Gateway acts as an application programming interface (API) in that it provides all the necessary functions for establishing a TCP/IP internet connection between an external program and an agent running on the modem. The API functions, which we call the Unetstack API, are open source and can be downloaded from a GitHub repository hosted by the Subnero company [29]. The Unetstack API contains multiple versions of the API each in a different language. Because MOOS applications are written in C++, the C version of the API was chosen. After downloading the Unetstack API functions from [29] onto the backseat computer, the user must build the API using the make command. Once the Unetstack API is
built, two library files will be created, called *libfjage.a* and *libunet.a*, which can be linked to when building the MOOS application.

To link to library files, two groups of code must be added to the MOOS application. First, the header files, *fjage.h* and *unet.h*, must be added at the start of the *ModemComms.cpp* file. Because the header files are written in C while the application is written in C++, the header declarations must be wrapped by an extern command, as can be seen in the *ModemComms.cpp* file. Second, MOOS applications are compiled using Cmake, so the *CMakeLists.txt* file, which contains all the necessary commands for building the MOOS application, must be changed. The completed text file can be found in Appendix B. To link to the two library files discussed previously, the text file must contain calls to the *find_library()* function, which finds the library files corresponding to the header files declared in the main C++ program, for both header files. Next, the *target_link_libraries()* function must be called to link the library files. Finally, the *include_directories()* function is called to include the directory of the library and header files. Once these changes have been made, the MOOS application is ready to use the Unetstack API.

The Unetstack helpfully provides the necessary variables for storing the location of the AUV. Using the NODE_INFO service of the NodeInfo agent of the Unetstack, each modem is treated as a node in an underwater acoustic network with its own address, location, etc. The node location is treated as an array of three floating point numbers representing the location of the modem in a three dimensional coordinate system, which is stored in a variable called *node.location* in the NODE_INFO service of the NodeInfo agent [25] [30]. To update the node location information, the user must first open a socket connection over Ethernet with the modem from the *ModemComms.cpp* file using the *unetsocket_open()* function. Next, the user must specify the agent providing the service. Originally, the Shell agent, which is found in the
The `org.arl.fjage.shell` package in Unetstack was used. The Shell agent, as part of its package, has a Groovy scripting engine, which allows the running of Groovy scripts. The Shell agent can be interfaced with either using the web interface or the API. However, when updating using the API, the modem would refuse to run any user input on its web interface, which prevented the user from running a transmission script. The refusal is due to the constant updating of the `node.location` from the API. A representation of the interactions between agents is described in Fig. 3.2.

Fig. 3.2. Interaction of Unetstack agents with ignored script request.

As can be seen in Fig. 3.2, the Shell agent is being inundated with requests from the Unetstack API to change the `node.location`, while intermittent requests for other user input from the web interface are being ignored. Therefore, a new agent was created called `shellForAUV`. To create the `shellForAUV` agent, a pair of lines were added to the `startup.groovy` file that created the `shellForAUV` as a new Shell agent. The new `startup.groovy` file, which was saved to the `/scripts` directory of the modem, can be found in Appendix B. Once the Shell agent was created, the node location could be updated simultaneously with the running of a transmission script. The `shellForAUV` operates as Shell agent similar to the Shell agent of the Unetstack, except without
the web interface. Therefore, the original Shell agent is left free to handle the user input from the web interface, while the new `shellForAUV` agent can handle the requests from the API. A representation of the interaction of the new agents is pictured in Fig. 3.3.

![Diagram of agent interactions with new shell agent.](image)

The `iModemComms` application takes the values of NAV_X, NAV_Y, and NAV DEPTH from the MOOSDB and creates a message that will be sent over the socket connection. The message was created in three sections. First, the message is specified to be a request to execute a command on the Shell. Second, the recipient must be set, which is the `shellForAUV` agent. Third, the command to execute, which is the `node.location` update command, is added to the message. Using the `fjage_request()` function, the command can then be sent to the modem and executed on the modem. The modem is now receiving up to date GPS and depth information from the backseat computer. The AUV location can now be integrated into a script for running an acoustic transmission from the modem.

To run a set of acoustic transmissions, a Groovy script is written and saved in the `/scripts` directory on the Subnero modem. Two examples of Groovy transmission scripts can be found in Appendix B. First, a log file for the transmissions must be created. The log file contains a record
of the signal being transmitted, the date and time of the transmission, and the location of the AUV at the time of transmission. To use the modem for transmissions, the program must subscribe to the physical layer and the power level of the physical layer must be set to a desired value, such as -10 dB. Next, the minimum depth of the required for transmission must be set. Ideally, the minimum depth would be roughly 3 meters below the surface for navigation safety. Namely, we want the AUV to be safe from collision from a passing boat and not submerged deep enough to strike the bottom of the lake. However, an issue was noticed when testing iModemComms that showed that the NAV_DEPTH variable never went below 0.2 meters. Due to a major experiment, the use of the safety features implemented by iModemComms was required before the issue was resolved. The issue was resolved after the major experiment, and the resolution will be described later in this chapter. Therefore, the minimum depth was decided to be 0.1 meters. Next, the acoustic signal is saved in the form of a .txt file, which is loaded into the program for transmission using the load() function. The acoustic signals were designed by researchers from multiple different universities and were stipulated have a total runtime of two minutes. The two-minute runtime was chosen to minimize the amount of storage space used on the modem. Usually, the loading of the signal takes a noticeable amount of time due to the size of the .txt file, so a four-minute delay is introduced to provide a set starting point for the transmissions. The four-minute delay was also chosen to provide a reasonable amount of time for preparing the vehicle for launch and allowing the vehicle to travel to a specific depth before beginning the transmissions. Of course, the four-minute delay will push the series of transmissions back, which may cause some missed transmissions at the end of the mission when the vehicle is surfacing. However, the losses caused by this would be at most two missed transmissions, because only two two-minute transmissions will fit in a four-minute window. The
minimal number of lost transmissions were determined to be an acceptable tradeoff for the time necessary for loading the waveforms, preparing the vehicle, and submerging the vehicle. If the node.location parameter is above the minimum depth threshold, the transmission script will use the pbtx function to transmit the signal waveform in the passband. A block diagram describing the interaction of all of the components in the system can be seen in Fig. 3.4.

![Diagram of Ecomapper-Subnero system integration.](image)

The iModemComms application was first deployed in a major experiment at Lake Tuscaloosa in Tuscaloosa County, Alabama during the summer of 2021. The experiment involved a total of four different missions. Fig. 3.5 shows the AUV track of the third mission performed during the experiment, with the vehicle depth included, as well as each location of an acoustic transmission. During the experiment, a boat was used as the starting position of the AUV, which is where the AUV would be launched. From the boat, a hydrophone, or underwater microphone, array was suspended on a rope in the water. The mission track, referred to as a lawnmower mission, was chosen to provide a variety of different distances and directions based on the acoustic researchers needs. The data label for each transmission corresponds to the index of the transmission.
Fig. 3.5. Recorded vehicle depth and transmission location with depth issue.

The transmission script for this mission, called *s2_adaptive_script_long.groovy*, is found in Appendix B. As can be seen in the transmission script, the total amount of transmissions is 120. However, only 116 transmissions occurred due to the depth value going above the minimum depth value in the script of 0.1 meters. For the most part, each transmission is below the 3-meter depth requirement. However, one will notice that at points 114 to 116 the transmission occurs above the depth requirement. The issue is caused by the depth of the vehicle, according to the NAV_DEPTH variable in the MOOSDB, never went below 0.2 meters, which does not equate to the values obtained from the frontseat. The log of the mission, called *s2-log.txt*, is found in Appendix B to corroborate the findings. The issue was found to be in the parsing functions for setting the NAV_DEPTH value. To obtain the depth, *iOceanServerComms* uses the value in the <DEPTH> field of the $C sentence that the backseat computer receives. The $C sentence structure is pictured in Table 3.1 [22].
The <DEPTH> field contains the data from the compass depth sensor in seawater. However, the AUV in use often operates in freshwater exclusively. If the AUV is operating in freshwater and the <DEPTH> field is used without any data correction applied, the NAV_DEPTH variable in the MOOSDB will not actually represent the true depth in recognizable units. To change this, a new set of commands was added to the parsing function for parsing the $OSI sentence. The $OSI sentence contains the <COR DFS> field, which is the corrected depth from surface of the vehicle in feet [22]. The value obtained in the <COR DFS> field was later validated, which will be described later. The corrected parse.cpp function is shown in Appendix B. The important change was to use the MOOSChomp() function, which is used to isolate parts of NMEA sentences to convert them into integer variable form, on the <COR DFS> field in the $OSI sentence. After converting the value into meters, the true depth of the vehicle can be saved as TRUE_DEPTH in the MOOSDB. The TRUE_DEPTH value is the new value that is sent to the Subnero modem as part of the location. In the future, it may be prudent to set the original NAV_DEPTH value in the MOOSDB to be equal to the TRUE_DEPTH. Currently, the missions on the backseat computer do not require submersion, so the discrepancy does not cause an issue at the moment. However, any experiment involving the backseat mission to submerge will require further redesign of the parsing functions of iOceanServerComms to provide adequate behavior.
After the experiment, the issue of incorrect depth readings was discovered and addressed. To test the corrected `iModemComms`, another experiment was carried out at Lake Tuscaloosa. The fourth mission from the previous experiment was redone with some minor changes. First, the number of acoustic transmissions was halved from 72 transmissions to 36 transmissions to shorten the length of the experiment. Accordingly, the total mission distance was shortened. Because the boat used in the previous experiment was unavailable for the retest, the mission was designed to start and end at the dock. Therefore, the lawnmower mission was eschewed for a shorter loop mission. The results of the experiment are shown in Fig. 3.6, which shows the AUV track as well as the location of each transmission.

![Fig. 3.6. Recorded vehicle depth and transmission location after correction.](image)

The Groovy transmission script, called `s3_adaptive_script_short.groovy`, can be found in Appendix B. The transmission script was designed to transmit the signal 36 times. However, the signal was only transmitted a total of 30 times, as can be seen in Fig. 3.4. For the last six transmissions, the AUV rose above the 4-meter minimum depth threshold, so the transmissions were not conducted. The transmission log, created by the Groovy transmission script running on the Subnero modem, shows that a total of 36 loops did occur by writing a log entry for each
desired transmission. However, the final six entries were left blank by the Groovy transmission script, which means that the Groovy script decided to not transmit based on the depth value being above the minimum depth threshold. The transmission log, which is called s3-log.txt, can be found in Appendix B, and the log corroborates that each transmission occurred below a depth of 4 meters. Also, the depth values in the log file corroborate the depth values pictured in the color of the track in Fig. 3.4, which were obtained from the frontseat navigation logs, so the value in the <COR DFS> field is validated. Based on the results of the redone experiment, the iModemComms application was proven to accurately relay the location and depth information to the modem for informed scheduling of acoustic transmissions.
CHAPTER 4
ACOUSTIC CONTROL OF THE IVER3 AUV

Section 4.1: Introduction

Another requirement for creating a lake testbed for acoustic communications is the creation of a way to communicate with and control the AUV while underwater. Typically, the user of an Iver3 AUV connects to the AUV using a Wi-Fi connection. If the desired mission requires the AUV to submerge, as is often the case in acoustic communications situations, the ability of the user to connect via Wi-Fi is terminated. Therefore, the only valid method of long-distance communication is acoustics. As discussed previously, the AUV contains a Subnero acoustic modem, which transforms the acoustic waveforms into computer useable information. In order to form a link between the user computer above the water and the modem onboard the AUV, another Subnero modem is connected to the user computer via an Ethernet cable. The Subnero modem on the shore is referred to as the shoreside modem. The graphic of the experimental setup is pictured in Fig. 4.1.

Fig. 4.1. Experimental setup for testing acoustic control.

Using the experimental setup, important tasks can be performed such as controlling the vehicle
to abort the current mission, controlling the modem to change the signal power level, and controlling the vehicle to change between separate mission paths. The basic building blocks of these methods are a program that transmits the command message and a MOOS application that receives and acts on the received command message.

**Section 4.2: Using the Unetstack API to Remotely Transmit Datagram Messages**

A Program must be created that runs on the user computer, which is connected to the shoreside modem. The majority of such transmission procedures follow the same structure based on the `txdata-reliable.c` sample code in [29]. The `return.c` program is an example of a transmission program found in Appendix C. The program uses command line arguments to specify the IP address of connected modem (i.e., the shoreside modem), the node number of the modem that will be communicated with (i.e., the modem on the AUV), and the port to connect to the shoreside modem. The port to connect to is always 1100. Next, the datagram message must be formed that consists of an array of single byte integer values. The datagram message to send, for example, is “A”, “B”, “O”, “R”, “T”. Each letter is transmitted in the form of a byte corresponding to the associated ASCII character value. The `unetsocket_open()` establishes a gateway connection with the API and sets the Datagram service as the agent. After opening a socket connection using the `unetsocket_open()` function, the datagram message can be sent using the `unetsocket_send_reliable()` function, which takes as input the socket connection identifier that was created by the output of the `unetsocket_open()` function, the message sequence, the number of bytes to be transmitted, the address of the receiver node, and finally the type of information to be sent, which is DATA. When a datagram message is received by a Subnero modem, an acknowledgement message is automatically sent by the receiving modem. The `unetsocket_send_reliable()` function instructs the transmitting modem to listen for the
acknowledgement message and use the `DatagramDeliveryNtf` class or the `DatagramFailureNtf` class in the Unetstack API to flag as a successful or unsuccessful communication, respectively.

Over the socket connection, a gateway is opened with the Unetstack API using the `unetsocket_get_gateway()` function. Using the gateway, the program will be notified if the `DatagramDeliveryNtf` or `DatagramFailureNtf` is set by the modem and will print a corresponding message to the terminal window informing the user of successful completion. Before terminating the program, the socket connection must be closed using the `unetsocket_close()` function to return the connection back to other users and applications.

**Section 4.3: Using the Unetstack API to Remotely Receive Datagram Messages**

On the receiving side, such as the backseat computer of the AUV, a program needs to be created that interfaces with the Unetsocket API to receive programs remotely. The procedure for remotely receiving datagram messages is based on the `rxdata.c` sample found in [29]. Like the programs for transmitting datagrams, the `unetsocket_open()` function must first be used to open a socket connection with the receiving modem. Next, the `unetsocket_bind()` function is used to set the protocol for listening as DATA. The `unetsocket_receive()` function is called to obtain any datagrams received by the modem. Before the `unetsocket_receive()` function is called, a timeout value in milliseconds must be set using the `unetsocket_set_timeout()` function, which sets the amount of time that the `unetsocket_receive()` function pauses the program and waits for a message to be received. Because the amount of information being received is short, the timeout value is usually set at 20 seconds. Finally, the datagram message is stored as an array of single byte integers using the `fjage_msg_get_byte_array()` function. Once the datagram message is stored in the byte array, the program can process the data and use it for the purpose of the application.
Section 4.4: Acoustic Control of the AUV to Abort Current Mission

Whenever the AUV is submerged, the user does not have a way of controlling the behavior of the vehicle. If an issue is noticed after the deployment, the user cannot abort the current mission to retrieve the AUV to fix the issue. Instead, the user must wait for the mission to be completed to retrieve the vehicle, which can take a significant amount of time depending on the length of the mission. To alleviate the issue, a set of programs needs to be created that uses the shoreside modem to send a command that tells the AUV to abort the mission and wait at a certain location for retrieval.

The modem API allows the sending and receiving of datagrams, which are transmissions in the form of a byte sequence. On the user computer, a program is used that creates an Ethernet socket connection with the modem. Over the socket connection, the program tells the modem API to transmit a datagram of the following sequence: “A”, “B”, “O”, “R”, “T”. The description of the method for doing this was described previously. The transmission program is called return.c and can be found in Appendix C. After the creation of the transmission program, the reception program must be created on the backseat computer of the AUV.

A MOOS-IvP application, called iModemAbort, was created that continuously checks the modem for any datagram reception. The C/C++ program for this, called ModemAbort.cpp, can be found in Appendix C. A flowchart describing the procedure of iModemAbort is pictured in Fig. 4.2.
Fig. 4.2. Flowchart for iModemAbort.

The procedure starts with an Ethernet socket connection being created, and then the API function to receive datagrams is called. The method to do this was previously described in this chapter. If the required datagram sequence is received by the modem, the application will then change several variables in the MOOSDB. First, the DEPLOY and RETURN Boolean variables must be set to true. The two variables will initiate the return behavior specified in the behavior file of the mission. Next, the ENGAGE_IVPHELM and VEHICLE_UNDERWAY are set to “true”, while the variable MOOS_MANUAL_OVERRIDE is set to “false”, which gives the IvP-Helm control of the AUV using iOceanServerComms. However, these sequences will only command the AUV to move to the return point specified by the behavior file. Once the return waypoint is reached, the original frontseat mission will restart from its last location. Therefore, NEXT_DESired_WAYPOINT is set to a specific value in the frontseat mission waypoints, preferably the endpoint value. This variable tells iOceanServerComms to send an $OJW sentence
to the frontseat computer, which commands the frontseat computer to skip to the waypoint specified. Once the endpoint is reached, the frontseat computer will end its mission, park at the endpoint, and wait for retrieval.

To test the set of programs, the experimental setup discussed at the beginning of this chapter was undertaken at Lake Tuscaloosa in Tuscaloosa County, AL. A frontseat mission was created with 17 waypoints, which performed a loop with some parts submerged. Therefore, the variable NEXT_DESIZED_WAYPOINT was set to be 17 if the abort sequence was received. The frontseat mission was run twice. First, the mission was run entirely without aborting the mission in order to provide a comparison. Second, the mission was rerun and then aborted using the return.c and iModemAbort programs. The frontseat mission was aborted soon after the vehicle submerged. In Fig. 4.3, the GPS coordinates saved by the AUV are plotted for both missions.

![Comparison of original and aborted mission tracks.](image)

Fig. 4.3. Comparison of original and aborted mission tracks.

The orange track shows the movement of the original mission, while the blue track shows the movement of the AUV after aborting the mission. As can be seen, after receiving the command, the AUV returned to, and parked, at the endpoint of the frontseat mission. Therefore, it can be concluded that the mission abort programs were verified to be successful.
Section 4.5: Acoustic Control of the Modem Signal Power Level

When starting a set of acoustic transmissions, the signal power level of the modem needs to be set to the proper value. When turning the Subnero modem on, the signal power level is automatically set to -42 dB. However, this signal level is too low for long distance transmissions, which is often the case with acoustic communications research. Often, the signal power level is preferred to be in the range of -10-0 dB, and 0 dB is the maximum power level available. Sometimes the AUV is deployed without setting the proper power level, which will make experimental results more difficult or impossible to capture. Therefore, it is helpful to create a set of programs that will use acoustic waveform transmissions to set the signal power level of the modem onboard the AUV remotely.

The first program to create is the transmission program on the user computer on the shore. The program is a C file called `setplvl.c` and can be found in Appendix D. Based on the method discussed previously for using the Unetstack API to command the Subnero modem to transmit datagrams, the transmission script uses a socket connection with the shoreside modem to send a datagram message using acoustic waveforms. The transmission program takes a command line argument for the desired power level, which is entered as the absolute value. The reason for using the absolute value is because the datagrams, which are single byte values, do not allow for negative number representations. Because the Subnero modem only allows negative numbers, it can be assumed that any positive entry represents its negative counterpart. The datagram message takes the form “P”, “L”, “V”, “L”, “#”, where “#” is the absolute value of the desired power level in dB. The datagram message is transmitted acoustically, where it is received by the modem on the AUV.
The MOOS-IvP application running on backseat computer is called *iModemPower*, which constantly checks the modem for datagram reception. The C/C++ program that governs the behavior of the application is called *ModemPower.cpp*, which can be found in Appendix D. The procedure used by *iModemPower* is described as a flowchart in Fig. 4.4.

![Flowchart for iModemPower](image)

Fig. 4.4. Flowchart for *iModemPower*.

The program is based on the method for using the Unetstack API to receive datagram messages remotely as described previously. If the modem receives the desired datagram to change the power level, the application then sends the dedicated API command to change the power level called *unetsocket_ext_set_powerlevel()*). The function is not found in the usual *unet.h* or *fjage.h* function libraries. Instead, the function is found in the extended API library called *unet_ext.h*. To access these functions, it was found to be easier to copy the *unet_ext.c* and *unet_ext.h* to the *iModemPower* source directory. Only the function *unetsocket_ext_set_powerlevel()* and the other functions it calls when operating were kept in the file, both of which can be found in Appendix D. The *unetsocket_ext_set_powerlevel()* function
takes three input arguments. The first argument is the identifier of the socket connection. The second input argument is the index of the channel requested to change the power level. The current Subnero modem in use has four channels assigned numbers 1 through 4. Each channel represents a different logical communication channel. For instance, channel 1 represents the CONTROL channel, while channel 2 represents the DATA channel [32]. The CONTROL channel is described in [27] as a “low-rate … communication link that is used for control information and link negotiation”, while the DATA channel is described as an “adaptively tuned high-rate communication link for large data transfer”. The third input argument is the desired power level, which must be in its negative form. In the unetsocket_ext_set_powerlevel() function, the fjage_agent_for_service() sets the PHYSICAL service as the agent to receive the request. A fjage request message is created to set the powerLevel variable in the PHYSICAL service as the new value. In the ModemPower.cpp program, the unetsocket_ext_set_powerlevel() is called four times for each communication channel. However, it was noticed that when using the Subnero shell environment that the plvl command, which is used to print or set the Subnero modem power level when using the shell environment, contains five different power level variables instead of four. The fifth variable is called signalPowerLevel and is found in the BASEBAND service instead of the PHYSICAL service. The Unetstack API does not have a function dedicated to changing the physical signal power level. Therefore, a new Unetstack API function, called unetsocket_ext_set_powerlevelBaseband(), was written in unet_ext.c to accomplish this. Similar to the original program, the fjage_agent_for_service() function sets the BASEBAND service to be the agent to receive the request. The message is created to request the signalPowerLevel variable to be changed to the desired power level value. The new Unetstack function is called by ModemPower.cpp to set the desired power level.
During an experiment at Lake Tuscaloosa, AL, the \textit{iModemPower} application was tested. The backseat mission, which calls the \textit{iModemPower} application, was launched. A Groovy script, called \textit{plvl-logging.groovy}, was started on the payload modem to monitor the \textit{phy.signalPowerLevel} parameter of the modem. The script captures the signal power level every 15 seconds and saves the value, along with a timestamp to a log file on the Subnero modem. The Groovy script can be found in Appendix D. Next, the AUV was deployed and the shoreside modem was set up to run the \textit{setplvl.c} code. Once the AUV submerged, the \textit{setplvl.c} code was executed on the user computer at the shore, and the power level was desired to be -10 dB. After the AUV was finished with the experiment and retrieved, the log file was downloaded, and the results were plotted. The log file, called \textit{power-log.txt}, containing the results can be found in Appendix D. The plot of the signal power level during the experiment is captured in Fig. 4.5.

![Power Level during Experiment](image)

\textbf{Fig. 4.5.} Signal power level of the modem during experiment.

As can be seen in Fig. 4.5, the power level starts at -42 dB, which is the value at startup of the modem. After the \textit{setplvl.c} program is executed at around 15:21 UTC, the \textit{phy.signalPowerLevel} value was set to the desired power level of -10 dB. Therefore, it can be concluded that the \textit{iModemPower} application was successful at setting the \textit{phy.signalPowerLevel} of the modem based upon acoustic commands.
Section 4.6: Acoustic Control of the AUV to Change Current Mission

It is sometimes warranted that the AUV mission needs to change after deployment. Previously, once the AUV is deployed with a certain mission track, the current mission track must be completed in its entirety before a new mission is started. However, because the iOceanServerComms application gives the backseat computer the ability to control the movement of the AUV based on new commands from IvP Helm, a new mission can be set with a different desired track. Using programs similar to previous applications, the user can acoustically control the AUV and specify which particular mission track he or she desires.

The shoreside transmission program takes in user input on the command line, such as the reference number of the mission to run. A socket connection is made with the shoreside modem to transmit a datagram message using the setmission.c program, which is based upon the method described previously on using the Unetstack API to command the AUV to transmit. The setmission.c program is found in Appendix E. The datagram message takes the form “D”, “P”, “L”, “Y”, “#”, where “#” represents the reference number of the desired mission. After transmitting the datagram message, the modem on the AUV will receive the message.

To retrieve and process the message, a MOOS application was created called iModemDeploy. The behavior of the iModemDeploy application is determined by the ModemDeploy.cpp program, which can be found in Appendix E. The procedure used by iModemDeploy is pictured as a flowchart in Fig. 4.6.
Like the MOOS application for aborting the current mission, the application constantly listens for datagram messages based on the method discussed previously for using the Unetstack API to receive messages remotely from a Subnero modem. If the desired datagram message is received, a new variable in the MOOSDB is created called MISSION, which is assigned the desired mission reference number. Also, the DEPLOY variable is set to “true” to start the desired mission. Finally, the ENGAGE_IVPHEME and VEHICLE_UNDERWAY are set to “true” and MOOS_MANUAL_OVERRIDE is set to “false” to grant control of the AUV to the backseat computer. The desired missions are instantiated in the mission behavior file called current.bhv, which can be found in Appendix E. Each mission is set to run on the condition of if the reference number of the mission is met. For example, for the behavior named waypt_survey3, the condition command is called in the following way

\[
\text{condition} = \text{MISSION} = 3.
\]
If the necessary condition is met, the backseat mission will then start the desired mission and run the mission either to completion or until another acoustic command is sent to the AUV by the shoreside modem to change the behavior of the AUV.

A field experiment was performed at Lake Tuscaloosa, AL to test the functionality of the iModemDeploy application. Three different mission tracks were created for the backseat behavior. The missions, called waypt_survey1, waypt_survey2, and waypt_survey3, were designed to take over the vehicle from its frontseat mission, surface, move to a position 330 meters to the east of park location or dock, and move 30 meters north, south, or east, respectively. The setmission.c program was executed multiple times, but only two missions were actually carried out. The reason for the unreliability has not been discovered and will need to be addressed in future studies. Fig. 4.7 shows a comparison of the original frontseat mission track and the AUV track after the iModemDeploy application-initiated changes in the mission.

![Comparison of original frontseat mission and mission modified by iModemDeploy.](image)

Both tracks are created from the GPS location data saved to the log files created by the frontseat computer. The color of the track represents the depth from the surface of the AUV. If a particular
section of the track is blue, the AUV was on the surface of the lake. If the track is red, the AUV was submerged at a depth of five meters or below. In the original frontseat mission, the AUV left the dock and submerged to a depth of five meters. After submersion, the AUV then moved to the center of the lake and looped around multiple times before returning to the dock. The length of the mission was chosen to give ample time to test the multiple missions. Fig. 4.7 shows three areas of difference from the frontseat mission. Two of these can be attributed to \textit{iModemDeploy} and the third and final difference can be attributed to \textit{iModemAbort}, which was called to abort the mission. In Fig. 4.8, the modified track of the AUV is annotated to better explain the movement of the vehicle to the reader.

Fig. 4.8. Annotated track of the AUV modified by \textit{iModemDeploy}.

It is important to note that the \textit{waypt\_survey1} did not complete its full objective of moving 30 meters north. Instead, it made a small loop, travelled southeast, and then returned to the frontseat mission. In this instance, the explanation of the behavior of the AUV has not been uncovered. Further study, including more robust mission tracks, will need to be performed to determine the cause of the error. However, the \textit{waypt\_survey3} did complete the mission of objective of moving 30 meters east from the center of the frontseat loops. Even though the mission behavior does not
entirely match the expected behavior, it can still be concluded that the \textit{iModemDeploy} application did indeed correctly initiate changes in the movement of the AUV based on user transmissions from the shore.

\textbf{Section 4.7: Acoustic Inquiries for Information from the AUV}

Previously, whenever the AUV was submerged, there was no method to interrogate the AUV for information. The current GPS location, current depth, or environmental data could only be saved to a log file and accessed after retrieval of the AUV. In order to obtain real time data from the AUV, a set of programs needed to be created that created two way communication between the AUV modem and the shoreside modem.

First, an interrogation program was created on the user computer that requested information from the AUV and listened for a response. The interrogation program, called \textit{depth.c}, is found in Appendix F. The procedure used by \textit{iModemInfo} is pictured as a flowchart in Fig. 4.9.

\begin{center}
Fig. 4.9. Flowchart for \textit{iModemInfo}.
\end{center}
Currently, the requested data is the current depth of the AUV; however, any value that is saved in the MOOSDB can be accessed, so the depth.c program can be easily modified to request any data. The program initiates a socket connection with the shoreside modem and commands that a datagram message be sent based upon the method previously described to use the Unetstack API to remotely transmit the Subnero modem. The datagram message that is sent is “I”, “N”, “F”, “O”, “D”. Once the transmission is complete and an acknowledgement is received, the program closes the current socket connection and reinitiates the socket connection to listen for the requested data sent in the form of a datagram response. The section of the program for receiving the datagram response is based upon the method described earlier for using the Unetstack API to remotely listen for datagram messages. The datagram response, which is a sequence of eight bytes that is the byte representation of the integer depth value, needs to be converted to a double variable, which is the decimal value for the depth. To do this, the message array is saved as a union, which is a data structure in C/C++ that allows access to both the byte representation of the number and the integer version of the number. The integer version is then printed to the terminal for the user.

The MOOS application on board the AUV for responding to the interrogation is called iModemInfo. The main code file for the MOOS application is called ModemInfo.cpp, which can be found in Appendix F. The application registers and receives the value of the MOOSDB variable called TRUE_DEPTH, which contains the current depth of the AUV underwater. The value is assigned to a union, so the byte level representation of the variable can be accessed. Also, the application constantly listens for the interrogation datagram message using a socket connection with the modem. Once the application receives the message, the socket connection is closed. A new socket connection is opened in order to send the requested data as a response in
the form of a datagram message. The byte representation of the depth value is assigned to be the response datagram message. The datagram is then transmitted from the AUV for reception at the shoreside.

An experiment was carried out at Lake Tuscaloosa, AL to test the *iModemInfo* application. A backseat MOOS mission was created that included the application. A frontseat mission was designed to control the movement of the vehicle, which was set to move the AUV in a looping pattern at a depth of 5 meters below the surface. The track of the AUV obtained after the experiment is plotted in Fig. 4.10, and the color of the track shows the depth of the AUV below the surface.

![Recorded AUV Track of Mission](image)

**Fig. 4.10.** Recorded mission track of the AUV during experiment.

As can be seen in Fig. 4.10, the AUV maintained a depth of 5 meters for the majority of the mission. Before the AUV was launched, the backseat mission was also launched, which waited for the interrogation signal that would emanate from the shore modem. After the AUV was
submerged, the *depth.c* code was executed on the user computer at the shore. A response was received from the payload modem on the AUV, which is captured in Fig. 4.11.

Fig. 4.11. Output and response from the interrogation program.

According to Fig. 4.11, the responded depth was approximately 4.97 meters. Because the responding value is close to the value AUV depth of 5 meters, it can be concluded that the *iModemInfo* application did successfully respond to the interrogation with accurate data.

**Section 4.8: Forwarding Information through Underwater Acoustic Networks**

Underwater acoustic networks of AUVs require communication between different nodes, which are separate AUVs, in the network. Often, the nodes in the network share location information with the other nodes, which provides a gauge of the relative movement of the nodes. The simplest method of sharing is forward information from one node to another. As an illustrative example, Fig. 4.12 shows the forwarding method described in this section.

Fig. 4.12. Method for forwarding information across the network.
As can be seen from Fig. 4.12, the forwarding path forms a ring, where each node forwards any information received to the next node in the ring while appending its own data at the end. Because it is imperative that each node knows the next node to forward the information, a list of the nodes must be sent first to each node in the order of the forwarding ring. To implement the forwarding method, a C program was written that uses the API of the shoreside modem to forward the transmit the list and initiate the forwarding process. Also, a MOOS application was created that forwards the list and the information to the next node in the network.

The C program on the user computer uses the API of the shoreside modem to remotely initiate the forwarding sequence. The C program is called broadcastList.c and can be found in Appendix G. First, the ordered list of nodes is transmitted using the method described earlier for remote transmission of datagrams. The datagram message starts with the letters “L”, “I”, “S”, and “T” as an identifier, which is followed by the ordered list of nodes. The list starts with the first node to receive the message, continues through each node as specified by the user, and ends with the node number of the shoreside modem. For instance, the network forwarding scheme described in Fig. 4.12 would require the message


The list can be set by the user in a text file, which is read by the C program using the fopen() standard library function. Once the list is transmitted, the socket connection is closed and reopened for safety purposes. Then, another message is transmitted to the first node on the network forwarding list using the method described previously. The list will be a 150 bytes empty message, which will be filled in by each node with the location information. Next, the shoreside modem will then wait to receive the forwarded information. In order to account for delay, the timeout for reception is two minutes. First, the C program must wait for the ordered
forwarding list to be received because it is the final node in the ordered list. Once this reception is complete, the C program can then wait to receive the final datagram message containing a filled in list of location values. Because each node appends to the datagram message a sentence with a set syntax, the location values can be easily isolated from the datagram message. The union data structure is used to easily convert between byte array and floating point number for the latitude, longitude, and depth. Finally, the node location and the associated node number are saved to a file for the user to access later.

On the backseat computer of the AUV, a MOOS application was written that receives the ordered node list and the list of the node location information and forwards the information to the next node in the network list. The MOOS application is called iModemForward. The main file, called ModemForward.cpp, can be found in Appendix G. In the registerVariables() function, the three location variables in the MOOSDB, NAV_LAT, NAV_LONG, and NAV_DEPTH, are registered by iModemForward and saved as variables in the OnNewMail() function. The Iterate() function forms the main code of the program. Fig. 4.13 is a flowchart of the process used by iModemForward for receiving the information.
First, the program uses the remote receiving method described previously to wait for the ordered list of nodes. Because of the potential length of the datagram messages, the timeout for waiting was set to two minutes. Once the ordered list of nodes is received, the socket connection is closed. A check of the first four values in the datagram message is performed to determine if the program should act on the message or disregard the message. If the correct datagram message was received, the application will then reopen the socket connection with the modem and start the remote receiving method again to receive the list of the node location information. Once the list is received in full, the application can begin the process of forwarding the information to the next node.

The forwarding procedure used by \textit{iModemForward} is pictured as a flowchart in Fig. 4.14.
Fig. 4.14. Forwarding flowchart for *iModemForward*.

To forward the information to the next node, the first step is to determine the node address number of the next node in the list. A linear search of the node list is performed to find the node address of the modem connected to the backseat computer, and the next node address number in the list is set to be the destination of the forwarded list transmission. First, a socket connection is established with the modem and the Unetstack API method for remotely transmitting a datagram message is used to forward the node list. After completing the transmission and closing the socket connection, the node location information is appended to the received information list. Because the information list is uniform in syntax, the appending of the data can be performed using precalculated array locations in a relatively simple manner. Finally, the completed information list can be transmitted to the next node in the network using the remote transmitting method described previously.
Multiple attempts were made during the experiments at Lake Tuscaloosa, AL to test the \textit{iModemForward} application. Each attempt ended in failure. In order to test the application, a laboratory experiment and a field test experiment at Lake Palmer were performed. For the laboratory experiment, two Iver3 AUVs were placed in a water tank with the bow of the AUV on the bottom of the tank. The laboratory experiment setup is pictured in Fig. 4.15.

The water tank was filled until the water covered the transducer on top of the AUV. Similar to the field tests, the surface modem was placed in the water and connected with an Ethernet cable to a laptop computer. The node address numbers associated with each piece of equipment is captured in Table 4.1.
Table 4.1
List of Nodes in Laboratory and Lake Palmer Experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>IP Address</th>
<th>Node Address Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iver3-3083 Embedded Unit</td>
<td>192.168.1.101</td>
<td>225</td>
</tr>
<tr>
<td>Iver3-3093 Embedded Unit</td>
<td>192.168.1.149</td>
<td>69</td>
</tr>
<tr>
<td>Surface Unit</td>
<td>192.168.1.127</td>
<td>97</td>
</tr>
</tbody>
</table>

Table 4.1. List of the IP addresses and node addresses in the laboratory and Lake Palmer field experiment.

The laboratory experiment involves one major drawback, which is the inability to obtain GPS readings inside the building. Because the GPS values will be zero, the solution is to set the location values manually. To do this, a MOOS application is used called *uPokeDB*, which allows the user to manually change variable values in the MOOSDB from the terminal window. If we want to change the value of the NAV_LAT variable in the MOOSDB to a value of 33.291, we can type in the terminal window the following:

```
uPokeDB NAV_LAT=33.291
```

The value of the variable will then change to the desired value. However, it is important to note that *iOceanServerComms* is in constant communication with the frontseat computer; therefore, *iOceanServerComms* will immediately overwrite any change to the NAV_LAT variable back to zero. To alleviate the overwriting issue, the *iOceanServerComms* application must be turned off during a laboratory experiment, which can be done by removing the call to run the application in the .moos file for the mission. Therefore, the laboratory experiment can only test the *iModemForward* application individually and not as a part of the larger MOOS Community system, which includes the *iOceanServerComms* application. Thus, the need of a field test is warranted.
During the laboratory test, the values changed by *uPokeDB*, and their final values, are captured in Table 4.2. In order to test the full functionality of the *iModemForward* application, the values were changed on both AUVs. The values are representative of GPS and depth values obtained during missions at Lake Tuscaloosa.

<table>
<thead>
<tr>
<th>Node Number</th>
<th>NAV_LAT</th>
<th>NAV_LONG</th>
<th>TRUE_DEPTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>225</td>
<td>33.291</td>
<td>-87.519</td>
<td>0.316</td>
</tr>
<tr>
<td>69</td>
<td>33.291</td>
<td>-87.518</td>
<td>3.083</td>
</tr>
</tbody>
</table>

Table 4.2. Variables and final values changed by *uPokeDB*.

The node routing list for the laboratory experiment is first to Node 225, then to Node 69, and back to Node 97. The output of the laboratory experiment was printed to the screen and is captured in Fig. 4.16. The output was also logged in the file *labtest5.txt*, which can be found in Appendix G.

As can be seen by the results captured in Fig. 4.16, the values that were set in the MOOSDB by *uPokeDB* were transmitted through the network based on the list. One can see that Node 69 transmitted to Node 97, which is the shore node, both its GPS and depth information as well as the location information of Node 225, which was obtained by Node 69 by the operation of *iModemForward*. Therefore, it can be concluded from the results of the laboratory experiment
that the *iModemForward* application is functionally correct; however, the application would still need to be tested in the field to provide a test of its functionality in the overall system along with iOceanServerComms.

In order to do a successful field test, two Iver3 AUVs were brought to Lake Palmer on the campus of the University of Alabama. The Lake Palmer location was chosen for the experiment due to its proximity on the campus. Because Lake Palmer is shallower than Lake Tuscaloosa and the AUVs were kept at the dock and in close proximity to the transmitter, Lake Palmer provided a better acoustic environment and, therefore, more promising results in the field. Both AUVs were setup to run a backseat mission with *iModemForward*. The name and node numbers of each node in the network are the same as the ones used in the laboratory experiment, which can be found in Table 4.2. The designated sequence was first to Node 225, then to Node 69, and finally back to Node 97 at the dock. After running the `broadcastList.c` program, the results were captured on the screen, which is pictured in Fig. 4.17, and also in `output613.txt`. The text file can be found in Appendix G for reference.

![Output from iModemForward field test.](image)

The results shown in Fig. 4.17 show that the broadcast was successful, which means that each node in the network received and then forwarded the message to the correct next node on the list. The GPS and depth data were printed to the screen. The information from Node 225 shows GPS coordinates that match expected values. However, Node 69 shows zeros for each value. It was
determined that the *iOceanServerComms* application was not correctly obtaining the information from the frontseat computer. The root cause has not yet been determined. Because Node 69 did transfer the information from Node 225 to Node 97, it can be concluded that the *iModemForward* application did conduct a multi-hop transmission as intended.
CHAPTER 5

CONCLUSION

The main emphasis of this work is to show the ability to use a software defined underwater acoustic modem to remotely control the operation of an autonomous underwater vehicle (AUV) using acoustic transmissions. This work gives a thorough overview of the steps and applications necessary to obtain this goal. Also, multiple experiment were performed and detailed in this work to verify and validate the ability of these applications to realize this goal.

First, the reader was introduced to the MOOS-IvP software package that is used as an autonomy software on many different aquatic vehicle platforms. In this case, the Iver3 vehicle was used, and the MOOS application iOceanServerComms was used to form a link between the MOOS-IvP autonomy software running on the backseat computer of the vehicle and the Underwater Vehicle Console (UVC), which controls the vehicle from the main computer. Because the iOceanServerComms application was specifically designed for the older model Iver2 vehicle, some issues were discovered and corrected in the course of integrating the two computers into a single system. After the corrections were made, an experiment was performed that showed the autonomy software on the backseat computer could control the movement of the vehicle.

After installing a Subnero software defined underwater acoustic modem as the payload of the Iver3 vehicle, a method was found to inform the modem of its location, which includes the depth below the surface. Because the iOceanServerComms application forms a link between the secondary and main computers, the application queries the main computer for sensor
information, including compass and GPS location information, which is then stored in the MOOSDB database. By creating a MOOS application called *iModemComms* that uses the Unetstack API over an Ethernet socket connection as an interface to the modem, the *iModemComms* can set the values of the modem location parameters to the values stored in the MOOSDB database. Once the application was created, some issues with the parsing function of *iOceanServerComms* were discovered and corrected. The *iModemComms* application was used in an underwater acoustic communications experiment to provide location information to modem so that the modem could make an informed decision of the safety of a given transmission. The created application was shown to perform the necessary task.

Using the methods for remotely transmitting and receiving datagram messages in the Unetstack API, the goal was to create multiple different applications that would allow for acoustic control of the vehicle along with acoustic communications with the vehicle. First, an application called *iModemAbort* was created that allowed the user to send acoustic datagram messages that commanded the vehicle to abort its current mission, continue to a designated location, and await retrieval of the vehicle. In that same vein, the *iModemDeploy* application allowed the user to command the vehicle to change between different predefined missions using acoustic commands. Next, the *iModemPower* application allowed the user to remotely command the modem to change the transmitted signal power level. Then, the *iModemInfo* application was created which responds to the query of the user for depth information. This application was then expanded in scope to become *iModemForward*, which shared vehicle location information throughout a multiple AUV network. Multiple experiments were performed that verified that the applications performed the required goals.
There is room for improvement of the applications presented in this work. First, the reliability of the programs should be increased in the future. In field test, the applications would occasionally not perform as expected. For instance, the iModemDeploy application enabled the backseat missions to take control of the vehicle, but the particular track chosen by the IvP-Helm in these missions is not readily apparent and needs to be further investigated. Each application could benefit from a shorter preamble identifier at the beginning of the message, such as a single byte as used in the CCL language in [18] and in other systems like the one described in [20]. The decrease in the identifier will shorten the overall message and provide a higher probability of successful transmission, as explained in [20]. Also, the iModemForward has not successfully passed a test in a long range field scenario. In order to increase the probability of success, the scheme employed by iModemForward will need to be reevaluated, including sending shorter messages between nodes. Finally, the implementation of MAC protocols, like in [12], and self-adaptive protocols, like in [7], should be researched in the further development of the underwater network.

The applications described in this work form the first steps toward using the Iver3 AUV as a mobile platform for acoustic communications and networking research. These applications can be used to form the basis of developing any other application that is needed in the development of these platforms. Also, the applications will be used future underwater acoustic communications experiments. This includes experiments involving multiple AUVs deployed in a network, and also synchronized acoustic transmission experiments involving multiple AUVs. These applications greatly assist the state of the art of using AUVs for underwater acoustic communications experiments.
REFERENCES


5. L3Harris Oceanserver, Iver3 AUV Spec Sheet, Fall River, MA., USA, 2019.


22. L3Harris Oceanserver, Remote Helm Manual – Backseat Commands - Revision 5.3.3, Fall River, MA., USA, 2019.


APPENDIX A

FILES FOR INTRODUCTION TO MOOS-IvP

The *current.bhv* file is the file used in the Bravo mission at Lake Palmer to test the integration of the frontseat and backseat computers, and it contains the behavior for the mission. The *last.moos* file is the MOOS application configuration file generated for the Bravo mission.

```
****** current.bhv ******
//-------------------------------------------------------------------------------
// Behavior (BHV) File for IVER Operations (generated)
initialize   DEPLOY = false
initialize   RETURN = false

//-------------------------------------------------------------------------------
Behavior = BHV_Waypoint
{
  name      = waypt_survey
  pwt       = 100
  condition = RETURN = false
  condition = DEPLOY = true
  endflag   = RETURN = true
  perpetual = true
  updates   = NEWPTS

  lead = 8

  lead_damper = 1
    speed = 0.5  // meters per second
    radius = 2.0

  //
  points = zigzag:35,45,90,1000,200,50
  points = 25,0:0,0:25,0:0,0:25,0:0,0
  repeat = 2
  visual_hints = nextpt_color=red, nextpt_lcolor=green
  visual_hints = vertex_color=yellow, line_color=white
  visual_hints = vertex_size=2, edge_size=1
}
```
Behavior = BHV_Waypoint
{
    name       = waypt_return
    pwt        = 100
    condition  = RETURN = true
    condition  = DEPLOY = true
    updates    = UPDATES_RETURN
    perpetual  = true
    endflag    = RETURN = false
    endflag    = DEPLOY = false

    speed = 0.5
    radius = 2.0
    nm_radius = 8.0
    points = 0,0
}

Behavior = BHV_ConstantDepth
{
    // General Behavior Parameters
    // ---------------------------
    name         = const_dep_survey   // example
    pwt          = 100                // default
    condition    = DEPLOY==true
    updates      = CONST_DEP_UPDATES  // example

    // Parameters specific to this behavior
    // -------------------------------------
    basewidth = 100          // default
    depth = 5            // default
    depth_mismatch_var = DEPTH_DIFF // example
    duration = no-time-limit // default
    peakwidth = 3            // default
    summitdelta = 50           // default
}

Behavior = BHV_ConstantDepth
{
    // General Behavior Parameters
    // ---------------------------
    name       = const_dep_return  // example
    pwt        = 100                // default
    condition  = RETURN==true    // example
    updates    = CONST_DEP_UPDATES  // example
// Parameters specific to this behavior
// ------------------------------------
basewidth = 100          // default
depth = 0            // default
depth_mismatch_var = DEPTH_DIFF // example
duration = no-time-limit            // default
peakwidth = 3            // default
summitdelta = 50       // default

******* last.moos *******
// -----------------------------------------------------------------------------------------------
---
// MOOS File for IVER Operations (generated)
ServerHost = localhost
ServerPort = 9000
Simulator = false
Community = iver
LatOrigin  = 33.290758
LongOrigin = -87.519400
//-------------------------------
---
ProcessConfig = ANTLER
{
  MSBetweenLaunches = 200

  //Essential
  Run = MOOSDB @ NewConsole = false
  Run = iOceanServerComms @ NewConsole = false
  Run = pHelmIvP @ NewConsole = false

  //Recommended
  Run = pNodeReporter @ NewConsole = false
  Run = uProcessWatch @ NewConsole = false

  //Logging
  Run = pLogger @ NewConsole = false

}
//-------------------------------
---
ProcessConfig = iOceanServerComms
{
    AppTick = 1
    CommsTick = 5

    // Frontseat Serial Port Configuration
    port = /dev/ttyS0
    BaudRate = 19200
    handshaking = false
    streaming = false

    // Use Local UTM coordinate system?
    UTM = false

    // Request data from DVL?
    DVL = false

    // Request general CTD data?
    CTD = true

    // Request data string from YSI Sonde?
    YSI = true

    // Are we in salt water? (Used for YSI sound velocity calculation.)
    SaltWater = false

    // Post NAV_* variables?
    PostNav = true

    // Average NAV_WATERDEPTH?
    AvgWaterDepth = false

    // Constraints
    CommTimeout = 2 // seconds
    ServoFilter = 0 // ticks
}

// ------------------------------------------------------------
// ProcessConfig = pHelmIvP
{  
    AppTick = 4
    CommsTick = 4

    Domain = course,0:359:360
    Domain = speed,0:1.8:16
Domain = depth,0:100:101:optional

Behaviors = current.bhv //Usually matches community name.

OK_SKEW = ANY

//-----
ProcessConfig = pNodeReporter
{
  AppTick = 5
  CommsTick = 5

  VESSEL_TYPE = AUV
}

//-----
ProcessConfig = uProcessWatch
{
  AppTick = 1
  CommsTick = 5

  watch_all = antler
}

//-----
ProcessConfig = pLogger
{
  AppTick = 10
  CommsTick = 10

  File = iver_ //Usually matches the community name.

  PATH = ./data_from_runs/
  SyncLog = true @ 0.2
  AsyncLog = true
  FileTimeStamp = true
  WildCardLogging = true
  //WildCardOmitPattern = DB_TIME,DB_UPTIME,DB_CLIENTS
}
APPENDIX B

FILES FOR iModemComms

The ModemComms.cpp files contain the code that governs the behavior of the iModemComms application. The CMakeLists.txt file contains the changes that needed to be made to create MOOS applications using CMake. To create the new SHELL agent, the startup.groovy script was used on the Subnero modem. The s2_adaptive_script_long.groovy and its associated log file s2-log.txt tested the iModemComms application. To fix the depth issue, parse.cpp of the iOceanServerComms application was changed. For brevity, the corrected parse.cpp file was shortened to only include the section where the State string is parsed. All the other parts of the original file remain unchanged. Finally, s3_adaptive_script_short.groovy and its associated log file s3-log.txt was used to test the application after correcting the depth issue.

***** ModemComms.cpp *****
/*****************************/
/* NAME: Connor Webb        */
/* ORGN: University of Alabama */
/* FILE: ModemComms.cpp     */
/* DATE:                    */
/*****************************/

#include <iterator>
#include "MBUtils.h"
#include "ACTable.h"
#include "ModemComms.h"

extern "C" {
    #include "fjage.h"
    #include "unet.h"
}

#define BUF_SIZE 12

74
using namespace std;

// Constructor

ModemComms::ModemComms()
{
    // initialize GPS and depth info to 0
    current_x = 0.0;
    current_y = 0.0;
    current_depth = 0.0;
}

// Destructor

ModemComms::~ModemComms()
{
}

// Procedure: OnNewMail

bool ModemComms::OnNewMail(MOOSMSG_LIST &NewMail)
{
    AppCastingMOOSApp::OnNewMail(NewMail);

    MOOSMSG_LIST::iterator p;
    for(p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key = msg.GetKey();

        // Keep these around just for template
        string comm = msg.GetCommunity();
        double dval = msg.GetDouble();
        string sval = msg.GetString();
        string msrc = msg.GetSource();
        double mtime = msg.GetTime();
        bool mdbl = msg.IsDouble();
        bool mstr = msg.IsString();

        #if 0
        #endif

        if (key == "NAV_LAT") {
            current_x = msg.GetDouble();
        }
        else if (key == "NAV_LONG") {
            current_y = msg.GetDouble();
        }
else if (key == "TRUE_DEPTH") {
    current_depth = msg.GetDouble();
}
else if(key != "APPCAST_REQ") // handled by AppCastingMOOSApp
    reportRunWarning("Unhandled Mail: " + key);

return(true);

//---------------------------------------------------------
// Procedure: OnConnectToServer

bool ModemComms::OnConnectToServer()
{
    registerVariables();
    return(true);
}

//---------------------------------------------------------
// Procedure: Iterate()
//      happens AppTick times per second

bool ModemComms::Iterate()
{
    AppCastingMOOSApp::Iterate();
    // test values
    //current_x = 5;
    //current_y = 5;
    //current_depth = 5;

    // notify MOOSDB of the GPS and depth info being sent to modem
    Notify("MODEM_LAT", current_x);
    Notify("MODEM_LONG", current_y);
    Notify("MODEM_DEPTH", current_depth);

    unetsocket_t sock;
    fjage_aid_t node;
    int port = 1100;
    const char* ip_address = "192.168.1.101"; // IP address of modem (this may change!)

    // Open a unet socket connection to modem
    sock = unetsocket_open(ip_address, port);
    fjage_gw_t gw = unetsocket_get_gateway(sock);
// Get the agent which provides SHELL service
//fjage_aid_t aid = fjage_aid_create("shellForAuv");
fjage_aid_t aid = unetsocket_agent_for_service(sock,
"org.arl.fjage.shell.Services.SHELL");

// Update depth when available
// initialize the GPS and depth values to char strings of buffer size
char depth[BUF_SIZE] = {0};
char lat[BUF_SIZE] = {0};
char lon[BUF_SIZE] = {0};
// assign char strings to values of current depth and GPS
snprintf(depth, BUF_SIZE, "%f", current_depth);
snprintf(lat, BUF_SIZE, "%f", current_x);
snprintf(lon, BUF_SIZE, "%f", current_y);

// make char string to send to API for update
char* cmd;
asprintf(&cmd, "node.location=[%s, %s, %s];", lat, lon, depth);
//asprintf(&cmd, "ps");
fjage_msg_t msg =
fjage_msg_create("org.arl.fjage.shell.ShellExecReq",
FJAGE_REQUEST);
fjage_msg_set_recipient(msg, aid);
fjage_msg_add_string(msg, "cmd", cmd);
//fjage_msg_add_string(msg, "cmd", "ps");
fjage_msg_t rsp = fjage_request(gw, msg, 1000);
if (rsp != NULL) fjage_msg_destroy(rsp);
free(cmd);

fjage_aid_destroy(aid);

// Close the socket
unetsocket_close(sock);

//current_depth = current_depth + 5.0;

AppCastingMOOSApp::PostReport();
return(true);
}

//---------------------------------------------------------
// Procedure: OnStartUp()
// happens before connection is open
bool ModemComms::OnStartUp()
{
    AppCastingMOOSApp::OnStartUp();

    STRING_LIST sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
        reportConfigWarning("No config block found for " +
                            GetAppName());

    STRING_LIST::iterator p;
    for(p=sParams.begin(); p!=sParams.end(); p++) {
        string orig  = *p;
        string line  = *p;
        string param = tolower(biteStringX(line, '='));
        string value = line;

        bool handled = false;
        if(param == "foo") {
            handled = true;
        }
        else if(param == "bar") {
            handled = true;
        }

        if(!handled)
            reportUnhandledConfigWarning(orig);
    }

    registerVariables();
    return(true);
}

//----------------------------------------------------------------------------
// Procedure: registerVariables

void ModemComms::registerVariables()
{
    AppCastingMOOSApp::RegisterVariables();
    m_Comms.Register("NAV_LAT", 0);
    m_Comms.Register("NAV_LONG", 0);
    m_Comms.Register("TRUE_DEPTH", 0);
    // Register("FOOBAR", 0);
}

//----------------------------------------------------------------------------
// Procedure: buildReport()

bool ModemComms::buildReport()
{
    m_msgs << "Current Latitude: " << current_x << endl;
    m_msgs << "Current Longitude: " << current_y << endl;
    m_msgs << "Current Depth: " << current_depth << endl;
    m_msgs << "Sending to Modem..." << endl;
    return(true);
}

******* CMakeLists.txt *******
#--------------------------------------------------------
# The CMakeLists.txt for:                           iModemPower
# Author(s):                              Connor Webb
#--------------------------------------------------------

SET(SRC
    ModemPower.cpp
    unet_ext.c
    ModemPower_Info.cpp
    main.cpp
)

ADD_EXECUTABLE(iModemPower ${SRC})

TARGET_LINK_LIBRARIES(iModemPower
    ${MOOS_LIBRARIES}
    apputil
    mbutil
    m
    pthread)

    find_library(LIBFJAGE fjage /home/iver3-backseat/unet-
        contrib/unetsocket/c)
    find_library(LIBUNET unet /home/iver3-backseat/unet-
        contrib/unetsocket/c)
    target_link_libraries(iModemPower ${LIBFJAGE})
    target_link_libraries(iModemPower ${LIBUNET})
    include_directories(/home/iver3-backseat/unet-
        contrib/unetsocket/c)

******* startup.groovy *******
// startup.groovy
//
// Any groovy commands placed in this file will be executed at UnetStack startup.
// This file can be used to pre-configure the modem at boot.
//
// Uncomment the following section to load an AT command interpreter on TCP
/*
iface ATScriptEngine, 5001
*/

// Uncomment the following section to load an AT command interpreter on RS232
/*
iface ATScriptEngine, '/dev/ttyTHS1', 115200, 'N81'
*/

// Uncomment the following section to load a UnetStack API interface on RS232
/*
iface API, '/dev/ttyTHS1', 115200, 'N81'
*/

// Uncomment the following section to load RS232 transparent mode
/*
container.add 'portal', new org.arl.unet.portal.Portal(conn)
*/

// Uncomment the following section to load a Groovy command interpreter on RS232
/*
iface GroovyScriptEngine, '/dev/ttyTHS1', 115200, 'N81'
*/

import org.arl.fjage.shell.*;
container.add 'shellForAuv', new ShellAgent(new GroovyScriptEngine())

******* s2_adaptive_script_long.txt ******
import java.time.Instant

filename = 'logs/s2-log.txt'; // name of file

File file = new File(filename);
i = 1;
subscribe phy;
plvl = -10; // power level in dB
//plvl = -40; // indoor testing

// set the minimum depth
minDepth = 0.1;
//minDepth = 0.0; // indoor testing

// load the signals
uALAO3 = load("UALA-TX0SR10SN03PR0-ADAP5K.txt");

//startTime = Instant.now().epochSecond; // current time
//endTime = "2021-10-06T22:00:00Z"; // IMPORTANT: Enter start
time of transmission before starting script
//unixEndTime = Instant.parse(endTime).epochSecond; // convert
to UNIX
//numDelay = (unixEndTime - startTime) * 1000; // convert to
milliseconds
delay(4*60*1000); // delay until start time

// 120 times
120.times {
    file << i << ": ";
    if (node.location[2] > minDepth) {
        file << "UALAO3 ~ " << phy.rtc << " ~ " <<
        node.location;
        pbtx uALAO3;
    }
    delay(20*1000);
    file << "\n";
    i++;
}

******* s2-log.txt *******
1): UALAO3 ~ Thu Oct 07 18:45:57 UTC 2021 ~ [33.289026, -
87.514876, 0.113264]
2): UALAO3 ~ Thu Oct 07 18:46:17 UTC 2021 ~ [33.289211, -
87.514951, 0.114391]
3): UALAO3 ~ Thu Oct 07 18:46:37 UTC 2021 ~ [33.289385, -
87.515085, 0.114513]
4): UALAO3 ~ Thu Oct 07 18:46:57 UTC 2021 ~ [33.289538, -
87.515217, 0.114635]
5): UALAO3 ~ Thu Oct 07 18:47:17 UTC 2021 ~ [33.289687, -
87.515348, 0.114727]
6): UALA03 ~ Thu Oct 07 18:47:38 UTC 2021 ~ [33.289843, -87.515486, 0.114696]
7): UALA03 ~ Thu Oct 07 18:47:58 UTC 2021 ~ [33.28999, -87.515617, 0.114635]
8): UALA03 ~ Thu Oct 07 18:48:18 UTC 2021 ~ [33.290138, -87.515749, 0.114727]
9): UALA03 ~ Thu Oct 07 18:48:38 UTC 2021 ~ [33.290295, -87.515889, 0.114666]
10): UALA03 ~ Thu Oct 07 18:48:58 UTC 2021 ~ [33.29044, -87.516019, 0.114605]
11): UALA03 ~ Thu Oct 07 18:49:18 UTC 2021 ~ [33.290587, -87.51615, 0.114635]
12): UALA03 ~ Thu Oct 07 18:49:38 UTC 2021 ~ [33.290744, -87.516288, 0.114513]
13): UALA03 ~ Thu Oct 07 18:49:58 UTC 2021 ~ [33.29089, -87.516417, 0.114544]
14): UALA03 ~ Thu Oct 07 18:50:18 UTC 2021 ~ [33.291045, -87.516557, 0.114635]
15): UALA03 ~ Thu Oct 07 18:50:38 UTC 2021 ~ [33.291193, -87.516689, 0.114574]
16): UALA03 ~ Thu Oct 07 18:50:59 UTC 2021 ~ [33.29134, -87.51682, 0.114574]
17): UALA03 ~ Thu Oct 07 18:51:19 UTC 2021 ~ [33.291496, -87.516958, 0.114544]
18): UALA03 ~ Thu Oct 07 18:51:39 UTC 2021 ~ [33.291643, -87.517089, 0.114605]
19): UALA03 ~ Thu Oct 07 18:51:59 UTC 2021 ~ [33.291793, -87.517222, 0.114605]
20): UALA03 ~ Thu Oct 07 18:52:19 UTC 2021 ~ [33.291945, -87.517346, 0.113873]
21): UALA03 ~ Thu Oct 07 18:52:39 UTC 2021 ~ [33.292094, -87.517213, 0.113965]
22): UALA03 ~ Thu Oct 07 18:52:59 UTC 2021 ~ [33.292247, -87.51707, 0.114056]
23): UALA03 ~ Thu Oct 07 18:53:19 UTC 2021 ~ [33.292254, -87.516926, 0.113843]
24): UALA03 ~ Thu Oct 07 18:53:39 UTC 2021 ~ [33.292076, -87.516815, 0.114513]
25): UALA03 ~ Thu Oct 07 18:53:59 UTC 2021 ~ [33.291919, -87.516687, 0.114452]
26): UALA03 ~ Thu Oct 07 18:54:19 UTC 2021 ~ [33.291758, -87.516549, 0.114757]
27): UALA03 ~ Thu Oct 07 18:54:39 UTC 2021 ~ [33.291607, -87.516419, 0.114757]
28): UALA03 ~ Thu Oct 07 18:54:59 UTC 2021 ~ [33.291457, -87.516287, 0.114757]
29): UALA03 ~ Thu Oct 07 18:55:19 UTC 2021 ~ [33.291296, -87.516148, 0.114727]
30): UALA03 ~ Thu Oct 07 18:55:40 UTC 2021 ~ [33.291144, -87.516016, 0.114696]
31): UALA03 ~ Thu Oct 07 18:56:00 UTC 2021 ~ [33.290993, -87.515882, 0.114757]
32): UALA03 ~ Thu Oct 07 18:56:20 UTC 2021 ~ [33.290831, -87.515742, 0.114635]
33): UALA03 ~ Thu Oct 07 18:56:40 UTC 2021 ~ [33.290679, -87.515609, 0.114635]
34): UALA03 ~ Thu Oct 07 18:57:00 UTC 2021 ~ [33.290527, -87.515477, 0.114544]
35): UALA03 ~ Thu Oct 07 18:57:20 UTC 2021 ~ [33.290366, -87.515336, 0.114452]
36): UALA03 ~ Thu Oct 07 18:57:40 UTC 2021 ~ [33.290215, -87.515205, 0.114483]
37): UALA03 ~ Thu Oct 07 18:58:00 UTC 2021 ~ [33.290063, -87.515073, 0.114544]
38): UALA03 ~ Thu Oct 07 18:58:22 UTC 2021 ~ [33.289896, -87.514927, 0.114544]
39): UALA03 ~ Thu Oct 07 18:58:42 UTC 2021 ~ [33.289749, -87.514798, 0.114605]
40): UALA03 ~ Thu Oct 07 18:59:02 UTC 2021 ~ [33.289601, -87.514671, 0.114513]
41): UALA03 ~ Thu Oct 07 18:59:22 UTC 2021 ~ [33.289446, -87.514535, 0.114422]
42): UALA03 ~ Thu Oct 07 18:59:42 UTC 2021 ~ [33.289346, -87.514396, 0.115092]
43): UALA03 ~ Thu Oct 07 19:00:02 UTC 2021 ~ [33.289511, -87.514262, 0.113904]
44): UALA03 ~ Thu Oct 07 19:00:22 UTC 2021 ~ [33.289666, -87.514139, 0.114026]
45): UALA03 ~ Thu Oct 07 19:00:42 UTC 2021 ~ [33.289828, -87.514251, 0.114148]
46): UALA03 ~ Thu Oct 07 19:01:02 UTC 2021 ~ [33.290001, -87.514383, 0.114391]
47): UALA03 ~ Thu Oct 07 19:01:22 UTC 2021 ~ [33.29016, -87.5145, 0.114666]
48): UALA03 ~ Thu Oct 07 19:01:43 UTC 2021 ~ [33.290326, -87.514622, 0.114605]
49): UALA03 ~ Thu Oct 07 19:02:03 UTC 2021 ~ [33.290483, -87.514735, 0.114635]
50): UALA03 ~ Thu Oct 07 19:02:23 UTC 2021 ~ [33.290648, -87.514856, 0.114635]
51): UALA03 ~ Thu Oct 07 19:02:43 UTC 2021 ~ [33.290805, -87.514969, 0.114513]
52): UALA03 ~ Thu Oct 07 19:03:03 UTC 2021 ~ [33.29096, -
87.515081, 0.114574]
53): UALA03 ~ Thu Oct 07 19:03:23 UTC 2021 ~ [33.291127, -
87.515201, 0.114574]
54): UALA03 ~ Thu Oct 07 19:03:43 UTC 2021 ~ [33.291282, -
87.515314, 0.114513]
55): UALA03 ~ Thu Oct 07 19:04:04 UTC 2021 ~ [33.291437, -
87.515426, 0.114574]
56): UALA03 ~ Thu Oct 07 19:04:24 UTC 2021 ~ [33.291603, -
87.515545, 0.114391]
57): UALA03 ~ Thu Oct 07 19:04:44 UTC 2021 ~ [33.291759, -
87.515659, 0.114544]
58): UALA03 ~ Thu Oct 07 19:05:04 UTC 2021 ~ [33.291915, -
87.515772, 0.114513]
59): UALA03 ~ Thu Oct 07 19:05:24 UTC 2021 ~ [33.292082, -
87.515894, 0.114605]
60): UALA03 ~ Thu Oct 07 19:05:44 UTC 2021 ~ [33.292224, -
87.516008, 0.114513]
61): UALA03 ~ Thu Oct 07 19:06:04 UTC 2021 ~ [33.292397, -
87.516122, 0.114513]
62): UALA03 ~ Thu Oct 07 19:06:24 UTC 2021 ~ [33.292564, -
87.516243, 0.114452]
63): UALA03 ~ Thu Oct 07 19:06:44 UTC 2021 ~ [33.29268, -
87.516175, 0.114056]
64): UALA03 ~ Thu Oct 07 19:07:04 UTC 2021 ~ [33.292773, -
87.515949, 0.113995]
65): UALA03 ~ Thu Oct 07 19:07:24 UTC 2021 ~ [33.292819, -
87.515766, 0.113294]
66): UALA03 ~ Thu Oct 07 19:07:45 UTC 2021 ~ [33.292643, -
87.515674, 0.114026]
67): UALA03 ~ Thu Oct 07 19:08:05 UTC 2021 ~ [33.292459, -
87.515562, 0.114544]
68): UALA03 ~ Thu Oct 07 19:08:25 UTC 2021 ~ [33.292295, -
87.515458, 0.114696]
69): UALA03 ~ Thu Oct 07 19:08:45 UTC 2021 ~ [33.292133, -
87.515354, 0.114757]
70): UALA03 ~ Thu Oct 07 19:09:05 UTC 2021 ~ [33.29196, -
87.515243, 0.114788]
71): UALA03 ~ Thu Oct 07 19:09:25 UTC 2021 ~ [33.291798, -
87.515137, 0.114696]
72): UALA03 ~ Thu Oct 07 19:09:45 UTC 2021 ~ [33.291637, -
87.515032, 0.114605]
73): UALA03 ~ Thu Oct 07 19:10:05 UTC 2021 ~ [33.291465, -
87.514922, 0.114574]
74): UALA03 ~ Thu Oct 07 19:10:25 UTC 2021 ~ [33.291304, -
87.514817, 0.114605]

84
75): UALA03 ~ Thu Oct 07 19:10:46 UTC 2021 ~ [33.291141, -87.514712, 0.114727]
76): UALA03 ~ Thu Oct 07 19:11:06 UTC 2021 ~ [33.290968, -87.5146, 0.114666]
77): UALA03 ~ Thu Oct 07 19:11:26 UTC 2021 ~ [33.290807, -87.514496, 0.114544]
78): UALA03 ~ Thu Oct 07 19:11:46 UTC 2021 ~ [33.290645, -87.51439, 0.114544]
79): UALA03 ~ Thu Oct 07 19:12:06 UTC 2021 ~ [33.290483, -87.514286, 0.114574]
80): UALA03 ~ Thu Oct 07 19:12:26 UTC 2021 ~ [33.290312, -87.514175, 0.114635]
81): UALA03 ~ Thu Oct 07 19:12:46 UTC 2021 ~ [33.290151, -87.51407, 0.114513]
82): UALA03 ~ Thu Oct 07 19:13:06 UTC 2021 ~ [33.289994, -87.513965, 0.114513]
83): UALA03 ~ Thu Oct 07 19:13:26 UTC 2021 ~ [33.29008, -87.513841, 0.113995]
84): UALA03 ~ Thu Oct 07 19:13:46 UTC 2021 ~ [33.290262, -87.513738, 0.113965]
85): UALA03 ~ Thu Oct 07 19:14:06 UTC 2021 ~ [33.290434, -87.513699, 0.114209]
86): UALA03 ~ Thu Oct 07 19:14:26 UTC 2021 ~ [33.29061, -87.513805, 0.113965]
87): UALA03 ~ Thu Oct 07 19:14:46 UTC 2021 ~ [33.290782, -87.513901, 0.114148]
88): UALA03 ~ Thu Oct 07 19:15:06 UTC 2021 ~ [33.290954, -87.513997, 0.114391]
89): UALA03 ~ Thu Oct 07 19:15:27 UTC 2021 ~ [33.291134, -87.514098, 0.114727]
90): UALA03 ~ Thu Oct 07 19:15:47 UTC 2021 ~ [33.291301, -87.514189, 0.114666]
91): UALA03 ~ Thu Oct 07 19:16:07 UTC 2021 ~ [33.291465, -87.514281, 0.114574]
92): UALA03 ~ Thu Oct 07 19:16:27 UTC 2021 ~ [33.291639, -87.514378, 0.114635]
93): UALA03 ~ Thu Oct 07 19:16:47 UTC 2021 ~ [33.291806, -87.51447, 0.114696]
94): UALA03 ~ Thu Oct 07 19:17:07 UTC 2021 ~ [33.291973, -87.514562, 0.114574]
95): UALA03 ~ Thu Oct 07 19:17:27 UTC 2021 ~ [33.29215, -87.514661, 0.114635]
96): UALA03 ~ Thu Oct 07 19:17:47 UTC 2021 ~ [33.292315, -87.514752, 0.114574]
97): UALA03 ~ Thu Oct 07 19:18:07 UTC 2021 ~ [33.292481, -87.514844, 0.114666]
98): UALA03 ~ Thu Oct 07 19:18:27 UTC 2021 ~ [33.292659, -
87.514942, 0.114635]
99): UALA03 ~ Thu Oct 07 19:18:47 UTC 2021 ~ [33.292822, -
87.515032, 0.114209]
100): UALA03 ~ Thu Oct 07 19:19:08 UTC 2021 ~ [33.292897, -
87.514869, 0.114117]
101): UALA03 ~ Thu Oct 07 19:19:28 UTC 2021 ~ [33.292954, -
87.51463, 0.114483]
102): UALA03 ~ Thu Oct 07 19:19:48 UTC 2021 ~ [33.292871, -
87.5145, 0.114239]
103): UALA03 ~ Thu Oct 07 19:20:08 UTC 2021 ~ [33.292683, -
87.514425, 0.114574]
104): UALA03 ~ Thu Oct 07 19:20:28 UTC 2021 ~ [33.292505, -
87.514336, 0.114544]
105): UALA03 ~ Thu Oct 07 19:20:48 UTC 2021 ~ [33.292324, -
87.514242, 0.114696]
106): UALA03 ~ Thu Oct 07 19:21:08 UTC 2021 ~ [33.292157, -
87.514157, 0.114635]
107): UALA03 ~ Thu Oct 07 19:21:28 UTC 2021 ~ [33.291988, -
87.514071, 0.114727]
108): UALA03 ~ Thu Oct 07 19:21:48 UTC 2021 ~ [33.291808, -
87.513979, 0.114696]
109): UALA03 ~ Thu Oct 07 19:22:08 UTC 2021 ~ [33.29164, -
87.513891, 0.114696]
110): UALA03 ~ Thu Oct 07 19:22:28 UTC 2021 ~ [33.291471, -
87.513804, 0.114635]
111): UALA03 ~ Thu Oct 07 19:22:48 UTC 2021 ~ [33.291293, -
87.513712, 0.114605]
112): UALA03 ~ Thu Oct 07 19:23:08 UTC 2021 ~ [33.291126, -
87.513625, 0.114544]
113): UALA03 ~ Thu Oct 07 19:23:29 UTC 2021 ~ [33.290958, -
87.513537, 0.114544]
114): UALA03 ~ Thu Oct 07 19:23:49 UTC 2021 ~ [33.29078, -
87.513445, 0.114513]
115): UALA03 ~ Thu Oct 07 19:24:09 UTC 2021 ~ [33.290611, -
87.513358, 0.114635]
116): UALA03 ~ Thu Oct 07 19:24:29 UTC 2021 ~ [33.29046, -
87.513311, 0.10607]
117):
118):
119):
120): UALA03 ~ Thu Oct 07 19:25:49 UTC 2021 ~ [33.290408, -
87.514615, 0.100249]

****** parse.cpp ******
// Standard Includes
#include <cmath>
#include <string>

// iOceanServerComms Includes
#include "OceanServerComms.h"
#include "conversion.h"

... 

/****************************************************************************
* Parse the State string (format in Ocean-Server's Remote Helm Manual). Post
* all data points from the string to the MOOSdb.
*
* @param   sMessage    a NMEA string
* @return  a boolean indicating success or failure
*/
bool CMOOSOceanServerComms::ParseState(std::string& sMessage)
{
    if(!DoNMEACheckSum(sMessage))
    {
        MOOSTrace("State CheckSum failed...\n");
        /* Failure */
        return false;
    }

    // Temporary storage locations
    std::string sTmp;
    double dfTmp;

    // Useful State values
    bool bSign;
    int nWaypoint;
    int nFinTYaw;
    int nFinBYaw;
    int nFinLPitch;
    int nFinRPitch;
    int nMotorSpeed;
    double dfLatNow;
    double dfLonNow;
    double dfSpeedNow;
    double dfXLocal;
    double dfYLocal;
    double dfDistToNext;
    double dfParkTime;
    double dfTrueDepth;
std::string sMode;
std::string sError;

// Print the message received
MOOSTrace("<< State: %s\n", sMessage.c_str());

ским***
******
* PARSE THE MESSAGE

************************************************************

// Get the time to use when posting to the MOOSdb
dfTimeNow = MOOSTime();

// Skip the $OSI header
sTmp = MOOSChomp(sMessage, ",");

// Get the fin and motor information
sTmp = MOOSChomp(sMessage, ",");
bSign = false; // Unsigned
nFinTYaw = (int)ASCIIHex2Double(sTmp.substr(0, 2), bSign);
nFinBYaw = (int)ASCIIHex2Double(sTmp.substr(2, 2), bSign);
nFinLPitch = (int)ASCIIHex2Double(sTmp.substr(4, 2), bSign);
nFinRPitch = (int)ASCIIHex2Double(sTmp.substr(6, 2), bSign);

// Post the fin and motor information
m_Comms.Notify("FIN_T_YAW", nFinTYaw, dfTimeNow);
m_Comms.Notify("FIN_B_YAW", nFinBYaw, dfTimeNow);
m_Comms.Notify("FIN_L_PITCH", nFinLPitch, dfTimeNow);
m_Comms.Notify("FIN_R_PITCH", nFinRPitch, dfTimeNow);
m_Comms.Notify("FIN_MOTOR_SPEED", nMotorSpeed, dfTimeNow);

// Get the mode of operation
sTmp = MOOSChomp(sMessage, ",");
if(sTmp == "N") // Normal operating UVC
{
    sMode = "NORMAL";
}
else if(sTmp == "M") // Manual override active
{
    sMode = "MANUAL_OVERRIDE";
}
else if(sTmp == "A")        // Servo mode
    {
        sMode = "SERVO";
    }
else if(sTmp == "S")        // UVC stopped, no mission running
    {
        sMode = "STOPPED";
    }
else if(sTmp == "P")        // Manual park active
    {
        sMode = "MANUAL_PARK";
    }
m_Comms.Notify("FRONTSEAT_MODE", sMode, dfTimeNow);

    // Get the waypoint UVC thinks it's driving towards
    sTmp = MOOSChomp(sMessage, ",");
    nWaypoint = std::stoi(sTmp);
    if(nWaypoint <= 0)
    {
        nWaypoint = 1;
    }
else
    {
        nWaypoint += 1;
    }
    // HACK: UVC appears to be reporting the waypoint that was
    // just passed, rather than the one ahead.
    // This violates the OceanServer documentation.
    m_Comms.Notify("FRONTSEAT_WAYPOINT", nWaypoint, dfTimeNow);

    // Get the Latitude
    sTmp = MOOSChomp(sMessage, ",");
    dfLatNow = std::stod(sTmp);
    m_Comms.Notify("FRONTSEAT_LAT", dfLatNow, dfTimeNow);

    // Get the Longitude
    sTmp = MOOSChomp(sMessage, ",");
    dfLonNow = std::stod(sTmp);
    m_Comms.Notify("FRONTSEAT_LONG", dfLonNow, dfTimeNow);

    // Get the Speed (in m/s)
    sTmp = MOOSChomp(sMessage, ",");
    dfSpeedNow = std::stod(sTmp) * KNOTS2MPS;
    m_Comms.Notify("FRONTSEAT_SPEED", dfSpeedNow, dfTimeNow);

    // Get the Distance to Next Waypoint (in meters)
    sTmp = MOOSChomp(sMessage, ",");
dfDistToNext = std::stod(sTmp) * 1852.;
m_Comms.Notify("FRONTSEAT_DIST_TO_NEXT", dfDistToNext, dfTimeNow);

// Get the Error State
sError = MOOSChomp(sMessage, ",");
m_Comms.Notify("FRONTSEAT_ERROR", sError, dfTimeNow);
if(sError.compare("N") != 0)
{
    // On error, release control to permit SRP execution
    m_Comms.Notify("VEHICLE_UNDERWAY", "false", dfTimeNow);
}

// Get the altitude (in feet)
sTmp = MOOSChomp(sMessage, ",");
dfTmp = std::stod(sTmp) * FEET2METERS;
if(dfPrevAlt == 0.0)
{
    // Initial condition
    if(dfTmp < 75.0)
    {
        // Filter initial bad values (e.g. 999.99 reported)
        dfAltitude = dfTmp;
        dfPrevAlt = dfTmp;
        // Else, default to zero.
    }
    else
    {
        // Normal condition
        if((dfTmp < 75.0) && (dfTmp < dfPrevAlt + 10.0) &&
            (dfTmp > dfPrevAlt - 10.0))
        {
            // Filter non-sensical spikes in values
            dfAltitude = dfTmp;
            dfPrevAlt = dfTmp;
            // Else, default to last known good value.
        }
    }
    m_Comms.Notify("FRONTSEAT_ALTITUDE", dfAltitude, dfTimeNow);
}

// Get the remaining Park Time (in seconds)
sTmp = MOOSChomp(sMessage, ",");
try
{
    dfParkTime = std::stod(sTmp);
}
catch (...)
{
// Erroneous leading 'P' observed in UVC v6.0.2+
dfParkTime = std::stod(sTmp.substr(1, sTmp.size()));
}
m_Comms.Notify("FRONTSEAT_PARK_TIME", dfParkTime, dfTimeNow);

// Get the Magnetic Declination (in degrees)
sTmp = MOOSChomp(sMessage, ",");
dfMagVar = std::stod(sTmp);
m_Comms.Notify("FRONTSEAT_MAGNETIC_VAR", dfMagVar, dfTimeNow);

// Get the Depth from Surface (in feet)
// move the message four info blocks forward to retrieve the depth
sTmp = MOOSChomp(sMessage, ",");
sTmp = MOOSChomp(sMessage, ",");
sTmp = MOOSChomp(sMessage, ",");
sTmp = MOOSChomp(sMessage, ",");
dfTrueDepth = std::stod(sTmp) * FEET2METERS /*+ 15.0*/; // convert to meters
m_Comms.Notify("TRUE_DEPTH", dfTrueDepth, dfTimeNow);

/**************************
**********
* CALCULATIONS FROM STATE DATA
****************************************************************
***********/

// Convert our coordinates to the chosen local plane
if(bUTM)
{
    // Local UTM
    m_Geodesy.LatLong2LocalUTM(dfLatNow, dfLonNow, dfYLocal, dfXLocal);
}
else
{
    // Local Grid
    m_Geodesy.LatLong2LocalGrid(dfLatNow, dfLonNow, dfYLocal, dfXLocal);
}
m_Comms.Notify("FRONTSEAT_X", dfXLocal, dfTimeNow);
m_Comms.Notify("FRONTSEAT_Y", dfYLocal, dfTimeNow);

// Update vehicle's estimated position (mimicking iGPS's posted variables)
SetMOOSVar("X", dfXLocal, dfTimeNow);
SetMOOSVar("Y", dfYLocal, dfTimeNow);

/*******************************************
********************
**********
* POST CRITICAL NAV VARIABLES
****************************************************************
**********/
if(bPostNav)
{
    m_Comms.Notify("NAV_X", dfXLocal, dfTimeNow);
    m_Comms.Notify("NAV_Y", dfYLocal, dfTimeNow);
    m_Comms.Notify("NAV_LAT", dfLatNow, dfTimeNow);
    m_Comms.Notify("NAV_LONG", dfLonNow, dfTimeNow);
    m_Comms.Notify("NAV_SPEED", dfSpeedNow, dfTimeNow);
    m_Comms.Notify("NAV_ALTITUDE", dfAltitude, dfTimeNow);
}

/* Success */
return true;
}

...
// endTime = "2021-10-06T22:05:00Z"; // IMPORTANT: Enter start time of transmission before starting script
// unixEndTime = Instant.parse(endTime).epochSecond; // convert to UNIX
// numDelay = (unixEndTime - startTime) * 1000; // convert to milliseconds

delay(2*60*1000); // delay until start time

// 72 times
36.times {

  file << i << ": ";
  if (node.location[2] > minDepth) {
    file << "UALA03 ~ " << phy.rtc << " ~ " << node.location;
    pbtx uALA03;
  }
  delay(20*1000);
  file << "\n";
  i++;
}

****** s3-log.txt ******
1): UALA03 ~ Thu Jan 27 22:02:45 UTC 2022 ~ [33.291082, -87.517881, 4.93776]
2): UALA03 ~ Thu Jan 27 22:03:05 UTC 2022 ~ [33.291126, -87.517602, 4.96824]
3): UALA03 ~ Thu Jan 27 22:03:25 UTC 2022 ~ [33.291169, -87.517326, 5.12064]
4): UALA03 ~ Thu Jan 27 22:03:45 UTC 2022 ~ [33.291158, -87.517053, 5.05968]
5): UALA03 ~ Thu Jan 27 22:04:05 UTC 2022 ~ [33.291113, -87.516775, 5.0292]
6): UALA03 ~ Thu Jan 27 22:04:25 UTC 2022 ~ [33.291066, -87.516502, 5.0916]
7): UALA03 ~ Thu Jan 27 22:04:45 UTC 2022 ~ [33.290861, -87.516314, 4.96824]
8): UALA03 ~ Thu Jan 27 22:05:06 UTC 2022 ~ [33.29063, -87.516314, 4.96824]
9): UALA03 ~ Thu Jan 27 22:05:26 UTC 2022 ~ [33.290423, -87.516333, 4.99872]
10): UALA03 ~ Thu Jan 27 22:05:46 UTC 2022 ~ [33.290232, -87.516517, 4.99872]
11): UALA03 ~ Thu Jan 27 22:06:06 UTC 2022 ~ [33.290111, -87.516715, 4.96824]
12): UALA03 ~ Thu Jan 27 22:06:26 UTC 2022 ~ [33.290189, -87.516997, 4.93776]
13): UALA03 ~ Thu Jan 27 22:06:46 UTC 2022 ~ [33.290252, -87.517275, 5.12064]
14): UALA03 ~ Thu Jan 27 22:07:06 UTC 2022 ~ [33.290311, -87.517548, 5.05968]
15): UALA03 ~ Thu Jan 27 22:07:26 UTC 2022 ~ [33.290529, -87.517615, 5.0292]
16): UALA03 ~ Thu Jan 27 22:07:46 UTC 2022 ~ [33.290771, -87.517666, 5.0292]
17): UALA03 ~ Thu Jan 27 22:08:06 UTC 2022 ~ [33.290965, -87.517555, 4.8768]
18): UALA03 ~ Thu Jan 27 22:08:26 UTC 2022 ~ [33.291152, -87.517364, 4.99872]
19): UALA03 ~ Thu Jan 27 22:08:46 UTC 2022 ~ [33.291257, -87.517156, 4.99872]
20): UALA03 ~ Thu Jan 27 22:09:06 UTC 2022 ~ [33.291195, -87.516866, 4.96824]
21): UALA03 ~ Thu Jan 27 22:09:27 UTC 2022 ~ [33.291145, -87.516586, 4.99872]
22): UALA03 ~ Thu Jan 27 22:09:47 UTC 2022 ~ [33.291034, -87.516377, 5.0292]
23): UALA03 ~ Thu Jan 27 22:10:07 UTC 2022 ~ [33.2908, -87.516291, 4.90728]
24): UALA03 ~ Thu Jan 27 22:10:27 UTC 2022 ~ [33.29057, -87.516198, 5.05968]
26): UALA03 ~ Thu Jan 27 22:11:07 UTC 2022 ~ [33.290176, -87.516452, 5.0292]
27): UALA03 ~ Thu Jan 27 22:11:27 UTC 2022 ~ [33.289989, -87.516624, 4.99872]
28): UALA03 ~ Thu Jan 27 22:11:47 UTC 2022 ~ [33.290052, -87.516889, 4.96824]
29): UALA03 ~ Thu Jan 27 22:12:07 UTC 2022 ~ [33.29012, -87.517166, 5.05968]
30): UALA03 ~ Thu Jan 27 22:12:27 UTC 2022 ~ [33.290185, -87.517442, 5.12064]
APPENDIX C

FILES FOR iModemAbort

The return.c program is used by the user at the shore to command the AUV to return, while the ModemAbort.cpp file governs the response to this command in the application called iModemAbort.

******** return.c ********

/*************************************************
****** Ask the AUV to return to the park location. ******
****** In terminal window (an example):
****** $ make samples
****** $ ./return <ip_address> <rx_node_address> [port]
****** **********************************************

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include ../unet.h
#ifndef _WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif

#define NBYTES 5

static int error(const char *msg) {
    printf("n*** ERROR: %s
", msg);
    return -1;
}
int main(int argc, char *argv[]) {
    unetsocket_t sock;
    fjage_gw_t gw;
    int address = 0;
    int port = 1100;
    // Preamble is "A", "B", "O", "R", "T"
    uint8_t data[NBYTES] = {65, 66, 79, 82, 84};
    int rv;
    if (argc <= 2) {
        error("Usage : return <ip_address> <rx_node_address> [port] \
        "ip_address: IP address of the transmitter modem. \n"rx_node_address: Node address of the receiver modem. \n"port: port number of transmitter modem. \n"A usage example: \n"return 192.168.1.20 5 1100\n");
        return -1;
    } else {
        address = (int)strtol(argv[2], NULL, 10);
        if (argc > 3) port = (int)strtol(argv[3], NULL, 10);
    }

    #ifndef _WIN32
    // Check valid ip address
    struct hostent *server = gethostbyname(argv[1]);
    if (server == NULL) { error("Enter a valid ip address\n");
        return -1;
    }
    #endif

    // Open a unet socket connection to modem
    printf("Connecting to %s:%d\n",argv[1],port);
    sock = unetsocket_open(argv[1], port);
    if (sock == NULL) return error("Couldn't open unet socket");

    // Transmit data
    printf("Transmitting %d bytes of data to %d\n", NBYTES,
            address);
    rv = unetsocket_send_reliable(sock, data, NBYTES, address,
            DATA);
    if (rv != 0) return error("Error transmitting data");

    // Wait for DatagramDeliveryNtf (or) DatagramFailureNtf message
    const char *list[] = {"org.arl.unet.DatagramDeliveryNtf",
                        "org.arl.unet.DatagramFailureNtf");
    gw = unetsocket_get_gateway(sock);
fjage_msg_t msg = fjage_receive_any(gw, list, 2, 60000);
if (strcmp(fjage_msg_get_clazz(msg),
"org.arl.unet.DatagramDeliveryNtf") == 0) {
    printf(" Datagram delivered successfully at the received node! \n");
}
if (strcmp(fjage_msg_get_clazz(msg),
"org.arl.unet.DatagramFailureNtf") == 0) {
    printf(" Datagram delivery failed! \n");
}

// Close the unet socket
unetsocket_close(sock);

printf(" Transmission Complete\n");

return 0;
}

******* ModemAbort.cpp *******
/**
 * NAME: Connor Webb
 * ORGN: University of Alabama
 * FILE: ModemAbort.cpp
 * DATE:
 */
/**
***********************************************************************************/
#include <iterator>
#include "MBUtils.h"
#include "ACTable.h"
#include "ModemAbort.h"

extern "C" {
    #include "fjage.h"
    #include "unet.h"
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

}

#define NBYTES 5
#define AWK_NBYTES 3
using namespace std;

// Constructor
ModemAbort::ModemAbort()
{
}

// Destructor
ModemAbort::~ModemAbort()
{
}

// Procedure: OnNewMail
bool ModemAbort::OnNewMail(MOOSMSG_LIST &NewMail)
{
    AppCastingMOOSApp::OnNewMail(NewMail);

    MOOSMSG_LIST::iterator p;
    for(p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key    = msg.GetKey();

        #if 0 // Keep these around just for template
        string comm  = msg.GetCommunity();
        double dval  = msg.GetDouble();
        string sval  = msg.GetString();
        string msrc  = msg.GetSource();
        double mtime = msg.GetTime();
        bool   mdbl  = msg.IsDouble();
        bool   mstr  = msg.IsString();
        #endif

        if(key == "FOO")
            cout << "great!";

        else if(key != "APPCAST_REQ") // handled by
            AppCastingMOOSApp
            reportRunWarning("Unhandled Mail: " + key);
    }

    return(true);
}

// Procedure: OnConnectToServer
bool ModemAbort::OnConnectToServer()
{
    registerVariables();
    return(true);
}

// Procedure: Iterate()
// happens AppTick times per second
bool ModemAbort::Iterate()
{
    AppCastingMOOSApp::Iterate();
    // Do your thing here!
    unetsocket_t sock;
    fjage_msg_t ntf;
    uint8_t data[NBYTES];
    int address = 69;
    int port = 1100;
    int endpoint = 17;
    int rv;

    // Open a unet socket connection to modem
    printf("Connecting to %s:%d\n", "192.168.1.127", port);
    sock = unetsocket_open("192.168.1.127", port);

    // Bind to protocol DATA
    rv = unetsocket_bind(sock, DATA);
    // Set a timeout of 10 seconds
    unetsocket_set_timeout(sock, 20000);

    // Receive and display data
    printf("Waiting for a Datagram\n");

    ntf = unetsocket_receive(sock);
    if (fjage_msg_get_clazz(ntf) != NULL) {
        printf("Received a %s : [", fjage_msg_get_clazz(ntf));
        fjage_msg_get_byte_array(ntf, "data", data, NBYTES);
        for (int i = 0; i<NBYTES; i++) {
            printf("%d, ", data[i]);
        }
        printf("\n");

            // Notify the MOOSDB that you want to return to the
            // dock
            Notify("RETURN", "true");
            Notify("DEPLOY", "true");
        }
    }
}
Notify("MOOS_MANUAL_OVERRIDE", "false");
Notify("VEHICLE_UNDERWAY", "true");
Notify("DESIRED_FRONTSEAT_WAYPOINT", 17);
}
else {
    fjage_msg_destroy(ntf);
    printf("Error receiving data\n");
}

// Close the unet socket
unetsocket_close(sock);

AppCastingMOOSApp::PostReport();
return(true);

//@------------------------------------------------------------------------
//@ Procedure: OnStartUp()
//@ happens before connection is open

bool ModemAbort::OnStartUp()
{
    AppCastingMOOSApp::OnStartUp();

    STRING_LIST sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
        reportConfigWarning("No config block found for " +
    GetAppName());

    STRING_LIST::iterator p;
    for(p=sParams.begin(); p!=sParams.end(); p++) {
        string orig = *p;
        string line = *p;
        string param = tolower(biteStringX(line, '='));
        string value = line;

        bool handled = false;
        if(param == "foo") {
            handled = true;
        } else if(param == "bar") {
            handled = true;
        }

        if(!handled)
reportUnhandledConfigWarning(orig);

}  
registerVariables();
return(true);
}

//===============================================
// Procedure: registerVariables

void ModemAbort::registerVariables()
{
    AppCastingMOOSApp::RegisterVariables();
    // Register("FOOBAR", 0);
//Register("RETURN", 0);
}

//===============================================
// Procedure: buildReport()

bool ModemAbort::buildReport()
{
    //m_msgs << "============================================" << endl;
    //m_msgs << "File:                                       " << endl;
    //m_msgs << "============================================" << endl;

    //ACTable actab(4);
//actab << "Alpha | Bravo | Charlie | Delta";
//actab.addHeaderLines();
//actab << "one" << "two" << "three" << "four";
//m_msgs << actab.getFormattedString();
    return(true);
}

APPENDIX D

FILES FOR iModemPower

The setplvl.c program is used by the user at the shore to command the AUV to change the signal power level of the Subnero modem, and the ModemPower.cpp file governs the iModemPower application that responds to the command. The ModemPower.cpp requires the file unet_ext.c, and its associated header file unet_ext.h, to link to the API functions controlling the signal power level. To log the power change during a mission, the plvl_logging.groovy script is used on the modem, and the power-log.txt file is the resulting log file.

****** setplvl.c ******

(/)---------/----------------------------------------------
(/)---------/----------------------------------------------
(/)---------/
(/)---------// Ask the modem (on an AUV) to remotely change its power level.
(/)---------// In terminal window (an example):
(/)---------// $ make samples
(/)---------// $ ./setplvl <ip_address> <rx_node_address> [port] <power_level>
(/)---------/----------------------------------------------
(/)---------/----------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "../unet.h"

#ifndef _WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif

#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "../unet.h"

#ifndef _WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif
```c
#define NBYTES 5

static int error(const char *msg) {
    printf("\n*** ERROR: %s\n", msg);
    return -1;
}

int main(int argc, char *argv[]) {
    unetsocket_t sock;
    fjage_gw_t gw;
    int address = 0;
    int port = 1100;
    // Preamble is "P", "L", "V", "L"
    uint8_t data[NBYTES] = {80, 76, 86, 76, 42};
    int rv;
    if (argc <= 2) {
        error("Usage : setplvl <ip_address> <rx_node_address> [port] <power_level> \n" "ip_address: IP address of the transmitter modem. \n" "rx_node_address: Node address of the receiver modem. \n" "port: port number of transmitter modem. \n" "power_level: Desired power level of modem (give absolute value)"
"A usage example: \n" "setplvl 192.168.1.20 5 1100 10\n");
        return -1;
    } else {
        address = (int)strtol(argv[2], NULL, 10);
        if (argc > 3) port = (int)strtol(argv[3], NULL, 10);
    }
    data[4] = (uint8_t)atoi(argv[4]);

    // Open a unet socket connection to modem
    printf("Connecting to %s:%d\n", argv[1], port);
    sock = unetsocket_open(argv[1], port);
    if (sock == NULL) return error("Couldn't open unet socket");

    // Transmit data
    printf("Transmitting %d bytes of data to %d\n", NBYTES, address);
    rv = unetsocket_send_reliable(sock, data, NBYTES, address, DATA);
    if (rv != 0) return error("Error transmitting data");

    // Wait for DatagramDeliveryNtf (or) DatagramFailureNtf message
```
const char *list[] = {"org.arl.unet.DatagramDeliveryNtf", "org.arl.unet.DatagramFailureNtf"};
gw = unetsocket_get_gateway(sock);
fjage_msg_t msg = fjage_receive_any(gw, list, 2, 60000);
if (strcmp(fjage_msg_get_clazz(msg), "org.arl.unet.DatagramDeliveryNtf") == 0) printf(" Datagram delivered successfully at the received node! \n");
if (strcmp(fjage_msg_get_clazz(msg), "org.arl.unet.DatagramFailureNtf") == 0) printf(" Datagram delivery failed! \n");

// Close the unet socket
unetsocket_close(sock);

printf("Transmission Complete\n");

return 0;
}

******* ModemPower.cpp *******
/*                       */
/*    NAME: Connor Webb   */
/*    ORGN: University of Alabama */
/*    FILE: ModemPower.cpp    */
/*    DATE:                 */
/*                       */
#include <iterator>
#include "MBUtils.h"
#include "ACTable.h"
#include "ModemPower.h"

extern “C” {
    #include “fjage.h"
    #include “unet.h”
    #include “unet_ext.h”
}

#define NBYTES 5

using namespace std;

// Constructor
ModemPower::ModemPower()
{
}

// Destructor
ModemPower::~ModemPower()
{
}

// Procedure: OnNewMail
bool ModemPower::OnNewMail(MOOSMSG_LIST &NewMail)
{
    AppCastingMOOSApp::OnNewMail(NewMail);

    MOOSMSG_LIST::iterator p;
    for(p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key    = msg.GetKey();
#
if 0 // Keep these around just for template
        string comm  = msg.GetCommunity();
        double dval  = msg.GetDouble();
        string sval  = msg.GetString();
        string msrc  = msg.GetSource();
        double mtime = msg.GetTime();
        bool   mdbl  = msg.IsDouble();
        bool   mstr  = msg.IsString();
#
endif

        if(key == “FOO“)
            cout << “great!“;

        else if(key != “APPCAST_REQ“) // handled by
        AppCastingMOOSApp
            reportRunWarning(“Unhandled Mail: “ + key);
    }

    return(true);
}

// Procedure: OnConnectToServer
bool ModemPower::OnConnectToServer()
{
    registerVariables();
    return(true);
}

// Procedure: Iterate()
// happens AppTick times per second

bool ModemPower::Iterate()
{
    AppCastingMOOSApp::Iterate();
    // Do your thing here!

    Unetsocket_t sock;
    fjage_msg_t ntf;
    uint8_t data[NBYTES];
    int port = 1100;
    int rv;
    float powerLevel = -42.0;

    sock = unetsocket_open("192.168.1.127", port);
    rv = unetsocket_bind(sock, DATA);
    unetsocket_set_timeout(sock, 20000);

    ntf = unetsocket_receive(sock);
    if (fjage_msg_get_clazz(ntf) != NULL) {
        printf("Received as %s : [", fjage_msg_get_clazz(ntf));
        fjage_msg_get_byte_array(ntf, "data", data, NBYTES);
        for (int I = 0; I < NBYTES; i++) {
            printf("%d, ", data[I]);
        }
        printf("]
    ");
            powerLevel = -1 * data[4];
            unetsocket_close(sock);
            sock = unetsocket_open("192.168.1.127", port);
            rv = unetsocket_ext_set_powerlevel(sock, 1, powerLevel);
            rv = unetsocket_ext_set_powerlevel(sock, 2, powerLevel);
            rv = unetsocket_ext_set_powerlevel(sock, 3, powerLevel);
        }
    }
rv = unetsocket_ext_set_powerlevel(sock, 4, powerLevel);
rv = unetsocket_ext_set_powerlevelBaseband(sock, powerLevel);
}
else {
    fjage_msg_destroy(ntf);
    printf(“Error receiving data”);
}
unetsocket_close(sock);

AppCastingMOOSApp::PostReport();
return(true);
}

// ---------------------------------------------
// Procedure: OnStartUp()
//            happens before connection is open
bool ModemPower::OnStartUp()
{
    AppCastingMOOSApp::OnStartUp();

    STRING_LIST sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
        reportConfigWarning(“No config block found for “ + GetAppName());

    STRING_LIST::iterator p;
    for(p=sParams.begin(); p!=sParams.end(); p++) {
        string orig  = *p;
        string line  = *p;
        string param = tolower(biteStringX(line, ‘=’));
        string value = line;

        bool handled = false;
        if(param == “foo”) {
            handled = true;
        }
        else if(param == “bar”) {
            handled = true;
        }

        if(!handled)
            reportUnhandledConfigWarning(orig);
    }

    return(true);
}
registerVariables();
    return(true);
}

//-----------------------------------------------------------------------------
// Procedure: registerVariables

void ModemPower::registerVariables()
{
    AppCastingMOOSApp::RegisterVariables();
    // Register("FOOBAR", 0);
}

//-----------------------------------------------------------------------------
// Procedure: buildReport()

bool ModemPower::buildReport()
{
    /*m_msgs << "============================================" << endl;
    m_msgs << "File:                                       " << endl;
    m_msgs << "==============================
          ==============
         " << endl;
    m_msgs << "one" << "two" << "three" << "four";*/
    m_msgs << actab.getFormattedString();
    return(true);
}

******* unet_ext.c *******
#define _DEFAULT_SOURCE
#include <stdlib.h>
#include <errno.h>
#include "pthreadwindows.h"
#include "fjage.h"
#include "unet.h"
#include "unet_ext.h"
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <inttypes.h>

#ifndef _WIN32
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#endif

typedef struct {
    fjage_gw_t gw;
    pthread_t tid;
    pthread_mutex_t rxlock, txlock;
    int local_protocol;
    int remote_address;
    int remote_protocol;
    long timeout;
    fjage_aid_t provider;
    bool quit;
    fjage_msg_t ntf;
} _unetsocket_t;

static fjage_msg_t receive(_unetsocket_t *usock, const char *clazz, const char *id, long timeout) {
    pthread_mutex_lock(&usock->txlock);
    fjage_interrupt(usock->gw);
    int rv = pthread_mutex_trylock(&usock->rxlock);
    while (rv == EBUSY) {
        Sleep(100);
        fjage_interrupt(usock->gw);
        rv = pthread_mutex_trylock(&usock->rxlock);
    }
    fjage_msg_t msg = fjage_receive(usock->gw, clazz, id, timeout);
    pthread_mutex_unlock(&usock->rxlock);
    pthread_mutex_unlock(&usock->txlock);
    return msg;
}

static fjage_msg_t request(_unetsocket_t *usock, const fjage_msg_t request, long timeout) {
    pthread_mutex_lock(&usock->txlock);
    fjage_interrupt(usock->gw);
    int rv = pthread_mutex_trylock(&usock->rxlock);
    while (rv == EBUSY) {
        Sleep(100);
        fjage_interrupt(usock->gw);
        rv = pthread_mutex_trylock(&usock->rxlock);
    }
fjage_msg_t msg = fjage_request(usock->gw, request, timeout);
pthread_mutex_unlock(&usock->rxlock);
pthread_mutex_unlock(&usock->txlock);
return msg;
}

static fjage_aid_t agent_for_service(_unetsocket_t *usock, const char *service) {
    pthread_mutex_lock(&usock->txlock);
    fjage_interrupt(usock->gw);
    int rv = pthread_mutex_trylock(&usock->rxlock);
    while (rv == EBUSY) {
        Sleep(100);
        fjage_interrupt(usock->gw);
        rv = pthread_mutex_trylock(&usock->rxlock);
    }
    fjage_aid_t aid = fjage_agent_for_service(usock->gw, service);
pthread_mutex_unlock(&usock->rxlock);
pthread_mutex_unlock(&usock->txlock);
    return aid;
}

static int agents_for_service(_unetsocket_t *usock, const char *service, fjage_aid_t* agents, int max) {
    pthread_mutex_lock(&usock->txlock);
    fjage_interrupt(usock->gw);
    int rv = pthread_mutex_trylock(&usock->rxlock);
    while (rv == EBUSY) {
        Sleep(100);
        fjage_interrupt(usock->gw);
        rv = pthread_mutex_trylock(&usock->rxlock);
    }
    int as = fjage_agents_for_service(usock->gw, service, agents, max);
pthread_mutex_unlock(&usock->rxlock);
pthread_mutex_unlock(&usock->txlock);
    return as;
}

int unetsocket_ext_set_powerlevel(unetsocket_t sock, int index, float value) {
    if (sock == NULL) return -1;
    _unetsocket_t *usock = sock;
    fjage_msg_t msg;
fjage_aid_t phy;
phy = fjage_agent_for_service(usock->gw,
    "org.arl.unet.Services.PHYSICAL");
msg = fjage_msg_create(parameterreq, FJAGE_REQUEST);
fjage_msg_set_recipient(msg, phy);
if (index == 0) index = -1;
fjage_msg_add_int(msg, "index", index);
fjage_msg_add_string(msg, "param", "powerLevel");
fjage_msg_add_float(msg, "value", value);
msg = request(usock, msg, TIMEOUT);
if (msg != NULL && fjage_msg_get_performative(msg) == FJAGE_INFORM) {
    fjage_msg_destroy(msg);
    fjage_aid_destroy(phy);
    return 0;
}
}

int unetsocket_ext_set_powerlevelBaseband(unetsocket_t sock,
float value) {
    if (sock == NULL) return -1;
    _unetsocket_t *usock = sock;
    fjage_msg_t msg;
    fjage_aid_t phy;
    phy = fjage_agent_for_service(usock->gw,
        "org.arl.unet.Services.BASEBAND");
    msg = fjage_msg_create(parameterreq, FJAGE_REQUEST);
    fjage_msg_set_recipient(msg, phy);
    //if (index == 0) index = -1;
    //fjage_msg_add_int(msg, "index", index);
    fjage_msg_add_string(msg, "param", "signalPowerLevel");
    fjage_msg_add_float(msg, "value", value);
    msg = request(usock, msg, TIMEOUT);
    if (msg != NULL && fjage_msg_get_performative(msg) == FJAGE_INFORM) {
        fjage_msg_destroy(msg);
        fjage_aid_destroy(phy);
        return 0;
    }
    fjage_msg_destroy(msg);
    fjage_aid_destroy(phy);
    return -1;
}
***** unet_ext.h *****
#ifndef _UNETEXT_H_
#define _UNETEXT_H_

#include "fjage.h"
#include "unet.h"

/// Set the transmission power level of frame
///
/// @param sock Unet socket
/// @param index Index of the modulation scheme (1 for CONTROL scheme and 2 for DATA scheme)
/// @param value Transmission power level
int unetsocket_ext_set_powerlevel(unetsocket_t sock, int index, float value);

int unetsocket_ext_set_powerlevelBaseband(unetsocket_t sock, float value);

#endif

***** plvl_logging.groovy *****
import java.time.Instant

filename = 'logs/power-log.txt'; // name of file
File file = new File(filename);
I = 1;

subscribe phy;
//delay(2*60*1000); // delay for two minutes

160.times {
    file << I << ""); ";
    file << phy.rtc << " ~ " << phy.signalPowerLevel;
    delay(15*1000);
    file << "\n";
    i++
}

***** power-log.txt *****
1): Thu May 19 15:03:47 UTC 2022 ~ -42.0
2): Thu May 19 15:04:02 UTC 2022 ~ -42.0
3): Thu May 19 15:04:17 UTC 2022 ~ -42.0
4): Thu May 19 15:04:32 UTC 2022 ~ -42.0
5): Thu May 19 15:04:47 UTC 2022 ~ -42.0
6): Thu May 19 15:05:02 UTC 2022 ~ -42.0
7): Thu May 19 15:05:17 UTC 2022 ~ -42.0
8): Thu May 19 15:05:32 UTC 2022 ~ -42.0
9): Thu May 19 15:05:47 UTC 2022 ~ -42.0
10): Thu May 19 15:06:02 UTC 2022 ~ -42.0
11): Thu May 19 15:06:17 UTC 2022 ~ -42.0
12): Thu May 19 15:06:32 UTC 2022 ~ -42.0
13): Thu May 19 15:06:47 UTC 2022 ~ -42.0
14): Thu May 19 15:07:02 UTC 2022 ~ -42.0
15): Thu May 19 15:07:17 UTC 2022 ~ -42.0
16): Thu May 19 15:07:32 UTC 2022 ~ -42.0
17): Thu May 19 15:07:47 UTC 2022 ~ -42.0
18): Thu May 19 15:08:02 UTC 2022 ~ -42.0
19): Thu May 19 15:08:17 UTC 2022 ~ -42.0
20): Thu May 19 15:08:32 UTC 2022 ~ -42.0
21): Thu May 19 15:08:47 UTC 2022 ~ -42.0
22): Thu May 19 15:09:02 UTC 2022 ~ -42.0
23): Thu May 19 15:09:17 UTC 2022 ~ -42.0
24): Thu May 19 15:09:32 UTC 2022 ~ -42.0
25): Thu May 19 15:09:47 UTC 2022 ~ -42.0
26): Thu May 19 15:10:02 UTC 2022 ~ -42.0
27): Thu May 19 15:10:17 UTC 2022 ~ -42.0
28): Thu May 19 15:10:32 UTC 2022 ~ -42.0
29): Thu May 19 15:10:47 UTC 2022 ~ -42.0
30): Thu May 19 15:11:02 UTC 2022 ~ -42.0
31): Thu May 19 15:11:17 UTC 2022 ~ -42.0
32): Thu May 19 15:11:32 UTC 2022 ~ -42.0
33): Thu May 19 15:11:47 UTC 2022 ~ -42.0
34): Thu May 19 15:12:02 UTC 2022 ~ -42.0
35): Thu May 19 15:12:17 UTC 2022 ~ -42.0
36): Thu May 19 15:12:32 UTC 2022 ~ -42.0
37): Thu May 19 15:12:47 UTC 2022 ~ -42.0
38): Thu May 19 15:13:02 UTC 2022 ~ -42.0
39): Thu May 19 15:13:17 UTC 2022 ~ -42.0
40): Thu May 19 15:13:32 UTC 2022 ~ -42.0
41): Thu May 19 15:13:47 UTC 2022 ~ -42.0
42): Thu May 19 15:14:02 UTC 2022 ~ -42.0
43): Thu May 19 15:14:17 UTC 2022 ~ -42.0
44): Thu May 19 15:14:32 UTC 2022 ~ -42.0
45): Thu May 19 15:14:47 UTC 2022 ~ -42.0
46): Thu May 19 15:15:02 UTC 2022 ~ -42.0
47): Thu May 19 15:15:17 UTC 2022 ~ -42.0
48): Thu May 19 15:15:32 UTC 2022 ~ -42.0
49): Thu May 19 15:15:47 UTC 2022 ~ -42.0
50): Thu May 19 15:16:02 UTC 2022 ~ -42.0
51): Thu May 19 15:16:17 UTC 2022 ~ -42.0
52): Thu May 19 15:16:32 UTC 2022 ~ -42.0
53): Thu May 19 15:16:47 UTC 2022 ~ -42.0
54): Thu May 19 15:17:02 UTC 2022 ~ -42.0

113
102): Thu May 19 15:29:04 UTC 2022 ~ -10.0
103): Thu May 19 15:29:19 UTC 2022 ~ -10.0
104): Thu May 19 15:29:34 UTC 2022 ~ -10.0
105): Thu May 19 15:29:49 UTC 2022 ~ -10.0
106): Thu May 19 15:30:04 UTC 2022 ~ -10.0
107): Thu May 19 15:30:19 UTC 2022 ~ -10.0
108): Thu May 19 15:30:34 UTC 2022 ~ -10.0
109): Thu May 19 15:30:49 UTC 2022 ~ -10.0
110): Thu May 19 15:31:04 UTC 2022 ~ -10.0
111): Thu May 19 15:31:19 UTC 2022 ~ -10.0
112): Thu May 19 15:31:34 UTC 2022 ~ -10.0
113): Thu May 19 15:31:49 UTC 2022 ~ -10.0
114): Thu May 19 15:32:04 UTC 2022 ~ -10.0
115): Thu May 19 15:32:19 UTC 2022 ~ -10.0
116): Thu May 19 15:32:34 UTC 2022 ~ -10.0
117): Thu May 19 15:32:49 UTC 2022 ~ -10.0
118): Thu May 19 15:33:04 UTC 2022 ~ -10.0
119): Thu May 19 15:33:19 UTC 2022 ~ -10.0
120): Thu May 19 15:33:34 UTC 2022 ~ -10.0
121): Thu May 19 15:33:49 UTC 2022 ~ -10.0
122): Thu May 19 15:34:04 UTC 2022 ~ -10.0
123): Thu May 19 15:34:19 UTC 2022 ~ -10.0
124): Thu May 19 15:34:34 UTC 2022 ~ -10.0
125): Thu May 19 15:34:49 UTC 2022 ~ -10.0
126): Thu May 19 15:35:04 UTC 2022 ~ -10.0
127): Thu May 19 15:35:19 UTC 2022 ~ -10.0
128): Thu May 19 15:35:34 UTC 2022 ~ -10.0
129): Thu May 19 15:35:49 UTC 2022 ~ -10.0
130): Thu May 19 15:36:04 UTC 2022 ~ -10.0
131): Thu May 19 15:36:19 UTC 2022 ~ -10.0
132): Thu May 19 15:36:34 UTC 2022 ~ -10.0
133): Thu May 19 15:36:49 UTC 2022 ~ -10.0
134): Thu May 19 15:37:04 UTC 2022 ~ -10.0
135): Thu May 19 15:37:19 UTC 2022 ~ -10.0
136): Thu May 19 15:37:34 UTC 2022 ~ -10.0
137): Thu May 19 15:37:49 UTC 2022 ~ -10.0
138): Thu May 19 15:38:04 UTC 2022 ~ -10.0
139): Thu May 19 15:38:19 UTC 2022 ~ -10.0
140): Thu May 19 15:38:34 UTC 2022 ~ -10.0
141): Thu May 19 15:38:49 UTC 2022 ~ -10.0
142): Thu May 19 15:39:04 UTC 2022 ~ -10.0
143): Thu May 19 15:39:19 UTC 2022 ~ -10.0
144): Thu May 19 15:39:34 UTC 2022 ~ -10.0
145): Thu May 19 15:39:49 UTC 2022 ~ -10.0
146): Thu May 19 15:40:04 UTC 2022 ~ -10.0
147): Thu May 19 15:40:20 UTC 2022 ~ -10.0
148): Thu May 19 15:40:35 UTC 2022 ~ -10.0
149): Thu May 19 15:40:50 UTC 2022 ~ -10.0
150): Thu May 19 15:41:05 UTC 2022 ~ -10.0
151): Thu May 19 15:41:20 UTC 2022 ~ -10.0
152): Thu May 19 15:41:35 UTC 2022 ~ -10.0
153): Thu May 19 15:41:50 UTC 2022 ~ -10.0
154): Thu May 19 15:42:05 UTC 2022 ~ -10.0
155): Thu May 19 15:42:20 UTC 2022 ~ -10.0
156): Thu May 19 15:42:35 UTC 2022 ~ -10.0
157): Thu May 19 15:42:50 UTC 2022 ~ -10.0
158): Thu May 19 15:43:05 UTC 2022 ~ -10.0
159): Thu May 19 15:43:20 UTC 2022 ~ -10.0
160): Thu May 19 15:43:35 UTC 2022 ~ -10.0
APPENDIX E

FILES FOR iModemDeploy

The setmission.c file is used to command the AUV to change the current backseat mission. The ModemDeploy.cpp file governs the behavior of the iModemDeploy application. The behavior file, current.bhv, defined the three different backseat mission behaviors.

****** setmission.c ******
////////////////////////////////////////////////////////////////
///////////////
//
// Ask the AUV to change between different missions.
//
// In terminal window (an example):
//
// $ make samples
// $ ./setmission <ip_address> <rx_node_address> [port]
// <mission_number>
//
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <../unet.h>
#if _WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif
#define NBYTES 5
static int error(const char *msg) {
    printf("\n*** ERROR: %s\n", msg);
    return -1;
}
int main(int argc, char *argv[]) {
    unetsocket_t sock;
    fjage_gw_t gw;
    int address = 0;
    int port = 1100;
    // Preamble is "D", "P", "L", "Y"
    uint8_t data[NBYTES] = {68, 80, 76, 89, 1};
    int rv;
    if (argc <= 2) {
        error("Usage : setmission <ip_address> <rx_node_address> [port] <mission_number> \n"
              "ip_address: IP address of the transmitter modem. \n"
              "rx_node_address: Node address of the receiver modem. \n"
              "port: port number of transmitter modem. \n"
              "mission_number: Number of mission to run"
              "A usage example: \n"
              "setmission 192.168.1.20 5 1100 3\n");
        return -1;
    }
    else {
        address = (int)strtol(argv[2], NULL, 10);
        if (argc > 3) port = (int)strtol(argv[3], NULL, 10);
    }
    data[4] = (uint8_t)atoi(argv[4]);
    // Open a unet socket connection to modem
    printf("Connecting to %s:%d \n",argv[1],port);
    sock = unetsocket_open(argv[1], port);
    if (sock == NULL) return error("Couldn’t open unet socket");
    // Transmit data
    printf("Transmitting %d bytes of data to %d\n", NBYTES, address);
    rv = unetsocket_send_reliable(sock, data, NBYTES, address, DATA);
    if (rv != 0) return error("Error transmitting data");
    // Wait for DatagramDeliveryNtf (or) DatagramFailureNtf message
    const char *list[] = {"org.arl.unet.DatagramDeliveryNtf",
                          "org.arl.unet.DatagramFailureNtf"};
    gw = unetsocket_get_gateway(sock);
    fjage_msg_t msg = fjage_receive_any(gw, list, 2, 60000);
}


if (strcmp(fjage_msg_get_clazz(msg),  
"org.arl.unet.DatagramDeliveryNtf") == 0) printf("Datagram delivered successfully at the received node! \n");
if (strcmp(fjage_msg_get_clazz(msg),  
"org.arl.unet.DatagramFailureNtf") == 0) printf("Datagram delivery failed! \n");
// Close the unet socket
unetsocket_close(sock);

printf("Transmission Complete\n");

return 0;
}

****** ModemDeploy.cpp ******

ICAST::ModemDeploy::ModemDeploy()
{
}

#include <iterator>
#include "MBUtils.h"
#include "ACTable.h"
#include "ModemDeploy.h"

extern "C" {
    #include "fjage.h"
    #include "unet.h"
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
}
#define NBYTES 5
#define AWK_NBYTES 3

using namespace std;

// Constructor
ModemDeploy::ModemDeploy()
{
}
// Destructor

ModemDeploy::~ModemDeploy()
{
}

// Procedure: OnNewMail

bool ModemDeploy::OnNewMail(MOOSMSG_LIST &NewMail)
{
    AppCastingMOOSApp::OnNewMail(NewMail);

    MOOSMSG_LIST::iterator p;
    for(p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key    = msg.GetKey();
        #if 0 // Keep these around just for template
        string comm  = msg.GetCommunity();
        double dval  = msg.GetDouble();
        string sval  = msg.GetString();
        string msrc  = msg.GetSource();
        double mtime = msg.GetTime();
        bool   mdbl  = msg.IsDouble();
        bool   mstr  = msg.IsString();
        #endif

        if(key == “FOO”)
            cout << “great!”;

        else if(key != “APPCAST_REQ”) // handled by AppCastingMOOSApp
            reportRunWarning(“Unhandled Mail: “ + key);
    }

    return(true);
}

// Procedure: OnConnectToServer

bool ModemDeploy::OnConnectToServer()
{
    registerVariables();
    return(true);
}
Procedure: Iterate()

bool ModemDeploy::Iterate()
{
    AppCastingMOOSApp::Iterate();
    // Do your thing here!
    Unetsocket_t sock;
    fjage_msg_t ntf;
    uint8_t data[NBYTES];
    int address = 69;
    int port = 1100;
    int endpoint = 17;
    int rv;

    // Open a unet socket connection to modem
    printf("Connecting to %s:%d\n", "192.168.1.127", port);
    sock = unetsocket_open("192.168.1.127", port);

    // Bind to protocol DATA
    rv = unetsocket_bind(sock, DATA);

    // Set a timeout of 10 seconds
    unetsocket_set_timeout(sock, 20000);

    // Receive and display data
    printf("Waiting for a Datagram\n");
    ntf = unetsocket_receive(sock);
    if (fjage_msg_get_clazz(ntf) != NULL) {
        printf("Received a %s : ", fjage_msg_get_clazz(ntf));
        fjage_msg_get_byte_array(ntf, "data", data, NBYTES);
        for (int i = 0; i < NBYTES; i++) {
            printf("%d,", data[i]);
        }
        printf("\n");
    }
    printf("\n");
        data[3] == 89) {
        if (data[4] == 1) {
            Notify("MISSION", 1);
            Notify("DEPLOY", "true");
            Notify("RETURN", "false");
        }
Notify("MOOS_MANUAL_OVERRIDE", "false");
Notify("VEHICLE_UNDERWAY", "true");
}
else if (data[4] == 2) {
    Notify("MISSION", 2);
    Notify("DEPLOY", "true");
    Notify("RETURN", "false");
    Notify("MOOS_MANUAL_OVERRIDE", "false");
    Notify("VEHICLE_UNDERWAY", "true");
}
else if (data[4] == 3) {
    Notify("MISSION", 3);
    Notify("DEPLOY", "true");
    Notify("RETURN", "false");
    Notify("MOOS_MANUAL_OVERRIDE", "false");
    Notify("VEHICLE_UNDERWAY", "true");
}
else {
    fjage_msg_destroy(ntf);
    printf("Error receiving data\n");
}

    // Close the unet socket
    unetsocket_close(sock);
    AppCastingMOOSApp::PostReport();
    return(true);

    // Procedure: OnStartUp()
    //            happens before connection is open

    bool ModemDeploy::OnStartUp()
    {
        AppCastingMOOSApp::OnStartUp();

        STRING_LIST sParams;
        m_MissionReader.EnableVerbatimQuoting(false);
        if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
            reportConfigWarning("No config block found for " +
                                GetAppName());

        STRING_LIST::iterator p;
        for(p=sParams.begin(); p!=sParams.end(); p++) {
            string orig  = *p;
string line = *p;
string param = tolower(biteStringX(line, '='));
string value = line;

bool handled = false;
if(param == "foo") {
    handled = true;
} else if(param == "bar") {
    handled = true;
}
if(!handled)
    reportUn handledConfigWarning(orig);

registerVariables();
return(true);

//------------------------------------------------------------------------------------------------------------------------
// Procedure: registerVariables
void ModemDeploy::registerVariables()
{
    AppCastingMOOSApp::RegisterVariables();
    // Register("FOOBAR", 0);
}

//------------------------------------------------------------------------------------------------------------------------
// Procedure: buildReport()

bool ModemDeploy::buildReport()
{
    m_msgs << "============================================" << endl;
    m_msgs << "File: " << endl;
    m_msgs << "============================================" << endl;
    ACTable actab(4);
    actab << "Alpha | Bravo | Charlie | Delta";
    actab.addHeaderLines();
    actab << "one" << "two" << "three" << "four";
    m_msgs << actab.getFormattedString();
return(true);
}

****** current.bhv ******
//----------------------------------------------------------------------
---------
// Behavior (BHV) File for IVER Operations (generated)
initialize DEPLOY = false
initialize RETURN = false

//----------------------------------------------------------------------
---------
Behavior = BHV_Waypoint
{
    name = waypt_survey1
    pwt = 100
    condition = RETURN = false
    condition = DEPLOY = true
    condition = MISSION = 1
    endflag = RETURN = false
    endflag = DEPLOY = false
    perpetual = true
    updates = NEWPTS

    lead = 8
    lead_damper = 1
    speed = 2.0 // meters per second
    radius = 8.0
    //
    points = zigzag:35,45,90,1000,200,50
    points = 333,0:333,30
    repeat = 0
    visual_hints = nextpt_color=red, nextpt_lcolor=green
    visual_hints = vertex_color=yellow, line_color=white
    visual_hints = vertex_size=2, edge_size=1
}
//----------------------------------------------------------------------
---------
Behavior = BHV_Waypoint
{
    name = waypt_survey2
    pwt = 100
    condition = RETURN = false
    condition = DEPLOY = true
    condition = MISSION = 2
    endflag = MISSION = 2
    endflag = RETURN = false
endflag = DEPLOY = false
perpetual = true
updates = NEWPTS

lead = 8
lead_damper = 1
speed = 2.0 // meters per second
radius = 8.0
// points = zigzag:35,45,90,1000,200,50
// points = 60,-40:60,-160:150,-160:180,-100:150,-40
points = 333,0:333,-30
repeat = 0
visual_hints = nextpt_color=red, nextpt_lcolor=green
visual_hints = vertex_color=yellow, line_color=white
visual_hints = vertex_size=2, edge_size=1

Behavior = BHV_Waypoint
{
  name = waypt_survey3
  pwt = 100
  condition = RETURN = false
  condition = DEPLOY = true
  condition = MISSION = 3
  endflag = RETURN = false
  endflag = DEPLOY = false
  perpetual = true
  updates = NEWPTS

    lead = 8
    lead_damper = 1
    speed = 2.0 // meters per second
    radius = 8.0
    // points = zigzag:35,45,90,1000,200,50
    points = 333,0:333,-30
    repeat = 0
    visual_hints = nextpt_color=red, nextpt_lcolor=green
    visual_hints = vertex_color=yellow, line_color=white
    visual_hints = vertex_size=2, edge_size=1
}

Behavior = BHV_Waypoint
{  
    name       = waypt_return  
    pwt        = 100  
    condition  = RETURN = true  
    condition  = DEPLOY = true  
    updates    = UPDATES_RETURN  
    perpetual = true  
    endflag    = RETURN = false  
    endflag    = DEPLOY = false  
    speed      = 0.5  
    radius     = 2.0  
    nm_radius  = 8.0  
    points     = 0,0  
}

Behavior = BHV_ConstantDepth
{
    // General Behavior Parameters
    // ---------------------------
    name         = const_dep_survey   // example
    pwt          = 100                // default
    condition    = DEPLOY==true    // example
    updates      = CONST_DEP_UPDATES  // example

    // Parameters specific to this behavior
    // ------------------------------------
    basewidth = 100          // default
    depth = 0            // default
    depth_mismatch_var = DEPTH_DIFF   // example
    duration = no-time-limit            // default
    peakwidth = 3            // default
    summitdelta = 50           // default
}

Behavior = BHV_ConstantDepth
{
    // General Behavior Parameters
    // ---------------------------
    name         = const_dep_return  // example
    pwt          = 100                // default
    condition    = RETURN==true    // example
    updates      = CONST_DEP_UPDATES  // example

    // Parameters specific to this behavior
    // ------------------------------------
    basewidth = 100          // default
    depth = 0            // default
depth_mismatch_var = DEPTH_DIFF // example
  duration = no-time-limit // default
  peakwidth = 3 // default
  summitdelta = 50 // default
}
APPENDIX F

FILES FOR iModemInfo

The depth.c file sends an inquiry to the AUV for the depth of the vehicle. The
iModemInfo application responds, which is defined by the ModemInfo.cpp file.

******* depth.c *******
////////////////////////////////////////////////////////////////////
///////////////
//
// Ask a node to return its depth data.
//
// In terminal window (an example):
//
// $ make samples
// $ ./depth <ip_address> <rx_node_address> [port]
//
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include ../unet.h
#ifndef _WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif
#define NBYTES 5
#define RXNBYTES 5

static int error(const char *msg) {
    printf("\n*** ERROR: %s\n", msg);
    return -1;
}

// Union to translate between datagram bytes and float
union union_t {
double d;
unsigned char array[sizeof(double)];
} double_u;

int main(int argc, char *argv[]) {

  unetsocket_t sock;
  fjage_gw_t gw;
  fjage_msg_t ntf;
  int address = 0;
  int port = 1100;
  // Preamble is "I", "N", "F", "O", "D"
  uint8_t data[NBYTES] = {73, 78, 70, 79, 68};
  uint8_t rxdata[sizeof(double)];
  int rv;
  if (argc <= 2) {
    error("Usage : depth <ip_address> <rx_node_address> [port]
    "ip_address: IP address of the transmitter modem. \n"rx_node_address: Node address of the receiver modem. \n"port: port number of transmitter modem. \n"A usage example: \n"depth 192.168.1.20 5 1100\n"); return -1;
  } else {
    address = (int)strtol(argv[2], NULL, 10);
    if (argc > 3) port = (int)strtol(argv[3], NULL, 10);
  }

#ifndef _WIN32
  // Check valid ip address
  struct hostent *server = gethostbyname(argv[1]);
  if (server == NULL) {
    error("Enter a valid ip address\n"); return -1;
  }
#endif

  // Open a unet socket connection to modem
  printf("Connecting to %s:%d\n", argv[1], port);
  sock = unetsocket_open(argv[1], port);
  if (sock == NULL) return error("Couldn't open unet socket");

  // Transmit data
  printf("Transmitting %d bytes of data to %d\n", NBYTES, address);

rv = unetsocket_send_reliable(sock, data, NBYTES, address, DATA);
if (rv != 0) return error("Error transmitting data");

// Wait for DatagramDeliveryNtf (or) DatagramFailureNtf message
const char *list[] = {"org.arl.unet.DatagramDeliveryNtf",
"org.arl.unet.DatagramFailureNtf"};
gw = unetsocket_get_gateway(sock);
fjage_msg_t msg = fjage_receive_any(gw, list, 2, 60000);
if (strcmp(fjage_msg_get_clazz(msg),
"org.arl.unet.DatagramDeliveryNtf") == 0) {
    printf("Datagram delivered successfully at the received node!
\n");
}
if (strcmp(fjage_msg_get_clazz(msg),
"org.arl.unet.DatagramFailureNtf") == 0) {
    printf("Datagram delivery failed!
\n");
}

// Close the unet socket
unetsocket_close(sock);

printf("Transmission Complete\n");

// Wait to receive the depth data back
#ifndef _WIN32
    // Check valid ip address
    server = gethostbyname(argv[4]);
    if (server == NULL) {
        error("Enter a valid ip address\n");
        return -1;
    }
#endif

// Open a unet socket connection to modem
printf("Connecting to %s:%d\n", argv[4], port);
sock = unetsocket_open(argv[4], port);
if (sock == NULL) return error("Couldn't open unet socket");

// Bind to protocol DATA
rv = unetsocket_bind(sock, DATA);
if (rv != 0) return error("Error binding socket");

// Set a timeout of 10 seconds
unetsocket_set_timeout(sock, 20000);
// Receive and display data
printf("Waiting for a Datagram\n");

ntf = unetsocket_receive(sock);
if (fjage_msg_get_clazz(ntf) != NULL) {
    printf("Received a %s : [", fjage_msg_get_clazz(ntf));
    fjage_msg_get_byte_array(ntf, "data", rxdata,
    sizeof(double));
    // Print data to screen
    for (long unsigned int i = 0; i<sizeof(rxdata); i++) {
        printf("%d,", rxdata[i]);
        double_u.array[i] = rxdata[i];
    }
    printf("]\n");
    double d = double_u.d;
    printf("Depth: %lf\n", d);
} else {
    fjage_msg_destroy(ntf);
    return error("Error receiving data");
}

// Close the unet socket
unetsocket_close(sock);
printf("Reception Complete\n");
return 0;
return 0;

}
union doubleunion_t {
    double d;
    unsigned char array[sizeof (double)];
} double_u;

#define NBYTES 5

using namespace std;

// Constructor
ModemInfo::ModemInfo()
{
    current_depth = 0.0;
}

// Destructor
ModemInfo::~ModemInfo()
{
}

// Procedure: OnNewMail
bool ModemInfo::OnNewMail(MOOSMSG_LIST &NewMail)
{
    AppCastingMOOSApp::OnNewMail(NewMail);

    MOOSMSG_LIST::iterator p;
    for(p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key    = msg.GetKey();
#if 0 // Keep these around just for template
        string comm  = msg.GetCommunity();
        double dval  = msg.GetDouble();
        string sval  = msg.GetString();
        string msrc  = msg.GetSource();
        double mtime = msg.GetTime();
        bool   mdbl  = msg.IsDouble();
#endif
bool mstr = msg.IsString();
#endif

if(key == "TRUE_DEPTH") {
    current_depth = msg.GetDouble();
}

else if(key != "APPCAST_REQ") // handled by AppCastingMOOSApp
    reportRunWarning("Unhandled Mail: " + key);

return(true);
}

// Procedure: OnConnectToServer

bool ModemInfo::OnConnectToServer()
{
    registerVariables();
    return(true);
}

// Procedure: Iterate()
//            happens AppTick times per second

bool ModemInfo::Iterate()
{
    AppCastingMOOSApp::Iterate();
    // Do your thing here!

    unetsocket_t sock;
    fjage_msg_t ntf;
    uint8_t data[NBYTES];
    int port = 1100;
    int rv;

    // Open a unet socket connection to modem
    printf("Connecting to %s:%d\n", "192.168.1.127", port);
    sock = unetsocket_open("192.168.1.127", port);

    // Bind to protocol DATA
    rv = unetsocket_bind(sock, DATA);

    // Set a timeout of 10 seconds
unetsocket_set_timeout(sock, 20000);

// Receive and display data
printf("Waiting for a Datagram\n");

ntf = unetsocket_receive(sock);
if (fjage_msg_get_clazz(ntf) != NULL) {
    printf("Received a %s : [", fjage_msg_get_clazz(ntf));
    fjage_msg_get_byte_array(ntf, "data", data, NBYTES);
    for (int i = 0; i<NBYTES; i++) {
        printf("%d," , data[i]);
    }
    printf("]\n");
        unetsocket_close(sock);
    }
}

unetsocket_t sock;
int address = 20;
int port = 1100;
uint8_t data[sizeof(double)];
int rv;

// current_depth = 5.0;
double_u.d = current_depth;

for (int i = 0; i < sizeof(double); i++) {
    data[i] = double_u.array[i];
}

// Open a unet socket connection to modem
printf("Connecting to 192.168.1.127:%d\n", port);
sock = unetsocket_open("192.168.1.127", port);
if (sock == NULL) printf("Couldn't open unet socket");

// Transmit data
printf("Transmitting %lu bytes of data to %d\n", sizeof(double), address);
printf("Sending %lf as: [", current_depth);
for (int i = 0; i<sizeof(double); i++) {
    printf("%d," , data[i]);
}
printf("]\n");
rv = unetsocket_send(sock, data, sizeof(double),
        address, DATA);
if (rv != 0) printf("Error transmitting data");

printf("Transmission Complete\n");
}
}
else {
    fjage_msg_destroy(ntf);
    printf("Error receiving data\n");
}

// Close the unet socket
unetsocket_close(sock);

AppCastingMOOSApp::PostReport();
return(true);

//---------------------------------------------------------
// Procedure: OnStartUp()
//            happens before connection is open

bool ModemInfo::OnStartUp()
{
    AppCastingMOOSApp::OnStartUp();

    STRING_LIST sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
        reportConfigWarning("No config block found for " + GetAppName());

    STRING_LIST::iterator p;
    for(p=sParams.begin(); p!=sParams.end(); p++) {
        string orig  = *p;
        string line  = *p;
        string param = tolower(biteStringX(line, '='));
        string value = line;

        bool handled = false;
        if(param == "foo") {
            handled = true;
        } else if(param == "bar") {
            handled = true;
        }
if(!handled)
    reportUnhandledConfigWarning(orig);
}

registerVariables();
return(true);

// Procedure: registerVariables
void ModemInfo::registerVariables()
{
    AppCastingMOOSApp::RegisterVariables();
    m_Comms.Register("TRUE_DEPTH", 0);
    // Register("FOOBAR", 0);
}

// Procedure: buildReport()
bool ModemInfo::buildReport()
{
    return(true);
}
APPENDIX G

FILES FOR iModemForward

The broadcastList.c file initiates the reception and forwarding of location information throughout the network of AUVs. The iModemForward application is defined by the ModemForward.cpp file. Two test were performed of this application. The laboratory test results are found in labtest5.txt, while the field results are found in output613.txt.

******** broadcastList.c *******
/*----------------------------------------*/

// Broadcast the ordered list of nodes and an empty list to be filled by the
// network. Then, wait to receive the list and the completed list back, which
// is then saved to an output .txt file.
//
// In terminal window (an example):
//
// $ make samples
// $ ./broadcastList <ip_address> <rx_node_address> [port]

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <../unet.h>
#ifndef __WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif
#define NBYTES 5

#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include ".\unet.h"

#ifndef _WIN32
#include <unistd.h>
#include <netdb.h>
#include <sys/time.h>
#endif

#define NBYTES 5
#define RXNBYTES 5

static int error(const char *msg) {
    printf("\n*** ERROR: %s\n\n", msg);
    return -1;
}

// Set up unions to translate data from datagram bytes to floats
union union_lat {
    double d;
    unsigned char array[sizeof(double)];
} double_lat;

union union_lon {
    double d;
    unsigned char array[sizeof(double)];
} double_lon;

union union_depth {
    double d;
    unsigned char array[sizeof(double)];
} double_depth;

// temperature data has some issues
/*union union_temp {
    double d;
    unsigned char array[sizeof(double)];
} double_temp;*/

int main(int argc, char *argv[]) {

    unetsocket_t sock;
    fjage_gw_t gw;
    fjage_msg_t ntf;
    int address = 0;
    int port = 1100;
    uint8_t txdata[150];
    uint8_t rxdata[40];
    uint8_t list[40];
    uint8_t rxlist[150];
    int rv, rv2;
    if (argc <= 2) {
        error("Usage : broadcastList <ip_address> <rx_node_address>
        [port] <list> <list_lenth> <shore_ip_address> <shore_port>\n"
        "ip_address: IP address of the transmitter modem. \n"
        "rx_node_address: Node address of the receiver modem. \n"
"port: port number of transmitter modem. \n"
"list: ordered list of nodes in the network. \n"
"list_length: length of list (i.e. number of nodes in network). \n"
"shore_ip_address: IP address of shore modem. \n"
"shore_port: port number of shore modem. \n"
"A usage example: \n"
broadcastList 192.168.1.20 5 1100 97 69 20 5 4
192.168.1.20 1100\n
return -1;
}
else {
    address = (int)strtol(argv[2], NULL, 10);
    if (argc > 3) port = (int)strtol(argv[3], NULL, 10);
}
#endif

FILE *fw = fopen(argv[7], "w");
for (int a = 0; a < atoi(argv[6]); a++) {

    // Open a unet socket connection to modem
    printf("Connecting to %s:%d\n", argv[1], port);
    sock = unetsocket_open(argv[1], port);
    if (sock == NULL) return error("Couldn't open unet socket");

    // Transmit ordered list
    int length = atoi(argv[4]);
    printf("Transmitting list to %d\n", address);
    // Preamble is "L", "I", "S", "T"
    list[0] = 76;
    list[1] = 73;
    list[2] = 83;
    list[3] = 84;

    FILE *fp = fopen("list.txt", "r");
    for (int i = 4; i < length+4; i++) {
        fscanf(fp, "%hhu", &list[i]);
    }
    for (int i = length+4; i < 40; i++){
list[i] = 0;
}
rv = unetsocket_send_reliable(sock, list, 40, address, DATA);
if (rv != 0) return error("Error transmitting data");

// Wait for DatagramDeliveryNtf (or) DatagramFailureNtf message
const char *list2[] = {"org.arl.unet.DatagramDeliveryNtf", "org.arl.unet.DatagramFailureNtf"};
gw = unetsocket_get_gateway(sock);
fjage_msg_t msg = fjage_receive_any(gw, list2, 2, 60000);
if (strcmp(fjage_msg_get_clazz(msg), "org.arl.unet.DatagramDeliveryNtf") == 0) {
    printf("Datagram delivered successfully at the received node! \n");
}
if (strcmp(fjage_msg_get_clazz(msg), "org.arl.unet.DatagramFailureNtf") == 0) {
    printf("Datagram delivery failed! \n");
}
unetsocket_close(sock);

// Transmit empty list
printf("Connecting to %s:%d\n", argv[1], port);
sock = unetsocket_open(argv[1], port);
if (sock == NULL) return error("Couldn't open unet socket");

printf("Transmitting empty list to %d\n", address);
for (int i = 0; i < 150; ++i) {
    txdata[i] = 0;
}
rv2 = unetsocket_send_reliable(sock, txdata, 150, address, DATA);
if (rv2 != 0) return error("Error transmitting data");

// Wait for DatagramDeliveryNtf (or) DatagramFailureNtf message
gw = unetsocket_get_gateway(sock);
fjage_msg_t msg2 = fjage_receive_any(gw, list2, 2, 60000);
if (strcmp(fjage_msg_get_clazz(msg2), "org.arl.unet.DatagramDeliveryNtf") == 0) {
    printf("Datagram delivered successfully at the received node! \n");
}
if (strcmp(fjage_msg_get_clazz(msg2), "org.arl.unet.DatagramFailureNtf") == 0) {

printf("Datagram delivery failed! \n");
}
// Close the unet socket
unetsocket_close(sock);
printf("Transmission Complete\n");

// Wait for Completed Ordered list to return
#ifndef _WIN32
// Check valid ip address
server = gethostbyname(argv[1]);
if (server == NULL) {
    error("Enter a valid ip address\n");
    return -1;
}
#endif

// Open a unet socket connection to modem
printf("Connecting to %s:%d\n", argv[1], atoi(argv[5]));
sock = unetsocket_open(argv[1], atoi(argv[5]));
if (sock == NULL) return error("Couldn't open unet socket");

// Bind to protocol DATA
rv = unetsocket_bind(sock, DATA);
if (rv != 0) return error("Error binding socket");

// Set a timeout of 10 minutes
unetsocket_set_timeout(sock, 600000);

// Receive and display data
printf("Waiting for a Datagram\n");
ntf = unetsocket_receive(sock);
if (fjage_msg_get_clazz(ntf) != NULL) {
    printf("Received a %s\n", fjage_msg_get_clazz(ntf));
    fjage_msg_get_byte_array(ntf, "data", rxdata, 40);
} else {
    fjage_msg_destroy(ntf);
    return error("Error receiving data");
}

unetsocket_close(sock);
printf("Connecting to %s:%d\n", argv[1], atoi(argv[5]));
sock = unetsocket_open(argv[1], atoi(argv[5]));
if (sock == NULL) return error("Couldn't open unet socket");

printf("Waiting for a Datagram\n");
ntf = unetsocket_receive(sock);
if (fjage_msg_get_clazz(ntf) != NULL) {
    printf("Received a %s\n", fjage_msg_get_clazz(ntf));
    fjage_msg_get_byte_array(ntf, "data", rxlist, 150);
    for (int j = 0; j < atoi(argv[4]) - 1; ++j) {
        fprintf(fw, "%d,", rxlist[5+(30*j)]);
        for (int i = 6; i < (6 + (int)sizeof(double)); ++i) {
            double_lat.array[i-6] = rxlist[(30*j)+i];
        }
        fprintf(fw, "%lf," , double_lat.d);
        for (int i = 14; i < (14 + (int)sizeof(double)); ++i) {
            double_lon.array[i-14] = rxlist[(30*j)+i];
        }
        fprintf(fw, "%lf", double_lon.d);
        for (int i = 22; i < (22 + (int)sizeof(double)); ++i) {
            double_depth.array[i-22] = rxlist[(30*j)+i];
        }
        fprintf(fw, "%lf\n", double_depth.d);
    }
}
else {
    fjage_msg_destroy(ntf);
    return error("Error receiving data");
}

// Close the unet socket
unetsocket_close(sock);
printf("Reception Complete\n");
printf("Broadcast Successful\n");
}
fclose(fw);

// Print to the Screen
FILE *f = fopen(argv[7], "r");
printf("Node Latitude Longitude Depth\n");
char buffer[100];
while (fgets(buffer, 100, f)) {
    char *token = strtok(buffer, ",");
    int i = 0;
    while (token) {

// Just printing each integer here but handle as needed
double n = atof(token);
printf("%lf ", n);

token = strtok(NULL, ",");
i++;
if ((i % 4) == 0) {
    printf("\n");
}
}
fclose(f);
return 0;

return 0;

****** ModemForward.cpp ******
/********************************************
**************
#include <iostream>
#include "MBUtils.h"
#include "ACTable.h"
#include "ModemForward.h"

text "C" {
    #include "fjage.h"
    #include "unet.h"
    #include <stdio.h>
    #include <stdlib.h>
}

union doubleunion_navLat {
    double d;
    unsigned char array[sizeof (double)];
} navLat_u;

union doubleunion_navLong {
    double d;
    unsigned char array[sizeof (double)];
} navLong_u;
union doubleunion_navDepth {
    double d;
    unsigned char array[sizeof (double)];
} navDepth_u;

union doubleunion_ysiTemp {
    double d;
    unsigned char array[sizeof (double)];
} ysiTemp_u; /*
#define NBYTES 40
#define DATA_NBYTES 150

using namespace std;

//----------------------------------------------------------
// Constructor

ModemForward::ModemForward()
{
    navLat = 0.0;
    navLong = 0.0;
    navDepth = 0.0;
    //ysiTemp = 0.0;
}

//----------------------------------------------------------
// Destructor

ModemForward::~ModemForward()
{
}

//----------------------------------------------------------
// Procedure: OnNewMail

bool ModemForward::OnNewMail(MOOSMSG_LIST &NewMail)
{
    AppCastingMOOSApp::OnNewMail(NewMail);

    MOOSMSG_LIST::iterator p;
    for(p=NewMail.begin(); p!=NewMail.end(); p++) {
        CMOOSMsg &msg = *p;
        string key    = msg.GetKey();
        string value  = msg.GetString();
#if 0 // Keep these around just for template
string comm  = msg.GetCommunity();
double dval  = msg.GetDouble();
string sval  = msg.GetString();
string msrcc = msg.GetSource();
double mtime = msg.GetTime();
bool   mdbl  = msg.IsDouble();
bool   mstr  = msg.IsString();
#endif

if (key == "NAV_LAT") {
    navLat = msg.GetDouble();
}
else if (key == "NAV_LONG") {
    navLong = msg.GetDouble();
}
else if (key == "TRUE_DEPTH") {
    navDepth = msg.GetDouble();
}
//else if (key == "YSI_TEMP") {
//    ysiTemp = msg.GetDouble();
//}
else if (key != "APPCAST_REQ") // handled by AppCastingMOOSApp

    reportRunWarning("Unhandled Mail: " + key);

    return(true);

// ----------------------------------------------------------------------
// Procedure: OnConnectToServer

bool ModemForward::OnConnectToServer()
{
    registerVariables();
    return(true);
}

// ----------------------------------------------------------------------
// Procedure: Iterate()
//            happens AppTick times per second

bool ModemForward::Iterate()
{
    AppCastingMOOSApp::Iterate();
    // Do your thing here!

unetsocket_t sock;
  fjage_msg_t ntf;
  uint8_t data[NBYTES];
  uint8_t longData[DATA_NBYTES];

  for (int i = 0; i < DATA_NBYTES; i++) {
    longData[i] = 0;
  }

  int myNodeAddress = 225;
  int nextNodeAddress;
  int port = 1100;
  int rv;

  // Open a unet socket connection to modem
  printf("Connecting to %s:%d\n", "192.168.1.101", port);
  sock = unetsocket_open("192.168.1.101", port);

  // Bind to protocol DATA
  rv = unetsocket_bind(sock, DATA);

  // Set a timeout of 10 seconds
  unetsocket_set_timeout(sock, 120000);

  // Receive and display data
  printf("Waiting for a Datagram\n");

  ntf = unetsocket_receive(sock);
  if (fjage_msg_get_clazz(ntf) != NULL) {
    printf("Received a %s : [", fjage_msg_get_clazz(ntf));
    fjage_msg_get_byte_array(ntf, "data", data, NBYTES);
    for (int i = 0; i < NBYTES; i++) {
      printf("%d," , data[i]);
    }
    printf("]\n");
  }
  else {
    fjage_msg_destroy(ntf);
    printf("Error receiving data\n");
  }

  unetsocket_close(sock);

      data[3] == 84) {
// Open a unet socket connection to modem
    printf("Connecting to %s:%d\n","192.168.1.101",port);
sock = unetsocket_open("192.168.1.101", port);

    // Bind to protocol DATA
    rv = unetsocket_bind(sock, DATA);

    // Set a timeout of 10 seconds:
    unetsocket_set_timeout(sock, 120000);

    // Receive and display data
    printf("Waiting for a Datagram\n");

    ntf = unetsocket_receive(sock);
    if (fjage_msg_get_clazz(ntf) != NULL) {
        printf("Received a %s : [", fjage_msg_get_clazz(ntf));
        fjage_msg_get_byte_array(ntf, "data", longData, DATA_NBYTES);
        for (int i = 0; i < DATA_NBYTES; i++) {
            printf("%d," , longData[i]);
        }
        printf("]\n");
    }
    else {
        fjage_msg_destroy(ntf);
        printf("Error receiving data\n");
    }
unetsocket_close(sock);

    for (int i = 0; i < NBYTES; i++) {
        if (data[i] == myNodeAddress) nextNodeAddress = data[i+1];
    }

    // Pass the ordered list of nodes to the next node on the list
    unetsocket_t sock;
    int port = 1100;
    int rv;

    // Open a unet socket connection to modem
    printf("Connecting to 192.168.1.101:%d\n",port);

    sock = unetsocket_open("192.168.1.101", port);
    if (sock == NULL) printf("Couldn't open unet socket\n");

    // Transmit data
printf("Transmitting %d bytes of data to %d\n", NBYTES, nextNodeAddress);
    for (int i = 0; i<NBYTES; i++) {
        printf("%d," , data[i]);
    }
    printf("]\n");
    rv = unetsocket_send(sock, data, NBYTES, nextNodeAddress, DATA);

    if (rv != 0) printf("Error transmitting data");

    printf("Transmission Complete\n");
unetsocket_close(sock);

for (int i = 4; i < NBYTES; i++) {
    if (data[i] == myNodeAddress) {
        longData[30*(i-4)] = 36;
        longData[30*(i-4) + 1] = 73;
        longData[30*(i-4) + 2] = 78;
        longData[30*(i-4) + 3] = 70;
        longData[30*(i-4) + 4] = 79;
        longData[30*(i-4) + 5] = myNodeAddress;

        navLat_u.d = navLat;
        for (int j = 6; j < (6 + sizeof(double)); j++) {
            longData[30*(i-4) + j] = navLat_u.array[j - 6];
        }
        navLong_u.d = navLong;
        for (int j = 14; j < (14 + sizeof(double)); j++) {
            longData[30*(i-4) + j] = navLong_u.array[j - 14];
        }
        navDepth_u.d = navDepth;
        for (int j = 22; j < (22 + sizeof(double)); j++) {
            longData[30*(i-4) + j] = navDepth_u.array[j - 22];
        }
    } /*ysiTemp_u.d = ysiTemp;*/

148
for (int i = 30; i < (30 + sizeof(double)); i++)
{
    data[i] = ysiTemp_u.array[i - 30];
}/*
for (int j = 0; j < DATA_NBYTES; j++) {
    printf("%d, ", longData[j]);
} printf("\n");
sock = unetsocket_open("192.168.1.101", port);
if (sock == NULL) printf("Couldn't open unet socket");
rv = unetsocket_send(sock, longData, DATA_NBYTES, nextNodeAddress, DATA);
if (rv != 0) printf("Error transmitting data");
printf("Transmission Complete\n");
unetsocket_close(sock);
}
}
AppCastingMOOSApp::PostReport();
return(true);
} //---------------------------------------------------------
// Procedure: OnStartUp()
// happens before connection is open
bool ModemForward::OnStartUp()
{
    AppCastingMOOSApp::OnStartUp();
    STRING_LIST sParams;
    m_MissionReader.EnableVerbatimQuoting(false);
    if(!m_MissionReader.GetConfiguration(GetAppName(), sParams))
        reportConfigWarning("No config block found for " + GetAppName());
    STRING_LIST::iterator p;
    for(p=sParams.begin(); p!=sParams.end(); p++) {
        string orig  = *p;
        string line  = *p;
        string param = tolower(biteStringX(line, '='));

string value = line;

bool handled = false;
if(param == "foo") {
    handled = true;
}
else if(param == "bar") {
    handled = true;
}

if(!handled)
    reportUnhandledConfigWarning(orig);

registerVariables();
return(true);

// Procedure: registerVariables

void ModemForward::registerVariables()
{
    AppCastingMOOSApp::RegisterVariables();
m_Comms.Register("NAV_LAT", 0);
m_Comms.Register("NAV_LONG", 0);
m_Comms.Register("TRUE_DEPTH", 0);
    //m_Comms.Register("YSI_TEMP", 0);
    // Register("FOOBAR", 0);
}

// Procedure: buildReport()

bool ModemForward::buildReport()
{
    m_msgs << "============================================" << endl;
    m_msgs << "File:                                       " << endl;
    m_msgs << "============================================" << endl;

    ACTable actab(4);
    actab << "Alpha | Bravo | Charlie | Delta";
    actab.addHeaderLines();
actab << "one" << "two" << "three" << "four";
m_msgs << actab.getFormattedString();

return(true);
}

******* labtest5.txt *******
225, 33.291000, -87.519000, 0.316000
69, 33.291000, -87.518000, 3.083000

******* output613.txt *******
225, 33.217933, -87.548119, 0.243840
69, 0.000000, 0.000000, 0.000000