Adaptive Firmware Framework for Microcontroller Development

Nicholas J. Tremaroli

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Alexander Leonessa, Chair
Leyla Nazhand-Ali
Alfred Wicks

May 8, 2023
Blacksburg, Virginia

Keywords: Firmware, Microcontroller, Framework, Robotics, Humanoid, Low-Level Networking, Low-Level Controls

Copyright 2023, Nicholas J. Tremaroli
Adaptive Firmware Framework for Microcontroller Development

Nicholas J. Tremaroli

(ABSTRACT)

Firmware development for Low-Level Controllers is an extremely complex task. Single-threaded microcontrollers are most commonly used for these controllers and thus are only capable of executing a single task at a time. Microcontroller software tends to be designed for an extremely specific task with little room for scalability or code reuse. Additionally, the state of a microcontroller at run-time is very difficult to observe and thus makes it harder to debug and develop these control systems. To alleviate these development issues, a software framework was designed to simplify firmware development for Hardware Abstract Layered (HAL) control systems. The software framework was implemented on Texas Instruments TM4C123GXL Tivas on a multi-joint robot with the purpose of experimenting on a distributed microcontroller system. All of the software for the microcontroller was implemented into one program with initialization files from the high-level controller to configure each individual Tiva based on its functionality in the distributed system. The EtherCAT communication protocol is used primarily for its fast communication speed between high-level and low-level controllers. A basic GUI development environment accompanies the framework to aid in the initial development of a custom controller firmware and thus reduce development time. Additionally, this framework is designed to be easily scalable such that a real-time operating system (RTOS) can be implemented with minimal effort should the developer desire to do so. The proposed software framework thus overcomes major challenges when developing firmware for low-level controllers making development overall less
time-consuming. Further, this framework can be used for many different robotic applications with a low-level multi-layered control architecture.
Adaptive Firmware Framework for Microcontroller Development

Nicholas J. Tremaroli

(GENERAL AUDIENCE ABSTRACT)

Microcontrollers and embedded systems are used everywhere in our daily lives in the technology we love. From microwaves, to cars, to phones, to toaster ovens, these systems are implemented in practically every piece of technology we use on a daily basis. Software development of these systems tends to be extremely complicated and complex. Hence, the software designed for these systems is usually overly specific to the device with little room for code reuse and/or scalability. This issue is extremely present in the field of robotics. To alleviate this issue, this work proposed, designed, and implemented a software framework for microcontrollers in a distributed network for robotics applications. Additionally, a graphical interface was developed to customize the software framework for a developer’s specific needs regarding what the application needs to do.
Acknowledgments

I would like to thank my advisor Dr. Alexander Leonessa for guiding me through the research process and giving me access to opportunities I could have only dreamed of. The feedback Dr. Leonessa gave me throughout my time as a graduate student made me a significantly better researcher and for that, I am forever grateful. I am extremely honored to have worked in the TREC Lab at Virginia Tech where this research was conducted. The experiences learned from being a part of this lab have given me an invaluable experience as a graduate student.

My student colleagues in the TREC lab have taught me so much about the field of robotics and the ability to work with them has been a dream come true. I would like to thank Connor Herron for supporting my research and giving me debugging tips related to electrical design. I would like to thank Ben Beiter for his knowledge of high-level controls and feedback when designing the low-level framework. I would like to thank Dr. Bhaben Kalita for his scientific research knowledge and guidance when writing professional publications. Finally, I would like to thank my TREC Lab colleague and friend, Maxwell Stelmack, it has truly been a pleasure to work with him throughout my time in the lab.

I would like to thank my Mom and Dad for their continuous love and support. When times throughout my research process were difficult they were always there to cheer me up and support me. My parents truly helped foster my love of technology from such a young age and have made me into the man I am today. They will always have my eternal love for all they have done to help me.

I would like to thank my childhood educators as these people helped me realize my love and passion for engineering. Thank you to Mr. Anthony Rich for sparking my interest in
technology from a very young age, and for showing me how much fun working with technology can be. Thank you to Mr. Douglas Albrecht for being such an amazing engineering teacher, for fostering my love to work in the field of robotics, and for traveling the country to compete in international robotics competitions. Thank you to Dr. Ken Ocheltree for also being a great robotics mentor and for truly helping understand software development for robotic applications. Thank you to Mr. Angelo Piccirillo and Mrs. Valerie Holmes for giving me such an incredible science research experience, for giving me the skills to work as a professional researcher, and for fostering my love for scientific research.
Contents

List of Figures x

List of Tables xii

List of Algorithms xiii

Listings xv

1 Introduction 1

  1.1 Motivation .................................................. 1

  1.2 Contributions ............................................. 2

  1.2.1 Open-Source Microcontroller Framework and Library ..... 2

  1.2.2 Integration with a high-level controller ................. 4

  1.2.3 Open-Source Guided Development ....................... 5

2 Review of Literature 7

  2.1 Current Popular Microcontroller Frameworks ............. 7

  2.1.1 Educational Frameworks ............................... 7

  2.1.2 Practical Frameworks ................................. 8

  2.2 The Gap .................................................. 10
# Low-Level Framework Features

## 3.1 Structure

- **3.1.1 Initialization Procedure** ................................................. 12
- **3.1.2 Control Loop Processing** .................................................. 12
- **3.1.3 EStop Check Loop** .......................................................... 13

## 3.2 Supported Sensors

- **3.2.1 Absolute Encoders** ........................................................... 15
- **3.2.2 Quadrature Encoders** ......................................................... 17
- **3.2.3 Force Sensors** ................................................................. 18
- **3.2.4 Force-Torque Sensors** ....................................................... 19
- **3.2.5 IMUs** ................................................................................. 20

## 3.3 Polymorphism and Sensor Integration

## 3.4 Dynamic Initialization

## 3.5 EtherCAT Communication

- **3.5.1 Tight-Chaining** ................................................................. 29
- **3.5.2 Loose-Chaining** ............................................................... 31
- **3.5.3 Communication codes** ...................................................... 32
- **3.5.4 Sample EtherCAT Frames** ................................................ 35

## 3.6 LED Debug interface

viii
4 Interface with High-Level

4.1 Structure ........................................... 39
   4.1.1 Initialization .................................. 40
   4.1.2 Control Loop Processing ...................... 42

4.2 Code Structure .................................... 43

4.3 Implementing for your application ................. 47
   4.3.1 Initialization Frame Processing Example .... 50
   4.3.2 Control Frame Processing Example .......... 55

5 GUI Development ..................................... 64

5.1 GUI Front End .................................... 64

5.2 GUI Back End .................................... 69

6 Results ............................................. 72

6.1 Data Visualization ................................ 72

6.2 PANDORA ........................................ 75

6.3 ForceBot .......................................... 77

6.4 Reinforcement Learning ............................ 78

7 Conclusions ....................................... 80

Bibliography ........................................ 82
List of Figures

3.1 The absolute encoders pre-programmed into the framework .................. 16
3.2 Encoder MR Type ML Part #225780 ........................................... 18
3.3 Futek Force Sensor Model LCM200 .......................................... 19
3.4 ATI Force Torque Transducer Mini45 Titanium .............................. 20
3.5 The MPU9250 IMU ............................................................... 21
3.6 The EasyCAT Pro ................................................................. 28
3.7 Tight Chaining Implementation .................................................. 30
3.8 Loose Chaining Implementation ................................................... 32
5.1 The starting home page of the GUI (enlarged to show detail) .............. 65
5.2 The window which shows when the user clicks on the development drop down window (enlarged to show detail) ................................. 66
5.3 The window which shows when the user clicks on a Tiva’s peripheral. In this case, the user clicked on Pin A2 and the corresponding information is being displayed (cropped and enlarged to show detail) ......................... 67
5.4 The interface for adding a sensor. Here the user clicked Add Sensors -> Absolute Encoders -> Gurley Encoder (enlarged to show detail) ......... 68
5.5 The interface for selecting a sensor color code (enlarged to show detail) ... 68
5.6 The result from adding the sensor, notice the Tiva Pins corresponding to the added sensor based on the user-selected color code. In this Figure, the user choose magenta (cropped and enlarged to show detail) .......................... 69

6.1 The setup used for testing the framework ............................................. 73

6.2 A plot of the MPID over time from which the user can determine the true update rate of the Tiva. The first row is the MPID, the second row is time.  .................................................. 74

6.3 A plot of the sensor readings changing as they are being interacted with. The first row is the MPID, the second raw is the Gurley Encoder reading, and the third row is the Futek Force Sensor reading.  ............................................. 74

6.4 A plot which shows the exact moment the Tiva’s EStop was triggered from the encoder going outside of its user-defined bounds. The first row is the MPID, the second row is the Gurley Encoder Value, and the third row is the halt code  .................................................. 75

6.5 The visualization setup in SCS2 made possible from the framework implemented 76
List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>CONTROL Signal Data Tiva-To-Master.</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>CONTROL Signal Data Master-To-Tiva.</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Error LED Blink codes</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>General LED Blink Codes</td>
<td>38</td>
</tr>
</tbody>
</table>
List of Algorithms

1  Initialization of Tiva .................................................. 14
2  Control Loop of Tiva .................................................. 14
3  EStop Check Of Tiva .................................................. 15
Listings

3.1 The Absolute Encoder Virtual Table ........................................ 22
3.2 Gurley Encoder Constructor Virtual Table Initialized .............. 22
3.3 Polymorphism Example .......................................................... 24
3.4 Initialization Enumeration .................................................... 25
3.5 EtherCAT Union of structures .................................................. 33
3.6 Brief EtherCAT Union of structures implementation example ...... 33
4.1 Sample JSON Sensor Initialization File for a Tiva .................. 40
4.2 Some of the TivaSlave helper functions to pack the EtherCAT frame 43
4.3 Abstract functions defined in TivaSlave.java which the user must implement for their application ........................................ 44
4.4 High-Level: The enumeration of Absolute Encoders programmed into the framework ......................................................... 45
4.5 Low-Level: The enumeration of Absolute Encoders being used to dynamically allocate the corresponding encoder ......................................................... 46
4.6 Code for structure of control signal frame to send to master computer ......................................................... 48
4.7 Example Implementation for sending initialization signals .......... 50
4.8 Example Implementation for receiving initialization signals ......... 52
4.9 Tiva Slave Deserialize helper function ..................................... 53
4.10 Control Data To Send example implementation .......................... 55
4.11 Packing Control Signal Frame example implementation ................ 56
4.12 Storing the incoming Control Signal Frame from the Tiva ............... 57
5.1 Use of mapping from command to function pointer in back-end .......... 70
Chapter 1

Introduction

1.1 Motivation

Embedded systems, particularly microcontrollers are used throughout the field of robotics to aid in the computation and movement of multi-joint robotic systems\cite{14, 15, 27}. The most common industry standard for microcontroller development on these multi-joint robots is to have a distributed microcontroller system\cite{38}. In this architecture, a central computer commands each individual controller to perform a specific task and then reports back with updates on the state of the system. This architecture makes complex robotic systems significantly more flexible and allows for the separation of high-level and low-level control. Usually, each individual microcontroller is responsible for a particular joint or sensory reading on the robot.

While microcontrollers provide a flexible way of providing control architecture development, there are some key issues with microcontrollers that cause time-consuming and tedious software development. Microcontrollers are significantly harder to develop software for as there is often ambiguity related to the internal state of the device during testing. These devices are significantly harder to work with by nature as they are not built to be a full computer system and thus do not have advanced features to aid with development and software debugging. These issues are so niche and complex that the field of Firmware Engineering branched from Computer Engineering\cite{20}. Firmware Engineers are specifically focused on tailoring a
microcontroller for the specific needs of a particular application.

To account for the limited resources of a microcontroller, a firmware engineer must carefully examine each task the microcontroller has to accomplish and tailor the code in such a way that it is able to run as efficiently as possible. With each microcontroller being tailored specifically for a particular application, a key problem arises, the limited ability for code reuse. Code reuse is an immensely valuable principle within the field of software development, significant time has been spent researching the most effective patterns for this concept[23]. This principle would be especially valuable for microcontrollers. If done properly, code reuse can eliminate the majority of the tedious tasks related to microcontroller development by structuring it to be more abstract and more applicable to a variety of other robotic systems.

There is a key trade-off with this philosophy, code abstraction, while great in concept, tends to lead to a slight degradation of performance. On a typical computer with virtually unlimited resources when compared to a microcontroller, this is no problem. However, depending on the application of the microcontroller, this slight degradation in performance can limit the scope of the applications in which the microcontroller can function. Additionally, microcontroller programming is most commonly done in the C programming language which makes code reuse difficult because C is not an Object-Oriented Language by nature[36] and thus makes software abstractions more difficult.

1.2 Contributions

1.2.1 Open-Source Microcontroller Framework and Library

This work aimed to find a so-called ”sweet spot” in this philosophy by developing an abstract framework and library for the Texas Instruments TM4C123GCL Tiva board specifically
geared toward multi-joint robotic systems. The goal is to abstract the software for various robotic applications in such a way that performance does not suffer tremendously. The framework and library were designed to provide the user with a variety of different useful features when working with microcontrollers. Such features include custom communication protocols built using EtherCAT, dynamic port/sensor configuration from the high-level computer, high-level re-initialization protocols allowing for sensor swaps without redeploying code to the microcontroller, custom-implemented polymorphism allowing for additional peripherals to be added to the framework with ease, easy to follow low-level control loop structure, virtual estop capabilities, and an easy to use high-level data visualization using the IHMC Open-Source Robotics Software [29].

Such a framework and library would greatly shorten the amount of time needed to be spent on microcontroller programming as all of the communication/control structure and sensor programming interface are already completed. This would allow roboticists to spend less time debugging microcontroller-specific errors, such as memory leaks, sensor communication, pin conflicts, and various faults, and instead focus on their high-level control architecture which is at the heart of what they are trying to solve. Should roboticists choose to use this library and framework developed for the TI Tiva, they would truly save a significant amount of time when developing their application.

Similar frameworks have been constructed in the field of computer science with the goal of saving development time. These software frameworks are free and open-source so software developers are able to contribute valuable changes they would like to see to the framework from which other programmers can benefit. A few examples of such frameworks include Vue JS and Django which have revolutionized the field of web development [21, 22]. These frameworks help simplify the development of their respective applications and common programmers can use them to reduce the development time of their applications.
This work aims to have a similar impact in the field of microcontroller development for robotic applications. While there do exist a few frameworks for microcontroller development which aim to solve this issue, they suffer from being over-abstracted, overly simplified, and as a result, usually do not take full advantage of the microcontroller’s capabilities. Upon completion of the library and the framework, the software will be made open-source to the growing robotics community. Roboticists from around the world will be able to use and take advantage of this software to skip the majority of the backend microcontroller development and instead focus on their final product. Additionally, this framework can be easily extended to other microcontrollers outside of the TI Tiva. The only major difference would be the HAL programming and drivers for the respective microcontroller. In an ideal case, the code written would be able to be copied over with the only change needing to be made would be the interface with the respective microcontroller’s drivers. From the vast and seemingly endless possibilities of this microcontroller framework, the open-source software has the potential to grow immensely in popularity throughout the robotics community and we could begin to see this framework being implemented by roboticists around the world for their particular applications.

1.2.2 Integration with a high-level controller

As mentioned previously, this framework will come built with various high-level communication protocols using the IHMC Open-Source Robotics Software. IHMC provides great open-source software related to EtherCAT communication[28] and data visualization for robotics[30]. These are the two main features taken advantage of when developing this framework. Strong high-level software combined with a flexible and easy-to-use low-level framework would make microcontroller development and testing significantly faster. Additionally, communication protocols would be built on top of the provided IHMC software
to provide more functionality to the low-level microcontroller. Dynamic initialization/re-initialization of the microcontroller would greatly assist when testing and will be added to the framework. This technique will allow the high-level computer to send the Tiva information regarding which sensors/peripherals to use upon an initialization sequence. The Tiva would then configure itself based on the supported functionality the high-level computer needs to run. This initialization sequence can be designed in such a way that re-initialization is possible without a power cycle. This concept would be immensely powerful for development. The Tiva could be initialized with a set of sensors on various pins, and then immediately be re-configured to serve different sensors without needing to edit and redeploy the low-level code. This is an extremely important feature of the framework. The most popular microcontroller frameworks do not focus on higher-level interaction and thus the user must employ this piece themselves.

1.2.3 Open-Source Guided Development

As an additional tool to help simplify microcontroller development, an interactive web interface GUI will be constructed to model the TI Tiva and the available sensors. This tool is appealing to engineers who do not come from a computer engineering background and who do not want to work directly with the library and framework developed but instead leverage its capabilities in a user-friendly way. A developer will be able to use this GUI to specify which peripherals they would like and which pins to place these peripherals on the Tiva. The GUI will interactively query a backend knowledge base of the Tiva to check the validity of these sensor configurations. It will be able to detect and prevent pin conflicts as well by having knowledge of each of the Tiva pins and their respective underlying communication protocols supported.
The development of a GUI to guide microcontroller development in this fashion will truly be a game-changer in microcontroller development. The developer can simply use the GUI to specify which peripherals are located on a particular microcontroller, generate the low-level code and high-level communication code with the click of a button, and progress through the development of their application without having to worry about microcontroller-related issues. This will allow microcontroller development to go from a tedious and difficult development problem to a plug-and-play solution for a wide range of robotic applications.

The work that will go into modeling the Tiva for such an interactive GUI will also have numerous benefits. By modeling the Tiva and its respective pin maps, support communication protocols, and various built-in functionalities, one can easily use this model as a basis for hardware simulations. While this is not the overall goal of developing the functionality required for an interactive GUI, it will still be able to serve as a starting point should someone choose to use the backend code as a basis for hardware-related simulations.
Chapter 2

Review of Literature

2.1 Current Popular Microcontroller Frameworks

In order to understand the full power and scope that a flexible microcontroller library and framework can provide, it is first necessary to understand which tools already exist along with how microcontrollers fit into various robotics applications. From understanding the current state of microcontroller frameworks and development tools, one will be easily able to see and identify the key gap that this work aimed to solve.

2.1.1 Educational Frameworks

A few frameworks have been developed to be centered around microcontroller development for educational purposes. Such a framework like PyBoKids [37] and ScratchX [10] were developed with the entire goal of abstracting microcontroller development so students can learn the fundamentals of robotics in a classroom environment. Frameworks like these truly have a valuable purpose in the realm of education and learning. The abstraction techniques used for these frameworks are geared to take out a significant amount of ambiguity related to the microcontroller such that the code is easy to read, understand, and learn. A downside to these educational frameworks is that they are overly abstracted and do not take full advantage of the microcontroller capabilities. As a result, such educational frameworks are
typically not used for more advanced systems.

### 2.1.2 Practical Frameworks

More practical microcontroller frameworks have been developed with the key idea of simplifying the development of such devices. There are three most common frameworks which are used throughout microcontroller development with this key goal in mind. Energia was an immensely popular framework and IDE designed for Texas Instruments products until it was discontinued in 2019 [2]. It is geared toward engineers who are not as familiar with microcontroller development and simplifies a lot of the low-level programming features and challenges. Energia has proven to be useful for simple applications related to microcontroller development and many have used the simplicity of the open-source framework to their advantage [32]. However, due to this abstraction, many developers faced various issues when trying to take advantage of more advanced features with their respective microcontrollers as not all features were enabled through this framework. Additionally, the Energia IDE did not provide comprehensive debugging features as CCS did, such a tool is required when designing and debugging more complex applications.

RPi.GPIO is an open-source library for Python to interface with the GPIO Pins of the Raspberry Pi microcomputer [9]. This library has many benefits for the end developer. The software is extremely easy to use for a variety of different applications [19]. The library is well documented. There is currently a strong community of developers who assist with development discussions in forums. Many successful Raspberry Pi projects have been made using this library. The only downside to this framework is that it uses Python. While Python is a great language for its simplicity and easy code structure[34]. It suffers from being a slow language and is not as resource efficient as a language like C[33] or C++[24]. Hence, while
this framework may be useful for a wide range of microcontroller-related projects, it is not ideal for multi-joint robotic systems in which speed and resource management are key to designing a successful system.

Wiring is an open-source software suite which contains a framework, library, language, and IDE that many easy-to-use microcontrollers, such as Arduino are based on[17]. Similar to RPi.GPIO, this framework is used by many microcontroller developers and thus has a lot of support in terms of documentation and online forums. The wiring language is written as what the developers call a "thin layer" which sits on top of C/C++ to simplify programming development for its end users[17]. Wiring was initially written for 8-bit AVR microcontrollers, however, other developers have extended this framework for their particular microcontrollers. Additionally, the Wiring IDE allows for all of the user’s written code to be constructed in a programming environment that has the libraries built into the development and thus the user does not have to worry about building-related issues. The wiring library is one of the best and most common frameworks for microcontroller hobbyists and has been used on several Arduino projects. However, there are some key drawbacks to this software suite. The wiring IDE, similar to the Energia IDE, does not provide complex debugging features which are essential to complex microcontroller development. The wiring language can be viewed as an over-abstraction of the microcontroller code which can lead to inefficiencies during the final application. Additionally, the wiring framework only provides a way to interface with various microcontroller peripherals and not the actual computation needed to make such peripherals useful. Therefore, while this framework is used for many microcontroller projects, it suffers from over-abstraction and does not provide an interface for common robotic sensors.
2.2 The Gap

Many of these microcontroller frameworks serve their purpose with their respective application. Frameworks centered around learning make for great experiences for students and help them begin to understand the world of microcontrollers. More practical developed frameworks and libraries such as Energia, RPi.GPIO, and Wiring, aim to make microcontroller development simpler for a specific type of application. These practical frameworks are used throughout the field of microcontroller development for certain projects, however, they lack a place when it comes to developing software for multi-joint robotic systems. Most of these practical frameworks suffer from being overly abstracted to simplify development or lack key software development tools such as a debugger. Additionally, all of these microcontroller frameworks only focus on basic communication for external peripherals and do not provide an easy way to process this data or communicate it to a high-level computer for other various applications.

The key gap in this field is the lack of an easy-to-use, simply abstracted microcontroller framework which supports various high-level communication, control, and testing features along with essential peripheral communications as well. Such a framework would not only assist with software development on a microcontroller but would also provide a solution for how to connect the device to a greater network of distributed microcontrollers. The valuable high-level communication features previously mentioned would greatly assist with microcontroller development and testing. The goal that this work aims to solve is to make an extendable and scalable microcontroller framework that supports peripheral integration, high-level networking, and high-level protocols which can be used for dynamic configuration of the microcontroller to limit hard coding peripheral locations and peripheral-related constants.
Chapter 3

Low-Level Framework Features

The framework and library proposed in this thesis are structured to provide the user with a smooth development process while not suffering from over abstraction. Several key features make up the structure of this software framework. Easy sensor integration provides the user with the ability to scale the framework as needed for their application by using polymorphism. Dynamic initialization provides the user with a method of easy testing during development as the microcontroller and its peripherals can be tested in various combinations without needing to redeploy code. The EtherCAT communication provides a state-of-the-art and blazingly fast communication method for the microcontroller to communicate with the high-level computer. An interface with the high-level computer is established for this framework and is based on the IHMC Open-Source Robotics Software. Finally, a virtual emergency stop is built in to prevent actuators from being commanded when a sensor reading goes out of its respective bounds. The virtual EStop is equipped with various halt codes that signal to the user which sensor triggered the EStop to assist with debugging. This section will go in-depth into each of these features to describe how they were implemented and their usefulness to the end user.
3.1 Structure

The simplicity of the framework’s structure is truly what attracts end-users. By having a structure that is easy to understand, the user can better implement software for their respective robotic application. This understanding of simplicity was taken into extreme consideration when designing this framework and was the driving force behind its development. After careful consideration, the flow of the underlying low-level controller code was organized into three main components: Initialization, Control Loop Processing, and EStop Check Loop. The algorithms for these components can be seen below in Algorithm 1, Algorithm 2, and Algorithm 3 respectively.

3.1.1 Initialization Procedure

For the Initialization Procedure, the Tiva waits in a loop for the high-level computer to startup and begin sending initialization frames. As the initialization frames come in, the Tiva stores them accordingly for the corresponding sensors. The correlations between the initialization frame number and the sensor are determined by the user of the framework, typically each initialization frame is meant to provide data for a single sensor, however, this is up to the user’s design pattern. Once all of the initialization frames have been processed, the Tiva breaks out of the loop and initializes all of its peripherals based on the data the high-level computer sent it.

3.1.2 Control Loop Processing

The Control Loop Algorithm is run inside an Interrupt Service Routine (ISR) and is a timer-based interrupt. The user can set the desired update rate of the Tiva by specifying the rate
at which this timer triggers. The reason all of the computation is done within an interrupt is to assist with discrete time assumptions. By having all of the computation done in a single ISR, we are able to reaffirm all of the discrete calculations related to the high level controller. Additionally, we are able to determine if the Tiva is updating at the desired rate. By having all of the computation done in a single interrupt, no other interrupts from sensory signals will be triggered. While this goes against the typical practice, it was determined that this described simplifies the structure of the software. Traditionally, having multiple interrupts for each sensor causes ambiguity in the rate which these sensors are updating. The use of a single ISR for the control loop allows the high-level computer to determine the true update rate of the Tiva because it waits for data for each of the connected sensors before it sends data back to the high-level computer. This allows the high-level computer to accurately determine if the Tiva and its sensors are actually achieving its desired update rate.

### 3.1.3 EStop Check Loop

The EStop algorithm is run inside another ISR and runs at the same rate as the control loop. The EStop check does not take a significant amount of time and is extremely fast as no peripherals need to be interacted with. The routine checks that each sensor reading is within its correct limits. If they are not, a corresponding halt code is set pertaining to that particular sensor and the Estop is set to be triggered. Once the EStop is triggered, the control loop will no longer send PWM commands to the motor controllers. However, it will continue to read data from the sensors and send that data along with a corresponding sensor halt code, back to the master for debugging purposes.

The Tiva cannot come out of an EStop state by simply moving the troubled sensor back to its acceptable range. This is for safety purposes so there is no case in which the Tiva goes
into an EStop, relaxes the actuators and immediately comes out, and continuously loops between an EStop state and a working state. The Tiva can only come out of an EStop triggered state by fixing the troubled sensor back to an acceptable range, and going through the process of re-initialization which will be described in a later section.

Algorithm 1 Initialization of Tiva
1: Microcontroller turns on
2: \textit{tiva.Initialized} $\leftarrow$ \textit{false}
3: \textbf{while} \textit{!tiva.Initialized} \textbf{do}
4: \hspace{1em} Receive initialization frame data from high-level
5: \hspace{1em} \textbf{if} new initialization data \textbf{then}
6: \hspace{2em} Process initialization frame data
7: \hspace{2em} Send response to high-level that the frame has been processed
8: \hspace{2em} \textbf{if} all initialization frames have been processed \textbf{then}
9: \hspace{3em} \textit{tiva.Initialized} $\leftarrow$ \textit{true}
10: \hspace{2em} \textbf{end if}
11: \hspace{1em} \textbf{end if}
12: \textbf{end while}
13: Proceed to Control Loop

Algorithm 2 Control Loop of Tiva
1: Control Loop Timer triggered
2: Read from Tiva Sensors
3: \textbf{if} EStop not triggered \textbf{then}
4: \hspace{1em} \textbf{if} new control signal received from high-level \textbf{then}
5: \hspace{2em} Store data from high-level
6: \hspace{2em} Process and apply data from high-level
7: \hspace{2em} Send sensor data to high-level
8: \hspace{1em} \textbf{else}
9: \hspace{2em} Process and apply data from high-level
10: \hspace{2em} Send sensor data to high-level
11: \hspace{1em} \textbf{end if}
12: \textbf{else}
13: \hspace{1em} Send sensor data and halt code to high-level
14: \hspace{1em} \textit{tivaInitialized} $\leftarrow$ \textit{false}
15: \textbf{end if}
Algorithm 3 EStop Check Of Tiva

1: EStop Timer triggered
2: if Sensor A out of user-define limits then
3:   \( tiva.haltCode \leftarrow SensorAHaltCode \)
4: else if Sensor B out of user-define limits then
5:   \( tiva.haltCode \leftarrow SensorBHaltCode \)
6: else if Sensor C out of user-define limits then
7:   \( tiva.haltCode \leftarrow SensorCHaltCode \)
8: ...
9: end if
10: EstopTriggered \( \leftarrow tiva.haltCode! = 0 \)

3.2 Supported Sensors

During the development of the library, common robotic sensors were built to give programmers a head start on their microcontroller development. The sensors pre-built into the library are common and used all around the field of robotics. The library includes various support for Absolute Encoders, Quadrature Encoders, Force Sensors, Force-Torque Sensors, and IMUs. Each of these sensors is programmed in its own unique structure and supports polymorphism for code conciseness, ease of operation, and scalability.

3.2.1 Absolute Encoders

Absolute Encoders provide important feedback when determining the position of a robot’s joint. These sensors are unique in that they can provide the exact position and velocity of a joint on a robot at any given time. They are widely used across robotic applications and provide essential feedback on the state of the joint with respect to the overall control system. This library comes pre-built with two popular absolute encoders, the Gurley Encoder model A19, and the Orbis Encoder model P01_07. The Gurley Encoder model A19 which can be seen in Figure 3.1a uses optical electronic technology to measure angular position.
has a maximum resolution of 18 bits and communicates over the SSI protocol to the TI Tiva. The encoder ranges from $0^\circ$ to $180^\circ$ and has a resolution of $\pm 0.021^\circ$ [12]. The Orbis Encoder model P01_07 which can be seen in Figure 3.1b provides a 14-bit resolution with an accuracy of $\pm 0.25^\circ$. The encoder ranges from $0^\circ$ to $360^\circ$ and communicates over the SSI protocol to the TI Tiva [13].

There is a simple algorithm that sits on top of the Absolute Encoders and is provided in the form of an overarching Joint structure. This structure adds extra features and algorithms to the absolute encoder readings to make them safer and easier to function with. The Joint structure adds a relative zero position, forward range of motion, and backward range of motion on top of the encoder readings. This allows the absolute encoder values to be zeroed at a corresponding position properly and yield results that are easier to read. Additionally, the range of motion values are used as a safety feature. If the robot joint goes outside the set ranges of motion, the framework’s virtual E-Stop will be triggered.
The Joint structure takes care of standard issues related to absolute encoders. Some absolute encoders have different "wrap-around values" such that once the encoder reaches the final value (180° or 360° depending on the encoder), it wraps back around to a value of 0°. This is not practical for mounting on a robotic system as it will have to be mounted in such a way to prevent this wrap-around. Instead, a custom algorithm moves this wrap-around value outside the user-defined range of motion of the encoder. This is computed by using the zeroing value and the ranges of motion provided by the Joint structure. Hence, the user can mount the absolute encoder any which way on the robotic joint, set the corresponding ranges of motion, and not have to worry about this wrap-around problem.

The final common issue this Joint structure handles has to do with the absolute encoder’s corresponding final value. Some encoders have their final value set to 180° and some others have it set to 360°. To account for this, the Joint structure treats each encoder as if their wrap-around value is 180° so that a wide variety of encoders can be added to the Joint structure and still work properly. Should the end user want to change this in the framework to function differently, they can easily do so.

### 3.2.2 Quadrature Encoders

Quadrature Encoders are types of incremental encoders that provide relative feedback on the position and the direction a joint or a motor is moving in. These encoders differ from absolute encoders in that these quadrature encoders need to have a relative home position in order to accurately measure the distance traveled. These encoders can be used to measure the position of a joint on a robot (just like absolute encoders) or can be attached to a motor to count the number of revolutions and direction the motor is moving in from its home value. This feedback can be essential when designing controllers for various actuators.
and be combined into a Kalman filter to provide important data relative to various sensor components on the robot. The Quadrature Encoder which was pre-programmed into the framework is the Encoder MR Type ML Part #225780 which can be seen in Figure 3.2. This encoder contains three channels, has a maximum operating frequency of 200KHz, a maximum speed of 12000RPM, a resolution of 1000 counts per turn, and used the QEI protocol to communicate to the Ti Tiva [1].

![Figure 3.2: Encoder MR Type ML Part #225780](image)

### 3.2.3 Force Sensors

Force Sensors are extremely valuable when it comes to robots as many popular controller algorithms use force or torque control. Typical force sensors output a linearly correlated varying voltage depending on the amount of compression or tension force being applied to it. Force sensors need some type of calibration process. By having linear properties, this calibration process can be set by having an offset point and a slope. These two values are required during the construction of a force sensor in the provided library to output the correct and corresponding force. The force sensor which was pre-programmed into the framework was the Futek Force sensor model LCM200 which can be seen in Figure 3.3. This force sensor features a rated output (RO) of 1-2mV/V nom, a nonlinearity factor of ±0.5% of the RO, a hysteresis factor of ±0.5% of the RO, and a nonrepeatability factor of ±0.1% of the RO. This sensor outputs voltages depending on the compression and tension forces the sensor is
experiencing. Hence, this sensor communicates its force data to the Ti Tiva via an ADC interface which converts these varying voltages to digital values [5].

3.2.4 Force-Torque Sensors

Force-Torque sensors (FT sensors) provide 6-DOF feedback related to the force vector and the torque vector that the sensor is experiencing. These sensors provide this feedback in three directions (Cartesian directions: X, Y, Z) and are usually placed at the end of a robotic limb to determine what the robot is interacting with. Many FT sensors have their own built-in microcontroller to handle calibration processing, sampling, and other processes related to sensor data extraction. The FT sensor pre-programmed into this library was the ATI Force Torque Transducer Mini45 Titanium which can be seen in Figure 3.4. This FT Sensor can accurately measure forces in the X and Y directions up to ±3000N and ±6400N in the Z direction. As for torques, this FT Sensor can measure up to 9700Nm/rad for the X and Y axis torques and up to 20000Nm/rad for the Z axis torque [16].
3.2.5 IMUs

Internal Measurement Units (IMUs) are extremely valuable sensors that help understand the internal positional state of the robot. These sensors usually measure acceleration, angular momentum, and direction. These measurements can be used in sensor fusion algorithms to obtain Euler angles for a particular component on the robot or the entire robot itself. IMUs are extremely common for state-space estimations. They are commonly used to process how a control algorithm affects the estimated state of the robot. The IMU which was pre-programmed into this framework was the MPU 9250 which can be seen in Figure 3.5. This IMU has a triple access accelerometer with $\pm 2g$, $\pm 4g$, $\pm 8g$, $\pm 16g$ scale ranges, a gyroscope with $\pm 250^\circ/sec$, $\pm 500^\circ/sec$, $\pm 1000^\circ/sec$, $\pm 2000^\circ/sec$, and a magnetometer (through an embedded AK8963) with a three-axis magnetic sensor, magnetic concentrator, an output resolution of 14-bits, and a full-scale measurement range of $\pm 4800\mu T$. The sensor uses the I2C protocol to communicate with the TI Tiva and requires an initialization procedure for all of the features and an additional calibration process for the gyroscope readings [6]. Due to the extensive functionality of this sensor, it takes the longest to initialize out of all the pre-programmed sensors in the framework with an initialization time of about two seconds. The initialization procedure is tailored to a respective resolution for the accelerometer, gyroscope, and magnetometer. These resolutions were chosen based on the most common applications and best user experience (reported on forums) for this sensors. Should the end-user require a different resolution, they are free to edit the initialization sequence.
3.3 Polymorphism and Sensor Integration

While the framework has various pre-programmed sensors for a user to implement and use in their application, it also supports additional sensor integration with ease. This is truly what makes the framework powerful: the ability to easily add more sensors and contribute to a larger open-source community. The framework takes advantage of custom-implemented polymorphism in C with the various different types of sensors. This allows sensors to be added and extended from various sensor categories in order to follow the same outline as the base sensor structure. The implementation of the Gurley and Orbis Encoders as Absolute Encoders will be used as an example to demonstrate the strength of polymorphism in this framework. The low-level firmware code contains header and source files for both the sensor category and the various sensors which fall under that sensor category. In an object-oriented language such as C++ the sensor category would be considered the base class and the various sensors which fall under that category would be the derived classes that inherit the base class. In this example, the Absolute Encoder structure can be treated as the equivalent base class, and the Gruley and Orbis Encoders can be treated as the equivalent derived class.

In order to custom implement polymorphism in C, the base structure must define a virtual table that re-directs functions call to the appropriate derived structure and its respective functions for processing the sensor. The virtual table for Absolute Encoders and their cor-
responding function names can be seen in Listing 3.1. This virtual table specifies that all Absolute Encoders which are implemented in the framework must have a corresponding function call for an enabling feature, a read sensor feature, and a freeing feature. During the construction of derived Absolute Encoder structures (the Gurley and Orbis Encoders in our example), the function pairing for the virtual table is assigned along with the base structure’s data fields. An example initialization of the virtual table for the Gurley Encoder can be seen in Listing 3.2. The initialization of the virtual table shown in this listing can be read as follows: AbsoluteEncoderEnable will point to GurleyEncoderEnable, AbsoluteEncoderReadSensor will point to GurleyEncoderRead, and AbsoluteEncoderFree will point to GurleyEncoderFree when the constructed Gurley Encoder structure is passed as an argument to the purely abstract virtual functions.

```c
struct AbsoluteEncoder_vTable
{
    void (*AbsoluteEncoderEnable)(struct AbsoluteEncoder*);
    void (*AbsoluteEncoderReadSensor)(struct AbsoluteEncoder*);
    void (*AbsoluteEncoderFree)(struct AbsoluteEncoder*);
};
typedef struct AbsoluteEncoder_vTable AbsoluteEncoder_vTable;
```

Listing 3.1: The Absolute Encoder Virtual Table

```c
AbsoluteEncoder* GurleyEncoderConstruct(uint32_t SSIBase, uint16_t sampleRate)
{
    static const AbsoluteEncoder_vTable vtable =
        { GurleyEncoderEnable, GurleyEncoderRead,
```

22
GurleyEncoderFree

};

AbsoluteEncoder AbsoluteEncoderBase;
AbsoluteEncoderBase.vtable = &vtable;
...

Listing 3.2: Gurley Encoder Constructor Virtual Table Initialized

After the derived structure initializes the base structure’s respective features, it then dynamically allocates space to store these fields as well as any other fields specific to the derived structure which is being constructed. In the given example, after this construction is complete, the Absolute Encoder functions (such as AbsoluteEncoderEnable for example) can be called on the corresponding Gurley and Orbis Encoder structures and the virtual table will point to the corresponding function to properly initialize these sensors.

This polymorphism feature offers immense power and flexibility when it comes to the framework’s scalability. Any new sensor can be added to a corresponding sensor category, create functions around the virtual table given, have its own respective private data elements used for processing, and be easily called from the base structure’s virtual functions. Additionally, each derived sensor structure can also have its own private functions for other types of processing should a particular sensor offer more functionality than is supported by the base structure. An example of the effectiveness of polymorphism can be clearly seen below in Listing 3.3. As one can see in the listing below, the true power of polymorphism comes from how the element is constructed. Both the Gurley and Orbis Encoders are Absolute Encoders hence their dynamic memory can be stored as an Absolute Encoder pointer and be called by Absolute Encoder functions. However, depending on the encoder constructed, the
Absolute Encoder function will point to the corresponding derived class function to handle the respective task.

```c
AbsoluteEncoder* encoder1 = GurleyEncoderConstruct(...);
AbsoluteEncoder* encoder2 = OrbisEncoderConstruct(...);
AbsoluteEncoderEnable(encoder1); // calls the gurley encoder enable function
AbsoluteEncoderEnable(encoder2); // calls the orbis encoder enable function
...
while(Running)
{
    ...
    AbsoluteEncoderReadSensor(encoder1); // will call the gurley encoder function
    AbsoluteEncoderReadSensor(encoder2); // will call the orbis encoder function
    ...
}
AbsoluteEncoderFree(encoder1); // calls the gurley encoder free function
AbsoluteEncoderFree(encoder2); // calls the orbis encoder free function
...
```

Listing 3.3: Polymorphism Example

### 3.4 Dynamic Initialization

Dynamic Initialization is a feature which provides the user an immense amount of flexibility when testing their design. The framework and library were programmed so the master computer has all knowledge of the Tiva’s respective low-level sensors and the types of sensors it is using. The high-level code was written to provide an interface to the user which allows
them to customize how these sensors are used on the Tiva (the interface is described in a later section). The true power from this initialization comes from dynamic port configuration. Once the desired sensors are programmed into the low-level framework, they are not assigned a port base or a set of pins to communicate on by default. Instead, this data is contained within the high-level initialization process and the Tiva initializes the corresponding pins for a sensor based on the desired pin map from the master computer. This means the user is able to immediately switch the pins used by a particular sensor from just interacting with the high-level computer and not needing to re-deploy an entirely new pin initialization sequence to interface with the sensor. This is part of the strength of this framework. All of the sensor interfaces are already pre-programmed and the user just needs to interface with the high-level to command the Tiva which pin map to initialize each sensor on; the underlying low-level framework an library takes care of the rest. There are also certain customization particular to each individual sensor. For example, Absolute Encoders will have a zero position and range of motion limits, while a Force Sensor will have an offset and slope value along with compression and tension force limits. These differences in sensors are handled by tailoring each initialization frame for each individual sensor. Hence, if a Tiva has five sensors, the master computer will send out five initialization frames tailored for each one. The only caveat is that the Tiva needs to know which initialization frame number corresponds to which sensor. This is addressed in the framework via the use of enumerations. During the initialization process, the Tiva examines the current initialization frame number being sent from the master, identifies the corresponding enumeration, and stores the data for that particular sensor accordingly. An example set of enumerations for initialization can be seen in Listing 3.4.

```cpp
enum TivaInitializationFrameType {
```

1 enum TivaInitializationFrameType

2 {

```
The framework also supports dynamic re-initialization at any point in time. This means that if the user wants to change a few values pertaining to a sensor’s pin map or a sensor’s limits, they can do so by editing these values in the high-level and by restarting the initialization procedure without the need for a power cycle of the microcontroller. This makes tuning the microcontroller much easier as all of the sensor arguments are stored on the high-level, can be edited at any point in time, and re-sent to the Tiva to view the changes in real time without needing to redeploy low-level code or power cycle. This was implemented using dynamic memory allocation on the Tiva. Upon initialization, the Tiva dynamically allocates all of the sensors due to the use of polymorphism. It then uses these sensors for controls processing under normal operation. However, if the master attempts to re-initialize the Tiva, all of the sensor structures are freed and get re-populated and allocated by the re-initialization sequence. After these sensors are re-initialized, they initialize their new pin map, set new constant value pertaining to the sensor’s processing of data, and then control procedure continues as normal with the updates values the user specified in the high-level interface.
This re-initialization can also be used to reset a Tiva which had its EStop triggered. When an EStop is triggered, it sets the tiva initialized flag to false as a safety feature. Hence the only way to reset this flag to continue normal operation is to fix the troubled sensor and re-do the initialization procedure to set this initialized flag back to true.

3.5 EtherCAT Communication

The framework and library includes support for the EtherCAT communication protocol for fast reliable communication to the high-level controller. EtherCAT stands for Ethernet for Control Automation Technology and is meant to specifically address the inefficiencies of Ethernet [25] [31]. These inefficiencies in Ethernet are caused by specific protocols such as IP and ARP which are not necessary in a static network on robotic systems; these traditional protocols mainly serve a purpose in dynamic networks (such as a home internet router) in which the number of devices on a network can change at any time. Since the majority of robotics systems have a static network, the EtherCAT communication protocol was chosen due to its minimalist approach to networking which can speed up the communication loop between the Tiva and the high-level computer.

The EtherCAT communication protocol was implemented into the framework through an external microprocessor responsible for maintaining this communication. The EasyCAT Pro was chosen as the EtherCAT interface because it offers an easy to use configuration tool and offers plug-and-play abilities with almost any EtherCAT master program. This board interfaces with the Tiva over the SSI communication protocol and uses the LAN9252 chip as the EtherCAT slave controller (ESC) [18]. An image of the EasyCAT Pro can be seen below in Figure 3.6.

The EtherCAT protocol works to guarantee real-time communication capabilities as the
master is the single entity capable of sending a communication frame and the slaves simply process the data on the fly and continue to move the frame downstream in a daisy chain manner. However, complications arise with this EtherCAT guarantee since the LAN9252 is a separate dedicated chip to process the EtherCAT protocol and is not part of the Tiva. Hence, while this real-time guarantee might be true for the LAN9252 on the EasyCAT board, it does not guarantee that the Tiva will see the data in real-time as it is a separate component which queries the LAN9252 for the latest information. If the Tiva is running slower than the master computer, the LAN9252 will still guarantee this real-time communication however, the rate at which the Tiva queries this board for the data will be slower and as a result, the Tiva will miss communication frames coming from the high-level computer. Moreover, if the Tiva does miss communication frames, the master has no way or knowing because it is only communicating directly with the EasyCAT Pro and not the Tiva board.

To address this problem, a counter value dubbed the master process ID (MPID) was added to the EtherCAT frame and is used on every Tiva Slave in the distributed EtherCAT network. For each communication data frame the master sends out (could be an initialization frame or control frame to command the microcontrollers), the master attaches an MPID to the frame and sends it to the Tiva. When the Tiva is processing this data, it then copies the MPID
received from the master and echos it back to signify that the data corresponding to the MPID the master sent out has been processed. Additionally, if the Tiva receives the same MPID as the one it just previously processed, the Tiva will reject the input as it already responded to the master’s request for that particular MPID (although the input is rejected, the Tiva will still respond to the master with updated values). The MPID is set to count up to a value of 200 and is then reset down to a value of 1. This gives the master computer a way of understanding how fast the Tiva is updating because the MPID is edited by the Tiva and not the LAN9252. Various protocols were implemented around the use of the MPID and serve different purposes for communication techniques.

3.5.1 Tight-Chaining

There are many different techniques for communicating to low-level microcontrollers. A common approach is to have the master computer send out a new frame with an updated MPID every time it runs its high-level control loop. However, this approach can have inherit disadvantages especially due to the separation of the LAN9252 from the Tiva board. These disadvantages come from the fact that the Tiva might not be able to keep up with the master computer and communication frames can be skipped. A occurrence of a skipped frame once in a while is no problem from a controls standpoint. However, for initialization procedures in which every frame contains valuable data on how the Tiva should function, a skipped frame can be catastrophic.

A protocol was put together on top of IHMC’s EtherCAT communication to address this problem. The Tight-Chaining protocol utilizes the MPID to ensure that the Tiva is processing each and every frame that the master computer is sending. This protocol is used for the initialization procedure in the framework and is implemented in the high-level controller
communication algorithm. The master computer will first send out a frame with an initial MPID. The Tiva will then receive that MPID and process the request of the master computer as normal. The protocol differs from typical communication in that the master computer waits for the Tiva to echo the MPID before it sends the next frame. This guarantees that the master waits for the Tiva as it is restricted on when it can send out another communication frame until the Tiva is done processing. In order for this to occur, the master must send out at least two communication frames (in the best case) to the Tiva in order to determine if the Tiva updated. The first frame is to initially send the data to the Tiva and will have a particular MPID attached to it. The second frame will have the same MPID attached to it (because the master computer must wait for a response to update the MPID) and is to collect the data from the Tiva.

From the Tiva’s perspective, it receives the first frame which is a new MPID, begins processing, and sends no data back to the master (because it is the initial frame). On the second frame, the Tiva finishes its processing and when the master sends out the second frame, the Tiva pulls it, realizes the MPID is the same as the first and rejects the input, but publishes the result from the first frame with the echoed MPID and sends it to the master. The cycle continues from there. The protocol can be seen in greater detail in Figure 3.7 for a simple case with one Tiva slave component.

Figure 3.7: Tight Chaining Implementation
This protocol has its advantages and disadvantages depending on the application it is being used in. For initialization purposes, this protocol has major benefits as the master computer is constrained to wait on the Tiva to echo back the MPID before it can send out another initialization frame. However, because it requires the master computer to send out two frames per updated MPID (the first to send the MPID to the Tiva, the second to get its response), the communication rate between the master and the Tiva is effectively cut in half. Hence, while this protocol is ideal for initialization purposes as it guarantees every frame will be read by the Tiva, it is not ideal for controls processes as the update rate of the controller will take a dramatic hit.

3.5.2 Loose-Chaining

Another protocol was designed specifically to manage control signals from the master. The protocol of Loose-Chaining was developed so that the control loop can update as fast as possible with occasional skipped frames. It was designed so that the master could also monitor how many frames were skipped by the Tiva. This protocol works by allowing the master computer to be one MPID ahead of the Tiva which accounts for the downside of Tight-Chaining. In Tight-Chaining, the master must wait for the response of a single MPID and thus it takes two communication cycles. However, Loose-Chaining works by initially sending the Tiva an MPID, then immediately sending the Tiva an incremented and new MPID. The point of doing so is that once the Tiva is done processing the first MPID, it sends it back to the master and then can immediately begin processing the second MPID. Once the master receives the MPID which is equivalent to one minus the current value of the MPID which is being sent, the Tiva has been updated and the master then increments the MPID and sends it again to stay one step ahead of the Tiva. This cycle then continues from there. It is important to note that frames can be skipped using Loose-Chaining. By the
master always being one step ahead of the Tiva, it is assuming the Tiva is able to keep up at the same speed. If this is not the case, the Tiva will appear to be 2 MPID values behind and the master will register this as a skipped frame. Under normal circumstances, the skipped frames are rare but do occur. Further analysis on the amount of frames skipped for this implementation was tested and recorded to test the protocol. The Loose-Chaining protocol can be seen in more detail in Figure 3.8 for a simple case with one Tiva slave components.

![Figure 3.8: Loose Chaining Implementation](image.png)

### 3.5.3 Communication codes

Various communication codes were programmed into the EtherCAT frame to command the Tiva to processes and interpret the data within the frame in a certain way. This allows the Tiva to treat initialization signals differently from control signals and extract the appropriate data based on which type of signal the master computer is sending it. To assist with this, the master computer will send a Signal Byte at the beginning of each EtherCAT frame which the Tiva can use to understand how to parse the incoming data.

From a low-level code point of view, this was accomplished using a union of various EtherCAT frame structures for each of the possible different signals which the master computer could send. By having a union of structures, all of the EtherCAT frames occupy the same memory.
in the low-level code within the union and each individual struct can be called based on the value of the signal from the master computer. For example, if the signal from master is a Control Signal, it will use the corresponding structure within the union to handle the data extraction for the received control signal. An example of how to use this EtherCAT union structure can be seen Listing 3.5 and in Listing 3.6.

```c
union EtherCATFrames_IN
{
    ControlSignalEtherCATFrame_IN controlSignalFrame;
    HaltSignalEtherCATFrame_IN haltSignalFrame;
    InitSignalHeaderEtherCATFrame_IN initSignalHeader;
    InitSignal0EtherCATFrame_IN initSignal0Frame;
    InitSignal1EtherCATFrame_IN initSignal1Frame;
    InitSignal2EtherCATFrame_IN initSignal2Frame;
    InitSignal3EtherCATFrame_IN initSignal3Frame;
    InitSignal4EtherCATFrame_IN initSignal4Frame;
    InitSignal5EtherCATFrame_IN initSignal5Frame;
    InitSignal6EtherCATFrame_IN initSignal6Frame;
    InitSignal7EtherCATFrame_IN initSignal7Frame;
    uint8_t rawBytes[BYTE_NUM];
};
typedef union EtherCATFrames_IN EtherCATFrames_IN;
```

Listing 3.5: EtherCAT Union of structures

```c
tiva->signalFromMaster =
    tiva->easyCAT.etherCATInputFrames.rawBytes[SIGNAL_INDEX];
if(tiva->signalFromMaster == CONTROL_SIGNAL)
```

33
Listing 3.6: Brief EtherCAT Union of structures implementation example

These respective listings clearly show how the signal from master allows the Tiva to correctly extract the underlying data relative to the EtherCAT frame sent. In the example provided, the etherCATInputFrames union serves as a middle-ground for accessing the correct structure based on the type of data the master computer is sending. If this kind of check was not implemented, the Tiva would have no way of knowing which frame the master would be sending and it could parse the data incorrectly. If the Tiva processes an incoming initialization signal from the master as a control signal, the byte order will not properly align as different data types of various sizes are stored in each frame (as will be seen in the next section). Hence this signal byte is important as the Tiva will not risk misinterpreting data and will thus avoid catastrophic failure. There are four main signals which are programmed into the framework, however, the user is able to add more should they choose. The following are the four pre-programmed signals:
• **INITIALIZATION**: Sends initialization data the microcontroller needs before it can start processing.

• **CONTROL**: Commands the Low-Level Controllers to move to a desired location. The Low-Level Controllers then sends the master the most updated sensor values.

• **IDLE**: Puts the Low-Level Controllers in standby.

• **NO CONNECTION**: When the Tiva is initialized but not connected to the master computer.

• **HALT**: Triggers the low-level controller’s emergency-stop and turns motors off.

Each of these signals and their respective responses are pre-programmed into the low-level framework. The end user is encouraged to design their application around these given signals as it greatly simplifies the organization of the possible Tiva States.

### 3.5.4 Sample EtherCAT Frames

The EtherCAT frame is the physical data frame which gets transferred from the master computer to the Tiva and visa verse. Each Tiva has two frames which interact with the master computer, one frame is for receiving data and the other frame is for sending data. These input and output frames are stored on the Tiva in the form of a union of structs and an example of an input frame union can be seen in Listing 3.5. It is important to note that each of these input and output frames are different in structure yet occupy the same amount of data total. This framework was designed to have a default frame size of 128 bytes for input and 128 bytes for output. This is the maximum amount of bytes which can be allocated to the EasyCAT board, and is one of the default configurations of the board which can be flashed with the EasyCAT configure tool. Sample EtherCAT frames for an input and output control signal frame can be seen in Table 3.1 and Table 3.2.
### Table 3.1: CONTROL Signal Data Tiva-To-Master.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Element</th>
<th>Byte Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>Signal To Master</td>
<td>1</td>
</tr>
<tr>
<td>uint8_t</td>
<td>Master Process ID (echo)</td>
<td>1</td>
</tr>
<tr>
<td>float[]</td>
<td>Force (Actuator 0 and 1)</td>
<td>8 (4 each)</td>
</tr>
<tr>
<td>uint32_t[]</td>
<td>Motor Encoder Raw Position (Actuator 0 and 1)</td>
<td>8 (4 each)</td>
</tr>
<tr>
<td>int8_t[]</td>
<td>Motor Encoder Direction (Actuator 0 and 1)</td>
<td>2 (1 each)</td>
</tr>
<tr>
<td>int32_t[]</td>
<td>Motor Encoder Raw Velocity (Actuator 0 and 1)</td>
<td>8 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Motor Current (Actuator 0 and 1)</td>
<td>8 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Joint Angle (0 and 1)</td>
<td>8 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Accelerometer (X, Y, and Z)</td>
<td>12 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Gyroscope (X, Y, and Z)</td>
<td>12 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Magnetometer (X, Y, and Z)</td>
<td>12 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Force-Torque Force (X, Y, and Z)</td>
<td>12 (4 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>Force-Torque Torque (X, Y, and Z)</td>
<td>12 (4 each)</td>
</tr>
<tr>
<td>uint8_t[]</td>
<td>Remaining Bytes</td>
<td>24</td>
</tr>
</tbody>
</table>

### Table 3.2: CONTROL Signal Data Master-To-Tiva.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Element</th>
<th>Byte Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t</td>
<td>Signal From Master</td>
<td>1</td>
</tr>
<tr>
<td>uint8_t</td>
<td>Master Process ID</td>
<td>1</td>
</tr>
<tr>
<td>uint8_t[]</td>
<td>Direction (Actuator 0 and 1)</td>
<td>2 (1 each)</td>
</tr>
<tr>
<td>float[]</td>
<td>PWM (Actuator 0 and 1)</td>
<td>8 (4 each)</td>
</tr>
<tr>
<td>uint8_t[]</td>
<td>Remaining Bytes</td>
<td>116</td>
</tr>
</tbody>
</table>
3.6 LED Debug interface

The framework comes pre-built with various LED codes to help understand the current status of the Tiva. Similar to how the framework allows for dynamic initialization of the sensor ports and pins, the pins which the LED functions on can also be dynamically configured from the master computer. The LED used for this debugging is an RGB three pin input standard LED and blinks or lights up various colors to signal to the user the current state of the microcontroller. These LED codes are meant to make the internal state of the microcontroller known and take out ambiguity related to its internal state. These LED codes are also extremely useful for understanding error codes which the Tiva experiences during processing. They provide feedback on exactly what went wrong and can direct the user in the correct direction as to how to fix this underlying issue. A summary of all of the LED color codes pertaining to possible errors on the Tiva can be seen below in Table 3.3.

<table>
<thead>
<tr>
<th>LED Color</th>
<th>LED Signal</th>
<th>Error Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>Single Blink, Pause</td>
<td>Pre-initialization error</td>
<td>Tiva Communication to the EasyCAT board timed-out</td>
</tr>
<tr>
<td>Red</td>
<td>Double Blink, Pause</td>
<td>Pre-Initialization error</td>
<td>Byte Order on EasyCAT board incorrect</td>
</tr>
<tr>
<td>Red</td>
<td>Triple Blink, Pause</td>
<td>Pre-Initialization error</td>
<td>Internal Timeout on EasyCAT board</td>
</tr>
<tr>
<td>Red</td>
<td>Quadruple Blink, Pause</td>
<td>Pre-Initialization error</td>
<td>Unknown Error on EasyCAT board</td>
</tr>
<tr>
<td>Red</td>
<td>Solid</td>
<td>Control Loop Error</td>
<td>EStop Triggered, check halt code for more information. Or master sent halt signal to Tiva</td>
</tr>
</tbody>
</table>

Table 3.3: Error LED Blink codes

It is important to note that the majority of these error codes listed can occur in a Pre-initialization stage. These pre-initialization problems can arise when the EasyCAT board cannot be initialized on a Tiva and thus cannot receive any data from the master computer.
on how to initialize itself. A key issue arises related to sending an LED signal during pre-
initialization, because the LED pins themselves are sent to the Tiva during the initialization
process. Hence, if initialization does not occur from one of these errors, the Tiva does not
know which pins to use as the debug LEDs. A solution to this problem was provided by
adding a low-level System LED which is only initialized when a problem arises with the
EasyCAT board. The user can select this System LED to be any pin, but it is recommended
to be set to the red pin of the intended installed LED pin to stay in line with the framework.

The other LED codes programmed into the framework depend on the state which the Tiva
is in. These general LED codes can be seen in Table 3.4. As mentioned previously, these
color codes per each Tiva state helps remove the ambiguity regarding their internal state of
the microcontroller. In the framework, not all color combinations are used to signal the Tiva
status, however, the user is able to easily change the colors used or add more color codes as
they see fit.

<table>
<thead>
<tr>
<th>LED Color</th>
<th>LED Signal</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Color</td>
<td>N/A</td>
<td>Tiva waiting to be initialized from the master computer</td>
</tr>
<tr>
<td>Red</td>
<td>Continuous Blink</td>
<td>Tiva Initialized, not connected to the master computer</td>
</tr>
<tr>
<td>Blue</td>
<td>Solid</td>
<td>Tiva Receiving and Processing Control Signals from the master computer</td>
</tr>
<tr>
<td>Purple</td>
<td>Solid</td>
<td>Tiva Receiving Idle Signal From Master</td>
</tr>
</tbody>
</table>

Table 3.4: General LED Blink Codes
Chapter 4

Interface with High-Level

As previously stated, the true power of this framework and library comes from the structure, it provides the ability to dynamically use it for a wide range of applications, and the ability to easily interface with a high-level computer. The framework is designed such that the high-level computer can make significant changes to the functionality of the Tiva without the need to redeploy code to the board. While this is an extremely usefully feature, it is only one component of many which were implemented into the high-level computer. The hierarchical organization and abstraction of the communication structure provides an easy way for the user to extend the provided code for their particular implementation.

4.1 Structure

Similar to how the low-level Tiva library implemented contains its own structural features for simplicity and abstraction, the high-level code aims to do the same so it can be simple for the end-user. The high-level structure mirrors that of the Tiva microcontroller structure. The master computer is responsible for proper initialization of the Tiva, commanding desired motions, processing halt-signals from the virtual EStop, and providing a simple interface for data visualization.
4.1.1 Initialization

The master computer is designed to have knowledge of all the sensors which are used for a respective Tiva and thus knows how to properly initialize it. Initialization files in a JSON format are stored on the master computer for each Tiva. During startup, the master parses each of these initialization files, constructs corresponding EtherCAT frames for each sensor, and sends them to the Tiva one at a time using the Tight-Chaining protocol. The JSON file is structured so that each sensor contains the communication data and the brand of the sensor for dynamic initialization and polymorphism purposes. Additional information regarding initialization is then particular to each individual sensor (such as joint limits for an absolute encoder for example). An example of how the JSON initialization file is structured can be seen in Listing 4.1.

```
...  
"ForceSensor1":
{
  "ForceSensorBrand": "Futek",
  "ADC Communication":
  {
    "ADCBase": 0,
    "ADC Sequence Number": 0,
    "ADC Port Base": "E",
    "ADC Pin Number": 5
  },
  "Offset": 1563.45,
  "Slope": 1.0563,
  "Force Upper Limit Newtons": 4000.0,
  "Force Lower Limit Newtons": -4000.0,
```
The master computer runs a validity check on this initialization data to make sure that all of the values are appropriate before sending it to the Tiva. The master computer has knowledge about all of the types of sensors programmed into the framework and is able to convert the sensor brand names (Futek for example) into enumerated sensor values for each sensor category. The Tiva has knowledge of this enumeration and is thus able to receive this enumerated value from the master computer and initialize the correct sensor.

The power of dynamic re-initialization can be seen here as well. Say the user sets a force sensor limit in the JSON initialization file which is determined to be set too low during testing. Instead of having to stop testing, redeploy code to the Tiva and start over, the user simply edits the JSON file to have a more updated value, restarts the high-level code, and the new value will be send during the initialization startup process. This allows for convenient and fast testing of the Tiva’s respective peripherals.

Listing 4.1: Sample JSON Sensor Initialization File for a Tiva

```json
   "enable": true
},
"QEIEncoder1":
{
   "QEIBase": 1,
   "EncoderBrand": "Encoder MR Type ML",
   "SampleRate": 1000,
   "CountsPerRotation": 1000,
   "enable": true
},
..."
4.1.2 Control Loop Processing

Once initialization is completed, the master computer starts the main control loop. For the control loop, the master computer switches from a tight-chaining protocol to a loose-chaining protocol for efficiency regarding the robotic system’s update rate. During the control loop, the master computer can command a pwm signal, torque, position, velocity, or whichever input the user configured to send to the Tiva and also receive data back from the Tiva sensors in real-time. The IHMC software offers great visualization tools and hence it was chosen to be the backbone of the high-level implementation. The high-level code uses IHMC’s Simulation Construction Set Version 2 (SCS2) for its data visualization [30]. This visualization tool can be used for various purposes. For the research projects which implemented this framework, SCS2 was mainly used to plot the following:

- The change of sensor data over time
- The change of the state estimation values of the robot over time
- Torque and linear force tracking for controller estimation
- The rate at which the Master Process ID is updating to calculate the update rate of the Tiva
- The time at which a halt-code is triggered by a particular Tiva in the network.

SCS2 additionally provides a convenient way to save and organize the visualization data the user wants to collect. The use of SCS2 for data visualization along with dynamic initialization/re-initialization algorithms, and effective communication protocols makes the framework flexible, convenient, and intuitive for a user’s robotic application.
4.2 Code Structure

The library provides high-level program files based on the IHMC Open-Source Robotics software. These added program files are used to assist with the communication between the high-level computer and the low-level distributed microcontroller system. One of the these program files is the TivaSlave.java, which is an abstract class that must be extended for the user’s particular application. This abstract class has some helper functions to assist with the user’s development of their own communication method. The TivaSlave.java file has functions which handle initialization, storing and sending EtherCAT frames, missed frame checks, and helper functions pertaining to Tight-Chaining and Loose-Chaining. Additional helper functions were created to pack/deserialize the EtherCAT frame based on the data type to send/receive. While the TivaSlave.java class provides various helper functions for handing basic tasks, there are also abstract functions which the user must implement in their application. These helper functions for packing the EtherCAT frame and the remaining abstract methods the user must implement can be seen below in Listing 4.2 and Listing 4.3 respectively.

```java
public int PackRawFrame(int[] inputFrame, int startIndex, byte valueToPack)
{
    // the size of a byte is 1
    inputFrame[startIndex] = valueToPack;
    return startIndex + 1;
}

public int PackRawFrame(int[] inputFrame, int startIndex, short valueToPack)
{
    // the size of a short is 2 bytes
    // ... (remaining code)
}
```
Listing 4.2: Some of the TivaSlave helper functions to pack the EtherCAT frame

// abstract methods to be overridden by end user
public abstract void ReadInitFileForSetSlaveLocation();
public abstract byte GetCurrentInitializationFrameNumber();
public abstract void SendInitializationSignal();
public abstract void SendControlSignal(ControlSignalDataToSend controlSignalDataToSend);

public abstract void SendIdleSignal();

public abstract void DeserializeReceivedHaltSignalFrame(int[] input);

public abstract void DeserializeReceivedControlSignalFrame(int[] input);

public abstract void DeserializeReceivedInitializationFrame(int[] input);

public abstract void DeserializeReceivedIdleSignalFrame(int[] input);

public abstract InitializationData getInitData();

public abstract boolean wasInitFileRead();

public abstract DataFromTiva getDataFromTiva();

Listing 4.3: Abstract functions defined in TivaSlave.java which the user must implement for their application

This overarching abstract class also interacts with a separate java file which contains a list of all the pre-programmed sensors into the framework. AvailableTivaSlaveSensors.java is meant to house all possible sensors and provides mappings from a sensor brand name to an enumeration code. As previously stated, the master then can send this enumeration sensor code to the Tiva to initialize the proper sensors. As more users add more sensors to this framework, it is recommended that they also update these enumerations to allow the high-level computer to dynamically allocate the new sensors added. The user would also need to update the low-level code to add an additional enumeration value pertaining to the sensor added, and also add a condition which the newly added sensor can be initialized. An example of these enumerations for both the high-level computer and the low-level Tiva code can be seen below in Listings 4.4 and 4.5 respectively. It is recommended that should the user add an additionally sensor, they edit both these enumerations in the high-level and low-level code to support dynamic initialization and polymorphism.
public enum availableAbsoluteEncoderBrands
{
    GurleyEncoder((byte)0),
    OrbisEncoder((byte)1);

    private byte enumId;

    availableAbsoluteEncoderBrands(byte id) { enumId = id; }

    public byte getEnumId() { return enumId; }
}

Listing 4.4: High-Level: The enumeration of Absolute Encoders programmed into the framework

Joint joint;

// Construct and initialize the Joint Encoder based on the provided encoder
// brand enumeration value given from the master computer
if(encoderBrand == Gurley_AbsoluteEncoder)
    joint.encoder = GurleyEncoderConstruct(CommunicationBase, sampleRate);
else if(encoderBrand == Orbis_AbsoluteEncoder)
    joint.encoder = OrbisEncoderConstruct(CommunicationBase, sampleRate);

Listing 4.5: Low-Level: The enumeration of Absolute Encoders being used to dynamically allocate the corresponding encoder

Both the TivaSlave.java file and the availableTivaSlaveSensors.java file provide the user
with a base to start the development of their high-level communication while still staying in-line with the framework’s advantages. The user can take advantage of these provided files by extending the TivaSlave, utilizing the helper functions defined, and designing their implementation around the framework provided. It is recommended the user update the framework accordingly based on their particular application to continue support for the framework’s key features such as dynamic initialization and polymorphism. Such features allow for easy testing and implementation. Additionally, the contribution of each user to the open-source software community would greatly make this framework powerful.

4.3 Implementing for your application

Once the user extends the TivaSlave for their application, they will have to implement the abstract functions defined to complete their implementation. The main functions to implement pertain to sending and deserializing Initialization Frames and Control Signal Frames which are being sent to and from the Tiva during operation. The abstract functions defined are left up to the user to implement for their particular application. However, an example of how to implement these functions in a test setup was put together to experiment with the flexibility and usability of the framework from the user’s perspective.

The examples which will be described will cover recommendations on how to implement the EtherCAT frames. Before the explanations can be shown in detail, it is first important to understand how the packing of the EtherCAT frames works from both the high-level and low-level perspective. As described in previous sections, the low-level implements the EtherCAT input and output frames as a union of structs, and an example of the data inside one of these structs is seen in Tables 3.1 and 3.2. How these structures are defined, and the order in which the data elements are listed are important when extracting and sending data.
via EtherCAT. The code for the structure definition of the control signal from the Tiva to the master can be seen in Listing 4.6.

```c
struct __attribute__((__packed__)) ControlSignalEtherCATFrame_OUT {
    uint8_t signalToMaster;
    uint8_t masterProcessId;

    float actuator0ForceInNewtons;
    uint32_t actuator0MotorEncoderRawPosition;
    int8_t actuator0MotorEncoderDirection;
    int32_t actuator0MotorEncoderRawVelocity;

    float actuator1ForceInNewtons;
    uint32_t actuator1MotorEncoderRawPosition;
    int8_t actuator1MotorEncoderDirection;
    int32_t actuator1MotorEncoderRawVelocity;

    float joint0angleRadians;
    float joint1angleRadians;

    float Ax;
    float Ay;
    float Az;
    float Gx;
    float Gy;
    float Gz;
    float Mx;
```
float My;
float Mz;

float ftForceX;
float ftForceY;
float ftForceZ;
float ftTorqueX;
float ftTorqueY;
float ftTorqueZ;

uint32_t haltCodeToMaster;
uint8_t remainingBytes[28];

};

typedef struct ControlSignalEtherCATFrame_OUT ControlSignalEtherCATFrame_OUT;

Listing 4.6: Code for structure of control signal frame to send to master computer

This structure is one of many defined as a possible output frame from the Tiva to the master computer. These frames are parsed byte by byte starting from the top of the structure. Meaning the first byte listed inside the structure takes up the first byte index of the EtherCAT frame. In the example above, signalToMaster, masterProcessId, and actuator0ForceInNewtons would take up the EtherCAT frame index 0, 1, and 2-5 respectively. Hence, when the master computer is sending or receiving data from the Tiva, it must know which data elements are located at the corresponding indexes of the EtherCAT frame along with the size of these elements. This information can be obtained from looking at the low-level EtherCAT structure. Additionally, notice how in the above listing that the structure used the packed attribute. This tells the C compiler not to add any sort of byte padding
into the structure. This is critical when designing EtherCAT frames as padding can add byte gaps in the EtherCAT frame which the master cannot predict and thus could lead to serialization problems when trying to interpret the EtherCAT frame.

### 4.3.1 Initialization Frame Processing Example

The following code listings will demonstrate how to use the helper functions provided to send an initialization frame to the Tiva and parse its response. As mentioned previously, each initialization frame is meant to initialize either one sensor or a group of sensors. Thus the data for each initialization frame, along with the byte orientation and data type sizes, will be different. For each individual initialization frame, the master computer must know which data the Tiva is expecting and pack the frame accordingly. The master computer implements an initialization enumeration to fix and address this issue. An example of how to pack an initialization frame can be seen below in Listing 4.7.

```java
1 // joint 1
2
3 else if (currentInitializationFrameToSend == PandoraInitializationFrames.Joint1)
4 {
5     startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
6         TivaState.InitializationSignal.getEnumId());
7     startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex, (byte)
8         masterProcessId);
9     startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
10        currentInitializationFrameToSend.getEnumId());
11
12     if(initData.joint1.enable)
13         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
14            initInfo.joint1.typeId);
15         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
16            initInfo.joint1.length);
17         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
18            (short)initInfo.joint1.dataSize);
19     start = startIndex + 50;
20
21     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
22 
23         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
24            0);
24     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
25         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
26            (byte)0x00);
27 
28     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
29         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
30            (byte)0x01);
31     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
32         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
33            (byte)0x02);
34 
35     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
36         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
37            (byte)0x03);
38     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
39         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
40            (byte)0x04);
41 
42     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
43         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
44            (byte)0x05);
45     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
46         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
47            (byte)0x06);
48 
49     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
50         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
51            (byte)0x07);
52     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
53         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
54            (byte)0x08);
55 
56     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
57         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
58            (byte)0x09);
59     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
60         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
61            (byte)0x0A);
62 
63     if (initData.joint1.typeId == PandoraInitializationTypes.Instruction)
64         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
65            (byte)0x0B);
66     else if (initData.joint1.typeId == PandoraInitializationTypes.Matrix)
67         startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
68            (byte)0x0C);
69 
68
```
(byte) 1);
else
    startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
        (byte) 0);

startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
    initData.joint1.AbsoluteEncoderSSIBase);
startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
    initData.joint1.AbsoluteEncoderEncoderBrand);
startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
    initData.joint1.AbsoluteEncoderSampleRate);
if (initData.joint1.JointReverse)
    startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
        (byte) -1);
else
    startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
        (byte) 1);
startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
    initData.joint1.ZeroPositionValue);
startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
    initData.joint1.ForwardRangeOfMotion);
startIndex = PackRawFrame(rawInitializationFrameToSend, startIndex,
    initData.joint1.BackwardRangeOfMotion);
this.sendRawFrameToSlave(rawInitializationFrameToSend);
nextInitializationFrameToSend = PandoraInitializationFrames.RGBLEDs;

Listing 4.7: Example Implementation for sending initialization signals
Each of the elements get added to the EtherCAT frame using the PackRawFrame helper method as shown previously in Listing 4.2. These functions use overloading to support a variety of different input data to pack into the frame. After the data is packed, it returns the index position equivalent to the starting index to pack plus the size of the data which was packed. As a result, the startIndex variable can be continually updated and sent as a parameter to the PackRawFrame function until all of the data is packed. This makes the code to pack an EtherCAT frame significantly easier as all of the byte orientation and shifting is handled by these helper functions.

As for deserializing the data coming in from the Tiva, the process becomes a mirror image. For sending data to the Tiva, there are helper functions for packing the EtherCAT frame, similarly, there are helper functions provided for a user’s implementation to help deserialize the input EtherCAT frame data from the Tiva. An example user implementation of the deserialization process for a response from the Tiva regarding an initialization frame can be seen below in Listing 4.8. Additionally, the definition for the helper function used to deserialize the data can be seen in Listing 4.9.

```java
public void DeserializeReceivedInitializationFrame(int[] input) {
    int startIndex = 0;
    Object[] parsedData = new Object[2];

    // for command signal responce from Tiva
    parsedData = GetDataFromRawTivaFrame(input, startIndex,
                                          DataFromTivaDataTypes.BYTE);
    dataFromTiva.commandSignalResponceFromTiva.set(((Integer)
                                                    parsedData[0]).intValue());
    startIndex = (int) parsedData[1];
}
```
parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.BYTE);

dataFromTiva.masterProcessID.set(((Integer) parsedData[0]).intValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.BYTE);
numInitFramesTivaReceived = ((Integer)parsedData[0]).byteValue();
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.BYTE);
maxInitFramesToSendToTiva = ((Integer)parsedData[0]).byteValue();
}

Listing 4.8: Example Implementation for receiving initialization signals

public Object[] GetDataFromRawTivaFrame(int[] inputFrame, int startIndex,
    DataFromTivaDataTypes dataType)
{
    // a byte and char is a single byte long
    if(dataType == DataFromTivaDataTypes.BYTE || dataType ==
    DataFromTivaDataTypes.CHAR)
        return new Object[] {inputFrame[startIndex], startIndex + 1};
    // a short is 2 bytes long
    else if(dataType == DataFromTivaDataTypes.SHORT)
    {
Listing 4.9: Tiva Slave Deserialize helper function

The given implementation and helper function provide a clear and concise example of how the user can deserialize the incoming EtherCAT data frame provided by the Tiva for various
initialization frames. In this implementation, the user stores the signal response, the echoed MPID and other data pertaining to the Tiva’s processing of the initialization signals. The helper function greatly assists with the deserialization of the incoming EtherCAT data frame. The helper function operates by specifying a starting index and the size of the data to read from that index. The function then deserializes the raw data frame into the correct byte format and returns both the correct value and the starting index plus the size of the data which was extracted (similar to how it functions in the PackRawFrame function). Both examples given in the sending and receiving of initialization frames can be implemented by the user should they choose to do so. They are also free to implement these functions however they see fit to best apply to their application.

4.3.2 Control Frame Processing Example

Control Signal Processing for the EtherCAT frame functions similar to that of the initialization frame with one key difference. This key difference comes from the fact that a control signal sends the same data types to the Tiva. This is different from an initialization signal in which each frame contains a different structure and different number of bytes. However, with a control signal, the data types of the structure stay constant, their values are what change. To simplify the Control Signal Frame processing, a nested abstract class was added inside TivaSlave which is meant to be extended based on the user’s particular application. The abstract class ControlSignalDataToSend is declared as an empty class and should be extended and populated with the corresponding data fields to send the Tiva. An example of how to extend this class and pack a control signal frame using this class can be seen below in Listing 4.10 and Listing 4.11 respectively.

```java
public class ControlSignalDataToSendPandora extends ControlSignalDataToSend
{
```
public int direction0;
public float pwm0;
public int direction1;
public float pwm1;
}

Listing 4.10: Control Data To Send example implementation

@Override
public void SendControlSignal(ControlSignalDataToSend controlSignalDataToSend) {
    int[] rawControlSignalFrameToSend = new int[easyCATSlaveFrameLength];
    int startIndex = 0;
    ControlSignalDataToSendPandora controlSignalDataToSendPandora =
      (ControlSignalDataToSendPandora) controlSignalDataToSend;
    startIndex = PackRawFrame(rawControlSignalFrameToSend, startIndex, TivaState.ControlSignal.getEnumId());
    startIndex = PackRawFrame(rawControlSignalFrameToSend, startIndex, (byte) masterProcessId);
    startIndex = PackRawFrame(rawControlSignalFrameToSend, startIndex, controlSignalDataToSendPandora.direction0);
    startIndex = PackRawFrame(rawControlSignalFrameToSend, startIndex, controlSignalDataToSendPandora.pwm0);
    startIndex = PackRawFrame(rawControlSignalFrameToSend, startIndex, controlSignalDataToSendPandora.direction1);
    startIndex = PackRawFrame(rawControlSignalFrameToSend, startIndex, controlSignalDataToSendPandora.pwm1);
    this.sendRawFrameToSlave(rawControlSignalFrameToSend);
Listing 4.11: Packing Control Signal Frame example implementation

In the above example, the user is only sending a direction and a PWM signal for two motor controllers. The user can define this class and update the variables associated with it within the control loop during processing. The user then passes this class to the SendControlSignal method where it gets parsed correctly and sent to the Tiva for further processing and actuation of the motor controllers.

The other side of this example happens when the Tiva sends the master a control signal and the master deserializes it, the example code provided for this deserialization and processing on an incoming control signal from the Tiva can be seen below in Listing 4.12. Notice the correlation between this listing and Listing 4.6 described previously. It is clearly noticeable how each of the data fields within the low-level EtherCAT output frame correspond to the master computer’s parsing of the input control signal frame from the Tiva.

```java
public void DeserializeReceivedControlSignalFrame(int[] input)
{
    int startIndex = 0;
    Object[] parsedData = new Object[2];

    // for command signal response from Tiva
    parsedData = GetDataFromRawTivaFrame(input, startIndex,
                                          DataFromTivaDataTypes.BYTE);
    dataFromTiva.commandSignalResponseFromTiva.set(((Integer)
                                                    parsedData[0]).intValue());
    startIndex = (int) parsedData[1];
}
```

57
parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFrameTivaDataTypes.BYTE);
dataFromTiva.masterProcessID.set(((Integer) parsedData[0]).intValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFrameTivaDataTypes.FLOAT);
dataFromTiva.actuator0.forceSensor.forceInNewtons.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFrameTivaDataTypes.INT);
dataFromTiva.actuator0.qeiEncoder.encoderRawPosition.set(((Integer)
    parsedData[0]).intValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFrameTivaDataTypes.BYTE);
dataFromTiva.actuator0.qeiEncoder.encoderDirection.set(((Integer)
    parsedData[0]).intValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFrameTivaDataTypes.FLOAT);
dataFromTiva.actuator0.qeiEncoder.encoderRawVelocity.set(((Float)
parsedData[0]).floatValue();

startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
   DataFromTivaDataTypes.FLOAT);

dataFromTiva.actuator1.forceSensor.forceInNewtons.set(((Float)
   parsedData[0]).floatValue());

startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
   DataFromTivaDataTypes.INT);

dataFromTiva.actuator1.qeiEncoder.encoderRawPosition.set(((Integer)
   parsedData[0]).intValue());

startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
   DataFromTivaDataTypes.BYTE);

dataFromTiva.actuator1.qeiEncoder.encoderDirection.set(((Integer)
   parsedData[0]).intValue());

startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
   DataFromTivaDataTypes.FLOAT);

dataFromTiva.actuator1.qeiEncoder.encoderRawVelocity.set(((Float)
   parsedData[0]).floatValue());

startIndex = (int) parsedData[1];
parsedData = GetDataFromRawTivaFrame(input, startIndex, 
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.joint0.angleRadians.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, 
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.joint1.angleRadians.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, 
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Ax.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, 
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Ay.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, 
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Az.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];
dataFromTiva.imu.Gx.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Gy.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Gz.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Mx.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.My.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
    DataFromTivaDataTypes.FLOAT);
dataFromTiva.imu.Mz.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];
parsedData = GetDataFromRawTivaFrame(input, startIndex, DataTypes.FLOAT);
dataFromTiva.ftSensor.forceX.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, DataTypes.FLOAT);
dataFromTiva.ftSensor.forceY.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, DataTypes.FLOAT);
dataFromTiva.ftSensor.forceZ.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, DataTypes.FLOAT);
dataFromTiva.ftSensor.torqueX.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, DataTypes.FLOAT);
dataFromTiva.ftSensor.torqueY.set(((Float) parsedData[0]).floatValue());
startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex, DataTypes.FLOAT);
DataFromTivaDataTypes.FLOAT);

dataFromTiva.ftSensor.torqueZ.set(((Float) parsedData[0]).floatValue());

startIndex = (int) parsedData[1];

parsedData = GetDataFromRawTivaFrame(input, startIndex,
   DataFromTivaDataTypes.INT);

dataFromTiva.haltSignal.set(((Integer) parsedData[0]).intValue());
}

Listing 4.12: Storing the incoming Control Signal Frame from the Tiva

The examples described in this section should give the user some ideas on how to use the full power of the framework provided to implement their application. It is important to note that the user does not have to follow the same structure as the examples and are free to implement the abstract functions however they see fit. Additionally, it is recommended that if any changes are made by the user to the base framework files (TivaSlave.java and AvailableTivaSlaveSensors.java) that they commit and push their changes to the open-source project so other developers can take advantage for additional functional support for the framework.
Chapter 5

GUI Development

To broaden the target audience of this framework and library, a GUI was constructed to assist with the development processes of the microcontroller code and high-level code for the TI Tiva. This GUI would mainly appeal to engineers who are not too familiar with microcontroller code and would provide them with an easy way to generate the code they need. The GUI utilizes the library written for the TI Tiva and can construct both the microcontroller and high-level communication code for the end user. This code generator creates code which is in line with the framework and thus take advantage of all the previous framework features described. The interface developed runs as a web application and thus can be used on every computer system platform (Windows, Mac, Linux, etc.). The back-end code which services this GUI can be deployed to any local server and any user on the same network can access this GUI through a web browser such as Mozilla Firefox. It should be noted that this GUI does not offer total functionality of the Tiva board, however, it can be used as a starting point for the end user’s development.

5.1 GUI Front End

The GUI Front-End was developed using the P5.js library. P5.js is described by the developers as ”a JavaScript library for creative coding”, which has a lot of support for graphical features [7]. P5.js is mainly used in web development to make online video games or ani-
mations for a website [35]. The library is simple, light-weight, and offers graphical elements which can be easily extended.

The GUI offers a lot of features which make the experience for the end user simple. At the start page, the GUI features a modeled Tiva board with its respective pins, a development drop down bar, a section which lists the added sensors, and a code generation button for when the user has finished their design. An image of the GUI homepage can be seen in Figure 5.1.

![Image of GUI homepage](image)

**Figure 5.1:** The starting home page of the GUI (enlarged to show detail)

The homepage provides a simple interface for the user to interact with the virtual Tiva object. From the homepage, the user can then start developing for their application. They
are able to click on the development drop down button in which they will have the option to add/remove sensors or edit the presence of the zero ohm resistors on the Tiva. Additionally, the user can also click on the Tiva pins or zero ohm resistors to display information about them and what they can be used for. Images of the development window and the peripheral information window can be seen below in Figure 5.2 and Figure 5.3 respectively.

<table>
<thead>
<tr>
<th>Zero Ohm Resistors</th>
<th>Add Sensors</th>
<th>Remove Sensors</th>
</tr>
</thead>
</table>

Figure 5.2: The window which shows when the user clicks on the development drop down window (enlarged to show detail)
Once the user enters into the development state, they can begin configuring the Tiva and adding peripherals as they need for their application. The sensors which are pre-programmed into the GUI are the same sensors which were pre-programmed into the framework. Additional sensors can be added to the GUI with ease by simply editing the back-end code which the GUI runs on top of. In order to generate code from this GUI, the user must first specify which pins to configure the EasyCAT board and which pins to configure the RGB LEDs. Once these two requirements are met, the user can add any combination of sensors they would like their Tiva to process. Once the user is ready to add a sensor, the GUI will prompt them with a list of pins which the sensor can be configured on based on the underlying sensor protocol. The GUI can also detect pin conflicts with other sensors should they occur.
user try to add a sensor to a pin which is already being occupied. Once the user selects a pin map, if there are no pin conflicts, the GUI will prompt the user to select a color code for the respective sensor. Once this color code is selected, the Tiva’s pins light up with the respective sensor’s color. An example of the interface for adding a sensor, selecting a color code, and the output effect on the Tiva can be seen below in Figures 5.4, 5.5 and 5.6 respectively.

![Figure 5.4: The interface for adding a sensor. Here the user clicked Add Sensors -> Absolute Encoders -> Gurley Encoder (enlarged to show detail)](image)

![Figure 5.5: The interface for selecting a sensor color code (enlarged to show detail)](image)

Once the user has added the required peripherals and the necessary sensors needed for their implementation, they can click the Generate Code Button. Once this button is clicked, the front-end sends all of the sensor information and respective pin mappings to the back-end server. The server then processes the request and generates code for the Tiva and the high-level master computer to run a basic communication loop. The code which is generated is able to communicate between the master and the Tiva and is able to visualize all of the Tiva sensor readings in real-time in IHMC’s SCS2.
5.2 GUI Back End

The front-end of the GUI is able to provide a useful and enlightening interface to the user while the back-end does all of the processing required to sustain this pleasant interface. The back-end for the GUI was programmed in C++ and contains all of the information about the Tiva’s pins, peripherals, and sensors which can be added to the GUI. The reason why the back-end was chosen to be written in C++ was for code reuse and scalability purposes beyond the GUI. The back-end code features a basic model of the Tiva and hence is able to provide important information to the front-end. This model of the Tiva which was created...
in the back-end can be used for much more than just a GUI. With some tweaks made, this model can be used as a starting point to simulate hardware features on the Tiva in a virtual environment. C++ is the best choice for these simulations rather than a strictly web-based backend language such as TypeScript.

The back-end provides all of the information regarding the Tiva to the front-end which then parses and organizes the data to make it appealing to the user. When the user clicks on a peripheral button or tries to add a sensor, the front-end sends a query command to the back-end regarding the corresponding information it needs. These query commands are organized in the back-end through a mapping which maps a query command to a function pointer. The back-end can then just take the corresponding command, place it into the mapping and call the corresponding function to respond in a clean and easy manner. The code for how this mapping works can be seen below in Listing 5.1.

```
// initialization

functionMap["Peripheral-Info"] = &BackendProcessor::SendPeripheralInfoToFrontEnd;
functionMap["Sensor-Get"] = &BackendProcessor::SendSensorTypesToFrontEnd;
functionMap["Sensor-Communication-Base-Get"] =
    &BackendProcessor::SendSensorCommunicationBasesToFrontEnd;
functionMap["Generate-Code"] = &BackendProcessor::GenerateCode;
...

// function call when information from the front-end is received
if(validCommand)
    ((*this).*functionMap[command])(argument);
```

Listing 5.1: Use of mapping from command to function pointer in back-end

The back-end server is also responsible for generating the code for both the Tiva and the high-level master once the user is done with their development from the front-end. The front-
end sends the server a list of the sensors, sensor pins to initialize them on, and the sensor communication protocol to use. The back-end is able to take advantage of the framework to simplify the code generation process. Since the framework provides the majority of the structure regarding processing, the back-end only needs to generate code in a few select places such as the EtherCAT Frame structure, and the EStop conditions. The code generation uses a skeleton project which is then populated based on how the user configured the Tiva in the front-end. Once the back-end has completed generating the code, it gets sent to the front-end web application and can be downloaded as a zip file. This zip file contains the Tiva project which can be imported into Code Composer Studio, and the high-level java files which can be imported into the IHMC eclipse environment.
Chapter 6

Results

In order to measure the effectiveness of this framework constructed, it was experimented across every project within the Virginia Tech Terrestrial Robotics Engineering and Controls Lab (TREC Lab) [11]. The following TREC Lab projects continued to use the framework after experimenting with its underlying features: PANDORA [8], ForceBot [26] [3], and Reinforcement Learning Testing. Additionally, many different visualization graphs were tested and evaluated using IHMC’s SCS2 to present the data visualization in an appealing way to the end user to display key feedback about the underlying robotic system.

6.1 Data Visualization

Before the framework was deployed and tested on various projects through the TREC Lab, it was first tested on a standalone microcontroller system with different sensors. The point of doing so was to experiment with the communication protocols, ensure everything was working properly, and to test out different visualization tools in the IHMC software. An image of the Tiva test board which was used during the testing of this framework can be seen below in Figure 6.1.

The testing setup contains an EasyCAT Pro for communication between the Tiva and the master computer, a Gurley Encoder for measuring rotation, a Futex Force sensor for measure compression and tension forces, and an RGB LED for debugging purposes. The RGB LED
is built into a custom designed PCB which sits underneath the Tiva. The Gurley Encoder and the Futek Force Sensor are also connected to this PCB to interface with the Tiva. Once the test setup was constructed, various high-level visualization graphs were made using IHMC’s SCS2 to test the underlying features of the framework. Many graphs were made to test the update of the MPID, triggering of halt signals, and the general accuracy of sensor responses through the control process. Additionally, dynamic initialization features were tested by moving the Gurley Encoder and Futek Force Sensor to different pin maps on the PCB board, editing the JSON initialization file to a different pin base and re-initializing the Tiva to observe the changes. Dynamic initialization was thoroughly tested on the sensors used for this setup and was tested for the rest of the implemented sensors once deployed on
various projects throughout the TREC Lab. The visualization graphs which were used to evaluate this test setup before deploying to the TREC Lab projects can be seen below in Figures 6.2, 6.3, and 6.4.

Figure 6.2: A plot of the MPID over time from which the user can determine the true update rate of the Tiva. The first row is the MPID, the second row is time.

Figure 6.3: A plot of the sensor readings changing as they are being interacted with. The first row is the MPID, the second row is the Gurley Encoder reading, and the third row is the Futek Force Sensor reading.

The three Figures listed are just some basic examples of what IHMC’s SCS2 can be used for. The customizable data visualization combined with the low-level framework give immense power and control when it comes to low-level microcontroller development. The practicality of the framework will be seen in much greater detail on the robotic applications for which it was applied to in the Virginia Tech TREC Lab.
6.2 PANDORA

PANDORA is a fully 3D printed bipedal general-purpose humanoid robot designed by the Virginia Tech TREC Lab [8]. The entirely 3D printed design is unique in that it allows for quicker turnaround times. These quick turnaround times comes from the fact that all of the components for PANDORA can be 3D printed within two weeks. PANDORA was the first project within the TREC Lab to adopt this microcontroller framework. Originally, all of the individual microcontrollers were specifically programmed for a particular joint on PANDORA, this made debugging and development extremely difficult and time consuming as there were many different software projects to maintain for each joint.

Upon adoption of this framework, all of these individual projects were removed and instead, one project with various different configurations was used. The dynamic initialization procedure allowed each Tiva to be initialized with the required sensor for its respective joint simply by editing the high-level initialization file. Additionally, all of the appropriate Tiva sensor limits for each joint were able to be dynamically configured and tested without the need to re-deploy code (a procedure that was extremely time consuming before).

The easy implementation and interface between the Tiva and the master computer combined
with IHMC’s SCS2 truly made the development of PANDORA significantly easier. The developers were able to stop focusing on microcontroller specific errors and instead do the majority of their testing on controller related algorithms. Additionally, these controller algorithms were able to be tested using the SCS2 visualization interface. For PANDORA, the visualization interface involved trajectory tracking for force and torque control. This visualization interface can be seen in Figure 6.5.

Figure 6.5: The visualization setup in SCS2 made possible from the framework implemented
From the implementation of this framework, the work on PANDORA has been able to shift its focus from debugging tedious microcontroller issues to implementing the complex control algorithms required for this humanoid robot. An immense amount of time has been saved by the developers by implementing this framework and has thus caused the PANDORA project within the TREC Lab to thrive in popularity among the research community.

### 6.3 ForceBot

ForceBot aims to design and construct a fully immersive VR collaborative robot designed to simulate physical interactions with virtual objects via force feedback to a user’s hands and feet. This project is being conducted by the Virginia Tech TREC lab to provide more realistic virtual reality experience. ForceBot uses separate EtherCAT motor controllers to actuate the robotic systems, however, the designer of this project was able to use the framework advance with ForceBot’s sensory abilities.

ForceBot uses the OmronR88D1SN EtherCAT motor controller\(^4\) to actuate the system and to provide forces to the user’s feet when interacting with a virtual reality environment. This motor controller is used in various control system algorithms to make sure the user is experiencing the appropriate forces which inside a virtual environment. However, while ForceBot had a way of applying forces to the user’s feet, it did not have a way of measuring the force the user was applying on their own. This user’s applied force is critical for transparency and force feedback purposes in a virtual environment. For example, if the user is not interacting with an object in VR, the system should function as a zero impedance controller and thus needs some force measurement from the user to perform the appropriate calculation.

The developers of ForceBot chose to use a Tiva board with an FT Sensor to measure this force. Additionally, they chose to use the low-level framework constructed as they saw its
true abilities being used on PANDORA. A new challenge presented itself to the developers of ForceBot in that their EtherCAT chain had a diverse set of motor controllers along with the EasyCAT Pro all within the same network chain. This originally had never been tested or implemented, however, they were able to use the provided high-level files (particularly the TivaSlave.java file), extend them, and edit it for their EtherCAT network implementation.

The framework saved the ForceBot developers a tremendous amount of time as they did not have to experiment with the underlying microcontroller code. Instead, they had a near plug-and-play solution for the problem in which they were trying to solve. After the framework was tuned to run on ForceBot, the developers were able to read the force value from the FT Sensor, determine how fast the Tiva was running from the MPID, and apply the force reading to their controller algorithm to implement accurate force feedback to the user’s feet.

6.4 Reinforcement Learning

The TREC Lab recently started a new project aimed at using reinforcement learning algorithms to account for nonlinear elements within a control system. Typically sources of these nonlinear elements include but are not limited to: actuator backlash, motor saturation, and motor PWM deadzones. These nonlinear elements can easily cause a control system to go unstable for many robotic applications. Handling these nonlinear elements is such a common issue that it is an entire field of study within the realm of robotics. The goal of this project was to have an agent learn the nonlinearity features of a particular control system and adjust for these nonlinearities which are present so that the control algorithm remains stable during operation.

At the start of this project, the developers chose to use a simple pendulum design and have an agent trail on the motion tracking of the device. The pendulum featured a Futek Force
Sensor to measure the force applied to the pendulum and a Gurley Encoder to determine the angle at which the pendulum is swinging. Additionally, the virtual EStop features provided by the framework were critical so the agent could learn to operate in an appropriate and safe region.

The provided framework was chosen by the developers of this project as many researchers within the TREC Lab recommended it. Upon its implementation, it immediately offered a plug-and-play solution for the project’s initial testing. The developers took the base framework code which PANDORA was using, removed any unnecessary sensors which they did not need, and edited the initialization file to program in the correct sensor values. After about an hour of tuning the framework for their application, they were immediately able to control the pendulum and read the appropriate values from it. This project truly shows the full power of the framework. What would have taken the developer possibly days to implement from scratch was able to be done within a single hour because of the use of this framework.
Chapter 7

Conclusions

The framework which was implemented as part of this work was designed with the goal of saving time when working with microcontrollers. The framework was implemented on a TI Tiva board and has a variety of different sensors built into it so users can get a head start with the development for their application. Additionally, features such as dynamic initialization/re-initialization, virtual EStop capabilities, EtherCAT communication processing, and a high-level interface were all implemented to give the end user a significant starting point when designing for their application.

The web-based GUI developed also allows engineers without a computer technical background to take advantage of the framework’s features from a simple interface. After constructing their desired Tiva system, they can easily download the code, import it to their development environment, and begin testing their control system. Hence, the GUI offers a plug-and-play solution for many robotic applications as well and can be used as a starting point for the user’s application.

This framework has been implemented on a variety of different projects all across the TREC Lab at Virginia Tech and has continued to show promising results in regards to performance. PANDORA, ForceBot, and our new reinforcement learning project, all take full advantage of the framework’s feature to simplify the development for their application. Developers across all of these projects have utilized the firm structure of the framework: all of the lower-level details were already taken care of and thus more complicated algorithms were built off of
it for various robotic applications. This framework will soon be made open-source so that the rest of the robotics community all across the world will be able to implement it for their application and take advantage of the framework’s features just as the developers of the TREC Lab have done. Additionally, they may contribute to the open-source project by providing additional sensors to integrate or different ways of processing data for efficiency purposes. Once the framework is published as open-source, it has the potential to grow in popularity among the robotics community, and we could soon begin to see the framework being implemented on a wide variety of other robotics applications.
Bibliography


[29] Ihmrobotics. Ihmrobotics/ihmc-open-robotics-software: Robotics software featuring legged locomotion algorithms and a momentum-based controller core with optimization. supporting software for world-class robots including humanoids, running
birds, exoskeletons, mechs and more., URL https://github.com/ihmcrobotics/ihmc-open-robotics-software.


