ABSTRACT

Over the last 23 weeks, our team has been designing what will be the framework for a stable control system to be used with an Underwater Unmanned Vehicle (UUV). Our end goal is to implement a graphical user interface that will allow the user to control the underwater platform’s lights, thrusters, and other components in a simple fashion. Via the GUI, visual feedback will be available to the user to provide real time information from hundreds of feet under water. Our design will be scalable, and the component addition limit has virtually no bound. Using an interface PCB driven by an Atmel Atmega 328 microprocessor, we will provide a framework for command prioritization standards to be used throughout the MSD robotic platforms program.

NOMENCLATURE

API - Application Programming Interface, a set of routines, data structures, object classes and/or protocols provided by libraries and/or operating system services in order to support the building of applications.

BAUD - the number of distinct symbol changes (signaling events) made to the transmission medium per second in a digitally modulated signal or a line code.

Bridge - The communications interface that this team has designed to facilitate data transfer to ancillary components, other bridges and/or a computer (running our software)

CTS - Clear to send protocol implemented on the RS232 data line.

RTS - Request to send protocol implemented on the RS232 data line.

GUI - Graphical User Interface

ROV - Remotely Operated Vehicle

RDP - ROV Data-Flow Protocol

CIP - Common Interface Protocol

BCP - Bridge Configuration Protocol

TCP - Thruster Control Protocol

LCP - Light Control Protocol

GDP - Gamma Device Protocol

SPI - Serial Peripheral Interface

USART - Universal Synchronous/Asynchronous Receiver Transmitter

TTL - Transistor to Transistor Logic

RS232 - standard for serial binary data signals connecting between a DTE (Data Terminal Equipment) and a DCE (Data Circuit-terminating Equipment)

RS485 - (aka EIA485) an electrical specification of a two-wire, half-duplex, multipoint serial communications channel.

UUV - Underwater Unmanned Vehicle

INTRODUCTION

The interaction between electrical and computer engineering is becoming more and more prevalent in today’s evolving technological world. The applications of these disciplines are virtually endless; ranging from control systems design to integrated communication systems. Our application of the hybrid computer/electrical engineering field brought us to the transportation phenomena discipline. Our task was to incorporate a stable system for controlling an underwater remotely operated vehicle (ROV). Our project piggybacked on the conclusion of two other related projects; P08454 and P08456. These legacy teams designed the hardware platform for the thrusters and lights that would be deployed on the underwater frame. Our mission is to take these components and interface them onto our own communications interface. The scope of the project includes taking a user input from a GUI on a computer, and sending the command down a data path to the communications bridge, where data is processed and routed accordingly. A completely new hardware system will facilitate these
In designing a reliable underwater control system, many considerations have to be made with the regard to power consumption, efficiency and streamlined communication. After carefully considering our customer needs and the outcome of the work we would be doing, we selected the best method for designing our software architecture and our hardware package. The team was split in half, with 3 members tackling hardware and 3 on software. Hardware customer needs were similar to the designs executed by the legacy teams. Much of the power distribution and communications mediums came directly from these legacy teams. For example, to facilitate an easier communications medium from the bridge to ancillary components, RS485 was chosen as the communications medium from the bridge to ancillary components.

**PROCESS ENGINEERING**

In designing a reliable underwater control system, many considerations have to be made with the regard to power consumption, efficiency and streamlined communication. After carefully considering our customer needs and the outcome of the work we would be doing, we selected the best method for designing our software architecture and our hardware package. The team was split in half, with 3 members tackling hardware and 3 on software. Hardware customer needs were similar to the designs executed by the legacy teams. Much of the power distribution and communications mediums came directly from these legacy teams. For example, to facilitate an easier communications medium from the bridge to ancillary components, RS485 was chosen as the communications medium from the bridge to ancillary components. From the get-go to facilitate easy modification of the code, and user interoperability. A GUI was then implemented in Java, accepting user input commands from a joystick, mouse and/or keyboard. The GUI provides visual feedback from the ROV as well, with feedback data coming from the lights, thrusters and any other components the user decides to attach. The customer needs were a key consideration in both facets of our project.

**HARDWARE DESIGN METHODOLOGY**

*Bridge Communications*

The hardware interface bridge was required to facilitate communication between the system components and the computer-based software. To properly support the legacy hardware, this required implementation of an RS485 communication bus to the slave devices. The bridge was also required to handle the condition where an ancillary module malfunctions by shorting the RS485 bus data lines. This failure condition required the ability to electrically isolate each module from the bridge. Because RS485 uses a differential pair signaling mechanism, opto-isolation would be too difficult to perform. To account for this failure condition, the communications bridge employs a time-sharing method whereby each module is given an equal amount of time on the RS485 data bus. In the case that a module is determined to be malfunctioning, the module will be given no more time on the bus, and thus will be unable to interfere with further communication. It should be noted that the communications bridge does not actually diagnose or look for hardware failures. This is left for the device or user to diagnose so that a proper course of action can be taken to fix the problem.

*Bridge Power*

The bridge is required to operate on a non-ideal, 24 V power supply based on its portability; the end-product will run on a battery. Since all digital components require 5 V input, some form of power (voltage) regulation is required.

![Power Division Circuitry](image)

A step-down buck regulator was chosen for this task due to its ability to operate over a wide variety of input voltages and due to its robust regulatory characteristics. The LM2674M-5.0 from National Semiconductor was implemented because of its robust regulatory characteristics. This specific buck regulator

---

**Figure 2 - Power Division Circuitry**

---

**Figure 1 - Rudimentary Hardware Block Diagram**

---

**User Interface**

**Surface Computer**

**Master Interface Microcontroller**

**Slave Interface Microcontroller**

---

Project P09201
will continue to regulate 5 V as long as the input voltage is over 5 V and can step-down input voltages as high as 48 V. Since one if the main specifications is low power consumption, efficiency is in the forefront of the design process.

![Figure 3 - Surface Mounted Power Circuit](image)

The need for low current makes the National Semiconductor product right for the application of small signal digital power. The LM2674M-5.0 can drive a maximum of 500 mA, which will be more than sufficient to power all components on the bridge. We have noted that a slight power glitch exists in the supply voltage to the microcontroller. Bypass capacitance of ~2uF stabilized performance.

**Test Bench**

In order to test the functionality of the bridge hardware, a test bench was designed to emulate all of the components and provide visual feedback. The architecture of the test bench closely mimics that of the bridge. The same power circuit is used to employ 5VDC to all digital electronics. In order to streamline the debugging process and simplify overall design, the test bench employed the same Atmega microcontroller as the communications bridge. The test bench contains various devices implemented to mimic any one of 3 devices. A 7-segment LED array is affixed to microcontroller outputs, allowing the user to test a generic ‘gamma’ device. The 7 segment displays obtain digitally converted analog data from a potentiometer also connected to the microcontroller. The test bench is also comprised of a multi-spectrum LED, and a small brushed DC motor to mimic the lights and thrusters. Through the use of all of the aforementioned testing subsystems, proof of function for the communications bridge is possible and also proves that the communications bridge will accept future senior design projects to add functionality to the ROV.

**SOFTWARE DESIGN METHODOLOGY**

Our software has one broad requirement: we require an interface capable of altering and displaying the state of a connected ROV. The state information must to be in the form of visual feedback (a GUI). The user interface is also expected to provide control of the ROV via a joystick, keyboard, and mouse combination.

To provide the required functionality, we divide our software into three distinct modules: microcontroller firmware, the hardware interface layer, and the GUI. This layered design came about for two reasons. First, we had three software developers in our team, and this allowed for an easy division of labor. Second, a layered architecture allows for modifications of code to be localized to a small section, saving developers from massive maintenance tasks.

To allow for inter-module communication, we also define several data protocols: a network protocol, termed RDP, for determining how to route data to modules in the ROV, CIP, a common protocol that all modules must implement, and several module specific protocols for interacting with the different modules on the ROV.

When designing CIP, consideration must be made with regard to goals greater than just the underwater ROV. CIP is meant to serve as the starting point for a device protocol common to all robotics platforms, including the RP-1, RP-10, and RP-100 projects.

Finally, our software is designed with the goal of being platform independent. Though it was not a requirement, we do not wish to place a restriction on the user of our system or on the next design team to use a specific operating system. This drives the design of the GUI to be written in Java.

**GUI DESIGN AND ARCHITECTURE**

The user interface allows the user to issue commands to, and receive feedback from, the ROV hardware. It is written in Java to allow cross-platform compatibility. The user interface is designed to be modular and to allow controls for additional hardware to be added or removed easily.

![Figure 4 - Command GUI Window](image)

The user interface is organized into separate windows. There is one window containing general
information regarding all hardware devices detected on the ROV as well as commands for interacting with multiple hardware components at once. If necessary, there may exist separate windows containing hardware-specific information and controls for individual types of hardware. These windows will only appear if a hardware device of the corresponding type is connected to the ROV.

Input to the interface is achieved through mouse, keyboard, and joystick. The visual components of the user interface may be interacted with through the mouse and, if necessary, the keyboard. A joystick is required to interact with the thruster hardware.

Each software component in the user interface can be classified as one of three types—window, hardware module, or protocol—as follows.

Window components are responsible for creating, showing, and hiding windows. They receive input from the user and the communications software and respond with appropriate actions by commanding hardware module and protocol items.

Hardware module items contain visual components specific to a certain type of hardware. A hardware module is bound to a corresponding window. The window governs the appearance of the module’s visual components, but all information necessary for proper operation of the components themselves is contained within the module. When the user interacts with a module’s components, the module must request its parent window to operate the appropriate protocol item if necessary.

Protocols implement the ROV communications protocol definitions. Each protocol should be owned by a parent window. The parent uses the methods of the protocols to send messages to the hardware devices through the communications software. Each protocol also has a matching callback interface which allows the protocol object to communicate asynchronously with its parent window. When the protocol needs to send a message to its parent window, it uses one of the methods in this interface. It is the responsibility of the parent to create an object that implements this interface and to pass it to the protocol upon its creation.

Protocols are implemented in the Java code (as opposed to native code) for a specific purpose. We expect the protocol definitions to change with time (especially the thruster and light control protocols), and by putting the protocol implementations entirely within Java, new releases of our main program can be created without needing to recompile the native libraries, saving future design teams that may not have access to all build systems.

**PROTOCOL DESIGN AND PURPOSE**

The protocol design is a two-layer stack. Using a two-layer stack allows a lower layer to deal specifically with getting data from place to place and a higher layer to generate and parse messages. The architecture is completely open to higher-level layers. The lower level is the ROV Data flow Protocol (RDP), which all modules will be required to implement. The Common Interface Protocol (CIP) is one of the higher layer protocols which all modules must also implement. Other protocols have been defined for the control layer, but are device specific and are not required on all modules. Similarly, any future modules will need to form their own control layer protocol.

RDP is designed to transfer data in the form of higher-level protocols. The goal of RDP is to make sure that data gets to its destination reliably. The RDP message format is simply a header followed by a variable amount of data. RDP has fields for data length, error detection, source address, destination address, protocol encapsulation, and protocol revision.
support for basic communications. The common commands include commands to retrieve a device’s information (e.g. device type), change a device’s RDP address number, report errors, upgrade software, and to mute/un-mute a device. Under normal operation, the hardware bridge will be utilizing the “get device info” command to discover devices and determine their addresses, and the “change device number” command to resolve address conflicts.

BCP is the device protocol for the hardware bridge. It is not required to be supported on any other devices. BCP contains commands related to the communications on the bridge, such as retrieving a list of attached devices, changing the hardware bridge behavior, and disconnecting a device from the bus.

TCP provides a set of commands specific to the thruster module. This command set includes the ability to change and retrieve the speed setting of the motor.

LCP provides a set of commands specific to the light module. This command set includes the ability to read the temperature of the light module, set the light color, and retrieve the current color.

GDP provides a small set of commands for an imaginary “gamma” device. The command set allows for the setting and retrieving of a value. This protocol is meant for use on our test fixture to prove that our software can interact with more than just a light and thruster module.

BRIDGE FIRMWARE DESIGN

The bridge is designed to move data from any source to any destination. There are several interfaces on the bridge; one is an uplink (to connect with the surface PC or another bridge), and the rest are downlinks. This conception of the bridge is constrained only in hardware; the software architecture is open, and would require modification only at the highest level to increase or decrease the number of interfaces in the system.

A Maxim® MAX3100 SPI controlled UART is used as the uplink interface. It is used in an interrupt-driven mode, with transmit and receive ring buffers. The sizes of the buffers are fixed at compile time, but are easily modifiable. The receive buffer is larger than the transmit buffer, since the interface can receive data at any point but can only transmit at specific intervals. The receive buffer is sized to accommodate the amount of data that can be received during the time that data is not being taken from the buffer. For the downlinks, the USART in the microcontroller is shared. This follows a similar buffering scheme as the uplink interface, but has a much smaller gap between receiving data and processing it. Since the same USART is used for all downlinks, it must be multiplexed in hardware. A decoder is used to perform the multiplexing, which is controlled by a port on the microcontroller.

To facilitate expansion of the system, each uplink and downlink is represented by a virtual interface. Each virtual interface is linked to one physical interface. A physical interface represents a specific means of transceiving data, such as a serial port. Multiple virtual interfaces can use a single physical interface. This facilitates the downlinks all sharing a single serial port.

Each virtual device can have several addresses associated with it. This is specifically designed to allow a slave bridge to expand the number of ports available to modules. When a packet is received and needs to be forwarded, the bridge will look at the list of addresses associated with each interface until it finds a match. The matched interface will then be used to determine the scheduling queue that the packet is put into. Packets with a low number in the priority field are placed at the front of the queue, and those with a high number in the priority field are placed in the back. The priorities can range from 0-255 for very fine-grain control.

The top level algorithm of the current bridge implementation is a time-shared system. In this, all of the virtual interfaces are allotted an approximately equal amount of time, which is further divided into transmit and receive phases. After device initialization, execution enters an infinite loop. The bridge will start with the first interface and begin sending packets. When a timer expires, the bridge stops transmitting packets and begin receiving them. When the timer expires again, it switches to the next interface and begins transmitting packets. When all interfaces have been handled, it starts the list of interfaces over and repeats the process.

HARDWARE INTERFACE DESIGN

The hardware interface layer of our software provides access to a joystick and the physical communication layer being used between the PC and our hardware bridge. These two components are referred to by the titles Joystick API (JSAPI) and Link Layer API (CLAPI). The Link Layer API’s acronym of CLAPI is a legacy from when the layer was originally referred to as the Communication Layer API.

JOYSTICK INTERFACE

The joystick API uses the SDL (http://www.libsdl.org/) library in order to access any joystick respecting the USB HID specification. The SDL library is cross-platform, allowing the joystick API to use a single code base on our end, making maintenance a much easier task. The SDL library is also developed.
out of RIT in an open-source community, so there will always be new features available for use.

The joystick API is a simple wrapper to a few key features of the SDL library (specifically, those pertaining to the use of a joystick). No additional functionality is provided by the joystick API that the SDL library does not already contain. JSAPI exists to present that functionality in a convenient form for developers in C and Java, and it allows developers to poll the joystick themselves or provide a callback function that will asynchronously update joystick state information.

The largest downside of the joystick API is its lack of plug-and-play support. The SDL developers are working on this limitation, and once it is officially released, it should not be hard to include this feature in the joystick API of this project.

LINK LAYER INTERFACE

The link layer API provides access to a computer’s physical interface. The link layer API is split into two sections: general functions and implementation specific functions. The general functions deal strictly with I/O, i.e. reading and writing of data. The implementation specific functions deal with details of the physical interface being used. In our case, these details are the BAUD rate, parity bit settings, stop bit settings, and hardware flow control. The link layer API is designed such that, should the physical layer be changed (for instance, if the physical layer is changed to USB instead of RS-232), the code using the link layer will need minimal modifications, confining the most significant changes to the link layer code itself. This will help decrease the amount of work required to maintain the project as it grows.

C-JAVA INTERFACE

In order for communications to flow between our GUI and hardware bridge, we needed to interface the GUI code, written in Java, with the link layer code, written in C. Java requires the use of the Java Native Interface (JNI) for it to be able to access functions compiled into the native language of the computer on which Java is running. Writing proper, robust JNI code represents a large amount of work, and has many pitfalls that must be avoided. To get around the problems of JNI development, the SWIG tool (http://www.swig.org/) was used to automatically generate the JNI code for us. This step is critical for accessing both the joystick and physical layer of the PC from Java code, and significantly reduces development time. It does not completely solve all problems associated with JNI development, though. Both JNI and SWIG can be extremely esoteric in their uses, and changes to either can sometimes be frustrating to those without a significant understanding of how either works. However, unlike JNI, SWIG restricts its esotericism to a small number of unlikely cases, and hopefully the SWIG files provided with our project will provide all the functionality required by future teams.

LESSONS LEARNED / CONCLUSIONS

The last 23 weeks provided the team with a vast array of tasks and hurdles to be overcome. Designing a system from the ground up teaches everyone pivotal important lessons about the design process. Perhaps the most important lesson learned is to always check your work several times before potentially making critical errors. When designing the PCB, the outline for the microcontroller was set in the layout software. This footprint was said to be the TQFP package in the software; but did not match the outline on the data sheet. Thankfully the package that was implemented was available for purchase from Atmel. The downside to this error was that the package on the PCB was 16 square millimeters in area. Pin spacing was a mere 0.2 mm. Additionally, the pad placement hindered our ability to solder the chip on the PCB. Finally, the most critical issue was an exposed ground plane on the bottom of the package that would short circuit on two vias that were placed inside the area of the chip outline. Our solution involved very small amounts of Kapton® tape and extremely precise soldering techniques. With a 100x microscope, the finest soldering iron tip available, and a vast amount of patience, the chip was secured and verified to function nominally. In order to fix the error, our PCB designer went back to the design software and implemented a change of package that matched the Atmega’s TQFP outline (much easier to place on board).

Another characteristic design challenge was creating our test bench. Surprisingly this turned out to be a task that would have been better done on a PCB. We found that using commercial off the shelf parts from Radio Shack didn’t provide the reliability needed for soldering and re-soldering. A similar design process used for creating the bridge applied to the test bench would have greatly streamlined the process. In the end, parts were placed on a bread board in a professional and easy to follow manner for ease of modification and component test.

This project also served to reinforce several ideals. Having a software team working alongside a hardware team as the hardware team developed a physical component showed the stark contrast between ease of testing in software and hardware. Whereas software developers can create a single, comprehensive, automated test that is able to verify a large amount of functionality very quickly (and more importantly, in a
repeatable manner), hardware developers are left with a much more arduous task of physically measuring and adjusting components. The idea that testing is often the largest, most time consuming component of a project was made clear. We were also reminded that it can be very helpful to add elements to a design that serve no functional purpose other than to aid in testing and verification.

The importance of clear communication between developers was also reinforced, as we ran into the occasional problem of wasted time while developers worked under different impressions on what their component was responsible for, how it would be used, or where exactly it fit in the entire project. It was interesting to see how the difference between a work environment where developers are within speaking distance of each other, and one in which the speed of communication is limited by e-mail and class schedules affected the output of each developer.

Finally, the design process expounded by RIT's senior design course served as an excellent reminder as to why the process has been abolished in industry as a viable strategy for software design. The tactic of having a requirements phase, a design phase, an implementation phase, and then a test phase--also known as the "waterfall model"--makes a critical misstep in assuming that any stage of the process can be completed before the next begins. Since the arguments against this model are already required reading for software engineering studies, there is no need to expand upon them here, and leave it up to the interested reader to further pursue the subject.

ACKNOWLEDGMENTS

The team would like to thank the institution for providing the financial and engineering support required to accomplish our mission. Individuals of merit include our customer, Dr. Hensel; our faculty advisor, Dr. Walter; and our TA, Todd Fernandez. Additionally, our most sincere gratitude is sent to Jeffery Lonneville at RIT’s Center for Electronic Manufacturing and Assembly. Without the support of the SMT lab, we would be nowhere!