NORTHWESTERN UNIVERSITY

Open-Source System for High-Level Control and User Interface to the RICNU Knee

A THESIS

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Mechanical Engineering

By

Alexey Revinski

EVANSTON, ILLINOIS

March 2018

Abstract

The work of this thesis aims to provide a low-power open-source high-level controller and an intuitive graphical interface to the RICNU Open Source Robotic Leg. Built on top of the FlexSEA motor control system developed at MIT, the RICNU high-level controller closes the planning control loop needed to complete the open-source electronics architecture on the Open Source Leg. The wireless nature of the user interface allows patient experimentation to leave the vicinity of a lab workbench and expand it into the rugged, real world scenarios.

Together, the high-level controller and the graphical user interface enable the researcher to monitor sensor data and state information on a hand-held device while directly interacting with patients; patient data is easily accessible via an on-board microSD card. Effective firmware architecture ensures real-time control over the prosthetic device, and an XML-based state machine configuration scheme provides an intuitive and simple way to experiment with different high-level state machines without embedded firmware intervention. The flexibility of the system has been validated using a custom finite state machine on a test setup simulating a lower-limb prosthesis.

Thesis Supervisor: Levi J. Hargrove, Ph.D.
Title: Associate Professor of Biomedical Engineering

Table of Contents

List of Figures

List of Tables

Table of Abbreviations

Abbr.	Meaning
ADC	Analog-to-Digital Converter
ANSI	American National Standards Institute
API	Application Programming Interface
ARM	Advanced RISC Machine
BBM	Break-Before-Make (switch type)
BGA	Ball Grid Array (IC package)
BGAPI	Bluegiga Application Programming Interface
BLDC	Brushless DC (motor)
BLE	Bluetooth Low Energy, also known as Bluetooth Smart
BOM	Bill of Materials
BR	Basic Rate (Bluetooth standard)
BSR	Bluetooth Smart Ready
CAN	Controller Area Network (communication standard and protocol)
COM	Communications (port)
CPU	Central Processing Unit
CS	Chip Select (SPI communication line)
DAC	Digital-to-Analog Converter
DCFSM	Device Control Finite State Machine
DFU	Device Firmware Upgrade
DIO	Digital Input/Output
DMA	Direct Memory Access (microcontroller peripheral)
DMIPS	Dhrystone Million Instructions Per Second (computational benchmark unit)
DOF	Degree of Freedom
DOM	Document Object Model
DPDT	Dual Pole Dual Throw (switch type)
DSP	Digital Signal Processing
EDR	Enhanced Data Rate (Bluetooth standard)
EMI	Electromagnetic Interference
ESD	Electrostatic Discharge
FAT	File Allocation Table (file system format); FAT32 – 32-bit FAT, exFAT - extended FAT
FCC	Federal Communications Commission
FIFO	First-in-First-out (software or hardware structure)
FlexSEA	Flexible Scalable Electronics Architecture
FM	Frequency Modulation (radio communication)
FPC	Flexible Printed Circuit (cable type)
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine

Abbr.	Meaning
FTDI	Future Technology Devices International
GAP	Generic Access Profile
GATT	Generic Attribute (Profile)
GCC	GNU Compiler Collection
GEPO	Git Repo
GNU	"GNU is Not Unix!" (free operating system)
GUI	Graphical User Interface
HSI	High-Speed Internal (oscillator)
HS	High Speed (Bluetooth Specification v3.0)
I/O	Input/Output
I2C	Inter-Integrated Circuit (communication standard and protocol)
IAP	In-Application Programming
IC	Integrated Circuit (chip)
ICP	In-Circuit Programming
IEC	International Electrotechnical Commission
IMU	Inertial Measurement Unit
JSON	JavaScript Object Notation (format, file)
JTAG	Joint Test Action Group (debugging interface)
LDO	Low-Dropout Regulator
LED	Light-Emitting Diode
LQFP	Low-profile Quad Flat Package (IC package)
MATLAB	Matrix Laboratory (data processing software)
MCU	Microcontroller Unit
MISO	Master-in-Slave-out (SPI communication line)
MIT	Massachusetts Institute of Technology
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MOSI	Master-out-Slave-in (SPI communication line)
NAND	Not AND (type of memory architecture)
NOR	Not OR (type of memory architecture)
NSS	Slave Select (SPI communication line)
NVIC	Nested Vector Interrupt Controller (ARM Cortex M interrupt controller)
OS	Operating System
OSI	Open Source Initiative
OTG	On-the-Go (USB standard)
PC	Personal Computer
PCB	Printed Circuit Board
PDR	Power-down Reset
PMUX	Power Multiplexer
POR	Power-on Reset
PPTC	Polymeric Positive Temperature Coefficient (fuse type)

Abbr.	Meaning
QFPN	Quad Flat No-leads Package (IC package)
RAM	Random Access Memory
RF	Radio Frequency
RICNU	Rehabilitation Institute of Chicago and Northwestern University (Open Source Leg)
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory
RS-485	Recommended Standard 485 - serial communication specification
RTC	Real-Time Clock (microcontroller peripheral)
RX	Receive
SCK	Serial Clock (SPI communication line)
SD	Secure Digital (card, protocol)
SDHC	High Capacity SD Card
SDIO	Secure Digital Input/Output
SDSC	Standard Capacity SD Card
SDXC	eXtended Capacity SD Card
SIG	Special Interest Group
SMB	System Management Bus (communication standard and protocol)
SMPS	Switch-Mode Power Supply
SoC	System-on-a-Chip
SPDT	Single Pole Dual Throw (switch type)
SPI	Serial Peripheral Interface
SPP	Serial Port Profile (Bluetooth profile)
SRALab	Shirley Ryan AbilityLab, formerly RIC
SRAM	Static Random-Access Memory
ST	STMicroelectronics
SWCLK	Serial Wire Clock (SWD communication line)
SWD	Serial Wire Debug (interface)
SWDIO	Serial Wire Data Input/Output (SWD communication line)
SWO	Serial Wire Output (SWD communication line)
TC	Transfer Complete (DMA interrupt)
TVS	Transient Voltage Suppression (diode type)
TX	Transmit
UART	Universal Asynchronous Receive Transmit (communication standard)
UHS	Ultra-High Speed (SD Card speed class)
USART	Universal Synchronous/Asynchronous Receive Transmit (communication standard)
USB	Universal Serial Bus (communication standard and protocol)
UUID	Universal Unique Identifier
WiMAX	Worldwide Interoperability for Microwave Access
XML	eXtensible Markup Language
XSD	XML Schema Definition (file format)

1 Introduction

Until recently, wearable robotic devices have been available to patients with disabilities only through established third-party companies. Vital technologies continue to be incorporated into expensive, custom devices, and patients in need of rehabilitation are often faced with an unpleasant choice – to pay "an arm and a leg" for an arm or a leg, or to forego the essential rehabilitation.

Recent developments in rapid prototyping technologies and the rising popularity of the Open Source Initiative (OSI) have partly alleviated this problem. Nowadays, there are a variety of free and open-source designs are made by companies and non-profits [1] [2] [3], skilled individuals [4] [5] and passionate enthusiasts. Open-source designs made by research labs mostly focus on upper-limb devices [6] [7].

One of the great benefits of this OSI trend is that it is now much easier for patients with disabilities to find and build low-cost alternatives to proprietary devices by themselves. But, some of these devices are made by individuals and groups lacking clear understanding of human movement and locomotion, which partly defeats the purpose of these devices as rehabilitative and assistive tools. Moreover, there is a clear disconnect between the work done by research labs around the world and the abovementioned "do it yourself" trend among the patient community. Each laboratory attempts to build its own novel wearable robotic device, then conducts a set of studies on the said custom device, and, perhaps, embarks on a long-term process of incorporating the invented technology into yet another highly functional, but unaffordable device [8] [9]. The research done on each of these custom devices is often not transferable to the field in general, because each research device has its own custom mechanical hardware, electrical hardware, and control software. Year after year, research labs build and test their own custom devices with no clear cross- standardization in sight, and advancements in optimal control algorithms for upper- and lower-limb prosthetic devices are stunted by the constant re-invention of wearable robots.

1.1 OSI in Wearable Robotics Research

The answer to this problem may lie, again, with the Open Source Initiative. Realistically, international standardization of wearable robotic devices in research settings may never be achieved, but certain strides towards that goal may be accomplished by sharing work between research laboratories, as foreign as it may sound. Sharing of pre-published research is close to impossible, but standardization of prosthetic systems *used* in this research is quite feasible. This would allow for greater transferability of research between multiple labs and promote quicker advancements in control algorithms for wearable robotics.

On the lower-limb front, this effort has been spearheaded by teams at Massachusetts Institute of Technology, Shirley Ryan AbilityLab (formerly Rehabilitation Institute of Chicago) in collaboration with Northwestern University, and University of Michigan. The effort entails the development of a full lower-limb prosthetic system openly accessible to research laboratories; the system consists of a low-cost and low-weight two-degree of freedom prosthetic leg and a scalable control electronics architecture.

1.2 Control of Robotic Leg Prostheses

Because human locomotion is a complicated and dynamic process, control systems hosted on wearable lower-limb prosthetic devices employ multiple control algorithms that assert low-, mid-, and high- level control algorithms over these systems.

Low-level control algorithms interface with the actuator components directly. They usually control the current passed through the actuating element by comparing a sensed the angular position of a motor shaft, or torque applied by the actuator to a desired value. They simply attempt to force their respective actuator to assume a certain desired state. Mid-level controllers provide the reference trajectories for the low-level controllers to maintain. Finally, high-level controllers entail state planning algorithms and select between multiple mid-level controllers based on the high-level state of the system.

For high-level control, wearable robotic devices most often employ finite state machines (FSM) as a means of controlling their behavior at different pre-defined phases of motion [10]. Lower-limb state machines usually subdivide the human gait cycle into four phases: stance, heel-off, swing, and heel-strike, along with other synonymous definitions [11]. Each state is linked to other states using event-based transitions. For example, a prosthetic leg may assume a "stance" state if the load on the vertical axis of an attached load cell reaches a certain minimum threshold and the leg is currently in the heel-strike phase.

Various controllers define their own triggers for transitioning between the different states and control their respective hardware systems using different low-level control loops. Although the low-level control of a wearable robotic system has not been the primary focus of this thesis, a FSM implementation was designed in the course of this project using various impedance states.

In the rest of the document, to avoid confusion of the device control state machine with embedded system state machines, any FSM that is used to modify behavior of the prosthetic device will be named a "device control finite state machine", or DCFSM.

1.3 MIT Flexible Scalable Electronics Architecture

The electronics control system employed by the Open-Source Leg is the Flexible Scalable Electronics Architecture (FlexSEA) developed at MIT. It features two electronic control boards that are assigned specific control functions in the overall prosthetic system, in step with the concepts discussed above [12]. The different components are shown in Figure 1.

The FlexSEA Execute board takes care of all of the low-level control functions, and effectively closes the low-level motor control loop on board. The device is able to apply position, current, and impedance control to a multi-commutator BLDC motor. It can interface with various external sensors and has a multitude of on-board sensors, making it a great all-in-one low-level controller.

The FlexSEA Manage board is a much more flexible component in the system. According to the original developer, it is able to take on a high-level control position in addition to taking care of mid-level control tasks. However, it lacks a good user interface and API to act in this role, but it can be an effective data processor; it can be used to route sensor data gathered on FlexSEA Execute upstream to another controller that is specifically assigned a high-level control role.

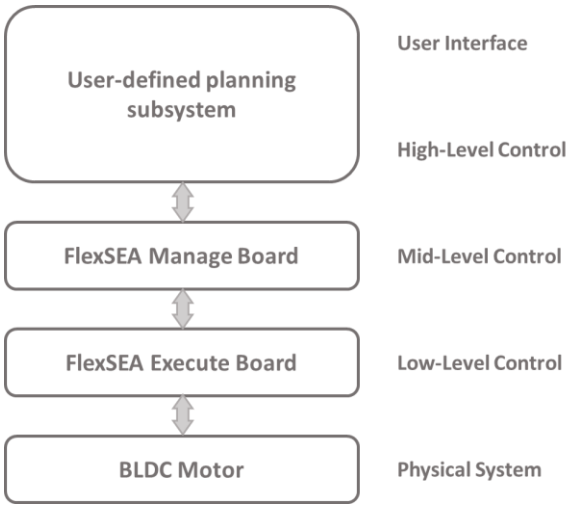


Figure 1: FlexSEA Architecture

For the planning level role, the original project employed a BeagleBone Black embedded computer, and the system has since shifted to using a desktop computer GUI interface instead. However, neither of these implementations of the planning component is sufficient for direct patient experimentation. Using an embedded computer on a battery-powered prosthetic device is very energy-inefficient. On the other front, the desktop computer GUI is only useful in a workbench setting, where the prosthetic device can be tethered to the computer. That leaves the system with no research-friendly high-level control component.

1.4 High-Level Control Gap

Keeping in mind the above shortcomings of the FlexSEA system, it is clear that the system needs a high-level control subsystem that incorporates the following essential components:

1) **Energy-efficient high-level control board:** To use the prosthetic device outside of the vicinity of a lab bench, a wireless but energy-efficient solution is required. To maintain real-time control performance, the high-level control component needs to be a part of the physical hardware of the

prosthetic device. The firmware of the controller needs to be compatible with and abstract away the intricacies of the FlexSEA architecture.

2) **Intuitive wireless graphical interface**: Researchers need to interface with patients directly while also monitoring the state of the prosthetic system. There is a need for an intuitive wireless graphical user interface to the high-level controller board, preferably hosted on a hand-held device.
3) **Accessible documentation:** Finally, the FlexSEA system still does not feature intuitive documentation, which is the main roadblock to using the FlexSEA system as a complete control package. The design files and documentation related to the high-level controller and user interface have to be made easily accessible to the research community, both in terms of availability and the level of skill required to get started with the system.

1.5 Project Scope

This thesis presents the development of an open-source high-level controller (RICNU Plan) and a wireless graphical user interface (RICNU User) for open-source powered robotic leg devices that utilize the FlexSEA motor control system. The major components of the project included the hardware design and firmware development of the high-level controller board, software and graphical design of the user interface mobile application, integration of the two components with the FlexSEA motor control system and testing of the resulting control system on a custom-built BLDC motor test setup.

1.6 Thesis Contribution

The result of this thesis work is a fully-functional first version of the abovementioned high-level controller board and user interface application, able to work together with the FlexSEA motor control system to provide planning support to powered robotic leg prostheses. Although any compatible mechanical platform can be used with the RICNU/FlexSEA system, the project has been geared towards the open-source robotic leg described above. All design documents and files associated with the project have been made publicly accessible in an open-sourced, version-controlled fashion.

1.7 Thesis Structure

This document has been created using a general-to-specific topology. First, Chapter 2 introduces to the reader the RICNU system as a whole, including brief overview of both control components – the RICNU Plan controller board, and the RICNU User application for mobile hand-held devices. Then, Chapters 3 and 4 delve into the detail on the design and development process of RICNU Plan and User, respectively. Final chapters include information on system testing and future work considerations.

Aside from serving as an academic thesis, this document will also be the first point of reference for potential users; to satisfy both audiences, Chapters 3 and 4 introduce general concepts and decision-making rationale first ("Design Considerations" sections), followed by a detailed discussion of the design.

2 RICNU System

This thesis presents the development of an open-source high-level controller and user interface for an open-source powered robotic leg prosthesis. The prosthesis, high-level controller and user interface have been collectively named the RICNU System, as they were developed through close collaboration between the Rehabilitation Institute of Chicago (RIC) and Northwestern University (NU). Since the conception of the system, the former organization has become the Shirley Ryan AbilityLab (SRALab), but the name for the leg and control system remained. The RICNU System consists of three open-source components:

- RICNU Leg – a two degree of freedom (DOF) lower limb prosthetic device
- RICNU Plan – a high-level controller board that performs planning and state-machine functions
- RICNU User – a mobile device application that serves as the interface to the user

The focus of this thesis has been the development of the RICNU Plan board and the RICNU User mobile application; the RICNU Leg has been developed independently by engineers at RIC/SRALab. At its foundation, the RICNU system is an extension of the open-source FlexSEA motor control system; Figure 2 depicts the control level roles in a typical RICNU/FlexSEA system implementation.

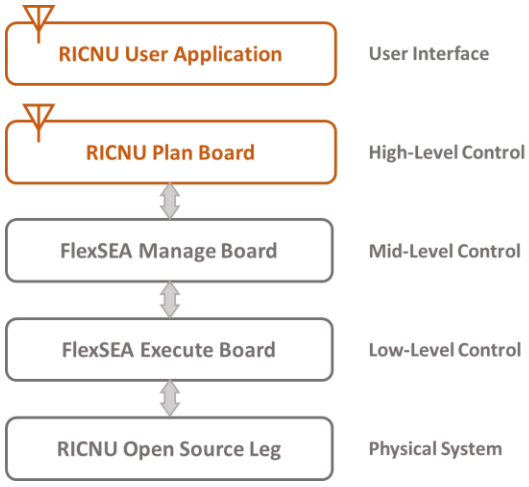


Figure 2: RICNU/FlexSEA Control System

The RICNU Plan board closes the full planning control loop in the absence of a desktop computer and the RICNU User application solve FlexSEA's main problem from the researcher's point of view – no user interface beyond a desktop computer-based high-level control GUI. If patient testing is to leave the vicinity of the lab bench, a wireless solution is required, which is where the RICNU system fills the void.

2.1 General Requirements

As mentioned above, research groups currently do not have access to a universal open-source prosthetic control system to enable researchers to quickly start experiments with patients. While the FlexSEA system has addressed the lower-level control roles in such a system, the RICNU System has been designed to take a high-level control role while maintaining the open-source, flexible nature of the FlexSEA system. To accomplish the above, it was decided that the RICNU System must:

- Provide high-level control in the form of a user-configurable DCFSM
- Enable the researcher to perform patient testing tasks away from a desktop computer
- Provide easy access to patient data during and after testing
- Insulate the researcher from software and hardware development as much as possible
- Be compatible with the FlexSEA system to ease the system learning curve
- Retain and enhance the system-level flexibility of the FlexSEA boards
- Be publicly accessible in a free and open source format
- Provide easy-to-understand documentation that would enable a researcher unfamiliar with the system to set up and start working with the RICNU system in no more than a few hours

2.2 Core Solutions

The following system-level design choices, with justifications, address the above requirements:

Patient testing away from a desktop computer: The choice was made to bring the UI to a hand-held mobile device, where the user can have wireless graphical control the prosthetic device. This will enable the prosthetic device to function autonomously from a desktop computer, and yet offer the user a way to control the device graphically. Also, the system may be used in places without access to wireless internet.

High-level control (device control finite state machine): The choice was made to host the high-level control algorithms on the prosthetic device, as opposed to a remote location connected to the prosthetic device wirelessly. Motion happens in real time, and responsive control has to be asserted accordingly. Only a native control system can achieve this without excessive latencies. Furthermore, the system may be used in settings not furnished with wireless networks. Consequently, the best choice is to host the planning algorithms on the prosthetic device itself – on a hardware controller board.

Easy access to patient data during and after testing: The choice was made to log data on the hardware board and display data on the mobile application. This prevents packet loss associated with wireless communication but still allows the user to view the data during testing.

Elimination of need for software and hardware development: The choice was made to create single configuration file in an intuitive XML-style language, which can be ported to the embedded system in a variety of ways. Once there, the board can parse this manifest and build its own internal representation of the finite state machine automatically. This way, user errors are isolated from the board's firmware, and no additional debugging is necessary. Among other benefits, this way of configuring the DCFSM also allows the user to track previous DCFSM configurations using version control software. While, ideally, the user would have access to a graphical configuration interface for quick control system prototyping and

testing cycles, there are simply too many components to configure in a practical way. The DCFSM could also be configured as part of the hardware controller board's firmware. But, that would defeat the purpose of this system as a shield between the intricacies of low-level firmware and direct patient experimentation. In addition, this would expose the system firmware to user-generated code bugs.

Compatibility with FlexSEA System: Fortunately, FlexSEA defines a single data interface to its Manage board – consequently the choice was made to use an SPI bus connecting FlexSEA Manage and the planning hardware. On the data link layer, FlexSEA protocol has to be used to request data and send commands to the FlexSEA System. Hardware compatibility (connectors, tools, etc.) is also maintained to limit confusion and frustration when building the combined architecture for the first time.

Retain system-level flexibility of FlexSEA: The hardware controller should be able to work as a simple data pass-through (SPI to Bluetooth), as well as a high-level controller. If the research team decides to host the high-level control algorithms somewhere else (mobile device, etc), the board can serve as a data logger and a data translator between the prosthetic device and a mobile application.

Public access and open-source initiative: The choice was made to maintain project code, design files, and documentation using a set of remote version control repositories. This allows all design documentation for all components to be made available online in a version-controlled manner, and the research community to participate in future development of the project.

User-friendly documentation: The choice was made to provide an additional user manual describing how to get started with the system, the DCFSM manifest specification, and example DCFSM manifests and code. The information contained within this thesis is more comprehensive than would be required in a quick-start manual.

2.3 RICNU System Architecture

The choices above converged to a system consisting of a hardware controller board and a mobile device application. To put this in a visual perspective, Figure 3 shows the default components and interactions in a typical combined RICNU/FlexSEA architecture. Components highlighted in orange have been the focus of this thesis work. While this particular setup may be used for a one degree of freedom (DOF) project, such as the RICNU Knee, the flexibility of the FlexSEA and RICNU systems also allows control of more than one actuator at the same time [12].

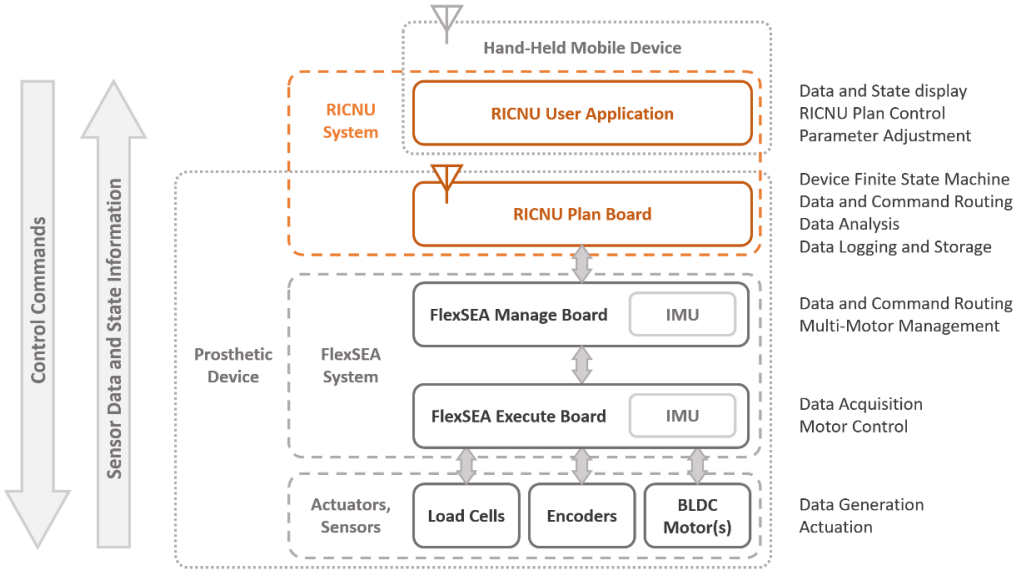


Figure 3: General system overview

2.3.1 FlexSEA Execute

The low-level control to the prosthetic actuator (a BLDC motor) is provided by the FlexSEA Execute board. Besides low-level motor control, this board is also able to gather on-board and off-board sensor data. It has an on-board IMU and a multi-interface expansion connector for communication with a variety of different devices if necessary. Projects involving multiple-DOF devices would incorporate multiple Execute boards, all interfacing with one or multiple FlexSEA Manage boards.

2.3.2 FlexSEA Manage

FlexSEA Manage, in theory, can handle high-level control algorithms, but it lacks good user interface and firmware documentation. Although the FlexSEA project tried to make it easier for the researcher to get started with the board, the user is still faced with low-level C code. While “flexible” in its system role, FlexSEA Manage requires significant engineering intervention to be usable for direct patient testing tasks or to act as the planning controller. Consequently, for this project the Manage board is only configured as a communication protocol translator – from RS-485 to SPI and vice versa.

2.3.3 RICNU Plan

The high-level planning algorithms are hosted on the RICNU Plan board. This board is furnished with a Bluetooth Smart Ready transceiver and a socket for a microSD card. The Bluetooth module is used to wirelessly send sensor and device state data to the RICNU User application, while the microSD card has two roles in the system: 1) it can store patient data logs, and 2) it carries the DCFSM configuration manifest, generated by the researcher on a desktop computer. Porting the DCFSM to the RICNU System is as easy as copying and pasting the DCFSM manifest file from the desktop computer to a microSD card, and plugging it into RICNU Plan. Adjustment of DCFSM parameters can be done using the RICNU User application.

To maintain compatibility with FlexSEA, RICNU Plan communicates with FlexSEA Manage using the FlexSEA communication protocol. And, while RICNU Plan is primarily intended to be a high-level controller, RICNU Plan can still be interfaced with a multitude of sensors using various communication protocols. The addition of the Bluetooth transceiver to the system enables communication with wireless sensors such as BLEfob by BlueRadios, which is a BLE-enabled wireless 12-bit accelerometer [13].

2.3.4 RICNU User

The three electronic boards (FlexSEA Execute, FlexSEA Manage, and RICNU Plan) are physically attached to the prosthetic device. The RICNU User application, on the other hand, interacts with the user via a mobile hand-held device. It can be used as a wireless data and state display, a graphical tool to control the operation of RICNU Plan, and even a configuration tool for the DCFSM control of the prosthetic device. For multi-limb prosthetic and orthotic systems, the RICNU User application can be used to synchronize and share information between prosthetic and orthotic devices on different limbs wirelessly. The graphical nature of the RICNU User application abstracts away the details of low-level firmware hosted on RICNU Plan and the two FlexSEA boards. The application can be used to control system states of RICNU Plan, as well as adjust configuration parameters of a pre-configured finite state machine controller hosted on the board.

2.3.5 Information Flow

Referring back to Figure 3, the four control components of the combined RICNU/FlexSEA system follow a strict top-down hierarchy to maintain the high-mid-low control structure. Data is collected by FlexSEA Execute and is sent upstream to Manage, Plan, and, finally, User. RICNU Plan adds device state information to this data flow upstream. Commands flow the opposite way. Generated by the user via RICNU User application, they are sent downstream to RICNU Plan wirelessly, and processed there. RICNU Plan autonomously maintains control of the prosthesis using a finite state machine; commands generated by RICNU Plan flow to FlexSEA Manage; they go through a translation process and finally reach FlexSEA Execute, where they are incorporated in low-level motor control loops.

Importantly, RICNU User is not used for direct motor control; it is only used to display sensor data and state information and to adjust the finite state machine parameters hosted on RICNU Plan.

2.3.6 Communication Hierarchy

The four control components of the combined RICNU/FlexSEA system follow a top-down master/slave hierarchy, in step with the FlexSEA communication protocol. Higher-level components behave as masters to lower-level components, which prevents data collisions and the need for bus arbitration. Figure 4 presents the relationships between the different components in the overall system.

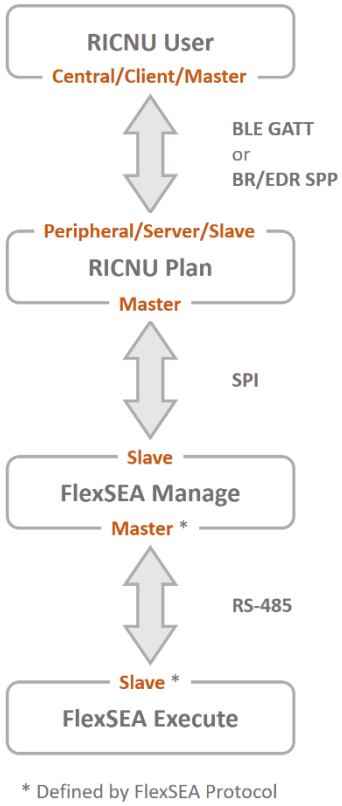


Figure 4: System communication

Starting from the bottom of the communication hierarchy, Execute and Manage communicate using RS-485, specified by the FlexSEA architecture. While the popular and time-tested electrical standard does not define masters and slaves on its own, the FlexSEA communication protocol defines Manage as a master to Execute.

Next level up, communication between Plan and Manage happens via SPI as, again, defined by the FlexSEA system architecture. The master/slave relationship follows a top-down hierarchy – RICNU Plan requests data and sends control commands downstream, while Manage only responds to command packets sent by Plan.

The Bluetooth communication link between Plan and User places the Plan board in the Peripheral/Server/Slave role, and the RICNU User application in the Central/Client/Master role. The asynchronous nature of the communication ensures that RICNU Plan receives important system commands as fast as they arrive; on the other hand, RICNU User is not part of the actual real-time device control loop; it merely reads sensor and state data to display to the user. So, deterministic timings are not essential here.

2.4 RICNU Plan Overview

RICNU Plan is a multi-functional board designed to provide high-level control to a powered prosthetic device through interacting with MIT's FlexSEA motor control system. Just as importantly, it is also a middleman between the prosthesis and the researcher – it provides data and control information to the researcher's mobile device in real time via a wireless link.

The board was designed to survive misuse in a lab environment and provide various communication interfaces for system flexibility, as well as an easy means of obtaining sensor and state data. The design also boasts very low power consumption, which is great for battery applications.

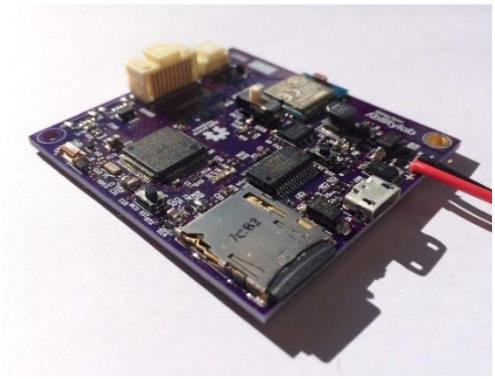


Figure 5: RICNU Plan board

2.4.1 System-Level Requirements

As a high-level controller geared towards a research environment, it was decided that RICNU Plan must:

- Assert autonomous planning and control over the lower-level boards
- Be compatible with the FlexSEA communication stack and physical features of FlexSEA boards
- Provide an intelligent, human-readable way of configuring its device control finite state machine
- Provide a seamless wireless data link to a hand-held mobile device
- Provide an easy means of extracting patient data from the system
- Provide an easy means of quickly porting the board from a benchtop onto the prosthetic device
- Be compatible with communication protocols like SPI, I2C, RS-485, CAN, and UART/USART
- Feature easy on-board programming interfaces for quick firmware setup and updates
- Consume less than 250mW during normal operation
- Withstand harsh misuse conditions of an integrated lab environment

2.4.2 System Design

The board's main microcontroller carries out all execute functions on the board. That includes serving as a master in Plan/Manage communication using the FlexSEA communication protocol, data server for the User application, and various configurable roles in user-defined external sensor applications. RICNU Plan routes data between User and Manage, analyzes that data, and controls the prosthetic device at a high level using a user-defined DCFSM.

The microSD Card is what enables easy data transfer and intuitive configuration of the DCFSM. Besides storing data and state log files generated during active control of the prosthetic device, the SD Card also hosts a user-generated manifest file that specifies the DCFSM used to control the prosthetic device. This file is created by the researcher using intuitive XML specification, isolating the researcher from the intricacies of low-level embedded firmware. The board's main microcontroller reads this file and builds its own internal representation of the DCFSM. Upon entering the active control state, the board uses this DCFSM representation to carry the prosthetic device through various control states based on sensor data.

The Bluetooth module serves as a serial data pass-through between RICNU Plan and RICNU User to maintain asynchronous communication between the two. Besides a slight processing delay, the module acts as a direct UART bus between the User application and the Plan control board, taking care of BLE and Bluetooth Classic connectivity events independently. When data comes in from the microcontroller, the module routes it directly to the mobile device without waiting for any response from the application. In the same fashion, since the control commands from RICNU User would be coming at random times, the module routes command data to RICNU Plan's main microcontroller whenever that data comes in.

As a member of a hierarchical system, Plan assumes a master role with respect to FlexSEA Manage (via SPI), and a slave/peripheral/server role with respect to RICNU User (via BLE GATT or BR/EDR SPP).

2.4.3 User Workflow

To control the prosthetic device using a new finite state machine, the user simply needs to follow the workflow shown in Figure 6.

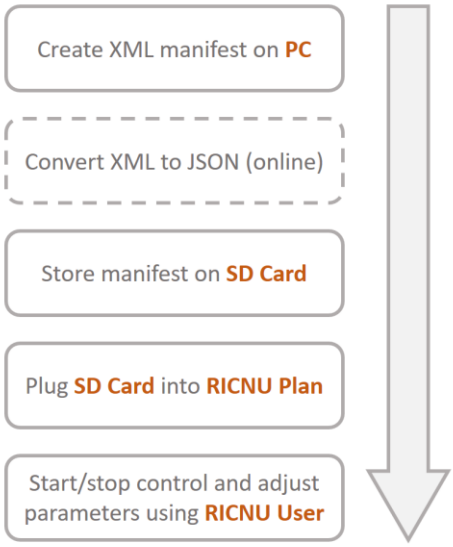


Figure 6: User workflow

The simple process starts by the user creating an XML manifest file based on one of the example manifests provided with the open-source code. The RICNU-specific XML specification is outlined the Quickstart Manual included with the online documentation package for the project; Appendix A contains an example manifest used in testing the system.

The manifest is then loaded onto a microSD card, and the card is inserted into RICNU Plan – no matter if powered or unpowered. RICNU Plan reads the manifest, parses through it, and builds an internal representation of the state machine. Then, when connected to RICNU Plan through the User app, the user can put Plan into the "active control" mode, in which Plan performs various tasks like actively controlling the prosthetic device using the finite state machine, data logging, and other functions.

In this first version of the firmware, there is an extra step involved – the XML manifest is converted into the JSON format using any appropriate online tool. As opposed to XML, JSON is much easier to parse using lightweight libraries like *cJSON* and *JSMN*, which cuts down on code size; JSON files can be much smaller than XML files as well, leaving more room on the SD Card for log files. Although an extra step for the user, this accelerated the firmware development for the author. But, even though an extra step, it is very quick and easy to perform. An example JSON manifest file generated from the example XML manifest listed in Appendix A is in Appendix B.

2.4.4 Hardware

Figure 7 shows the general system-level diagram of the RICNU Plan board. A much more detailed hardware diagram can be found in Section 3.4: Hardware Design.

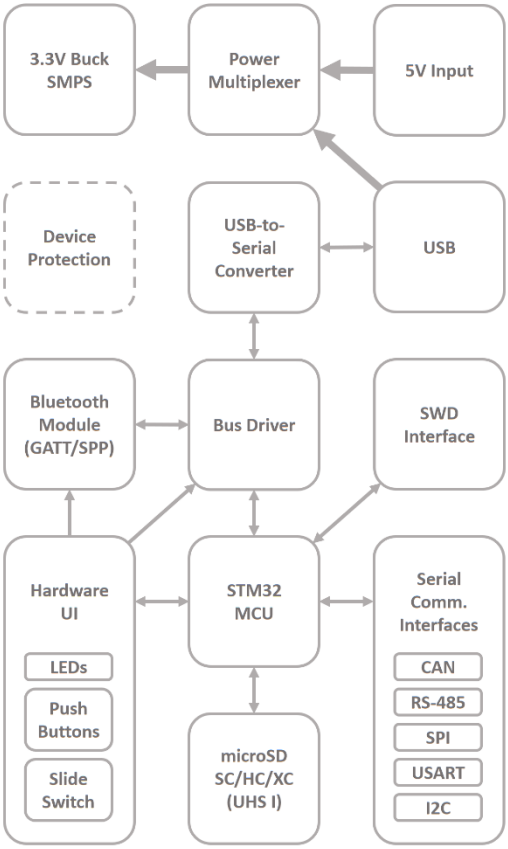


Figure 7: RICNU Plan general system diagram

At its core, the board uses an ARM Cortex-M3 microcontroller (MCU) from STMicroelectronics, STM32F103RD. The microcontroller hosts the prosthetic device state machine and routes data to the RICNU User from downstream sources, such as FlexSEA Execute and Manage. For that, the board incorporates a variety of serial communication interfaces.

The MCU uses a UART bus to communicate with the Bluetooth module, which then communicates data to and from the User application. The module can be programmed in-application using the board's USB port, and also through its UART peripheral. To manage communication between the three components, the board utilizes a quad tri-state buffer chip. The MCU also communicates with a microSD card (SC/HC/XC are supported up to UHS I class) using one of its SPI peripherals.

The MCU can be programmed and debugged through the Serial Wire Debug (SWD) interface used by all ARM Cortex M processors. The processor can also be flashed using an embedded UART-based bootloader through the multi-interface connector. The Bluetooth Module can be programmed in-application through the USB connection on the board.

Power can be provided to RICNU Plan through either USB or via another standalone 5V supply. Both are managed by a power multiplexer, which preferentially chooses USB when available. Both inputs can safely handle reverse polarity hot plug events, overvoltage conditions (up to 20V), and ESD events. The output of the input and multiplexer stage is fed to a 3.3V buck switch-mode power supply (SMPS) and a filtering stage, which provides a steady 3.3V source to the rest of the board. All inputs and outputs are protected against ESD, overvoltage, and overcurrent events, with additional protection on the power inputs, as mentioned.

2.4.5 Hardware Capabilities

In summary, the hardware capabilities of the RICNU Plan board include:

- Flexible power options – compatible with any 3.3V-5V volt sources, including USB 2.0/3.0
- MicroSD SC/HC/XC Card interface for easy access to real-time patient data
- Serial debugging interface for its powerful STM32F103RD (ARM Cortex-M3) microcontroller
- Seamless USB programming capability for its on-board Bluetooth module
- Simple physical user interface offering debugging capabilities and mode selection
- Up to 3 UART interfaces
- One full USART interface that can be used in flow control and clocked modes
- Up to 2 I2C interfaces, including one compatible with SMB alert protocol
- One Controller Area Network interface
- One half-duplex RS-485 interface
- Up to 2 SPI interfaces (one by configuration of the USART peripheral)
- Bluetooth Low Energy custom GATT and/or Bluetooth Classic SPP
- Low standalone power consumption – below 250mW during normal operation
- Robust electrical protection from user errors in lab environments

2.4.6 Software

RICNU Plan acts as the high-level control component in the system. As such, it maintains a DCFSM structure in real time. The Plan board builds this structure based on a special manifest configuration file hosted on the board's microSD card. The building process utilizes a variety of third-party libraries in combination with a custom-developed API-like parser.

The board's state is fully controlled by the RICNU User application. The app can be used to make Plan perform a calibration procedure, log data, assert active control on the prosthetic device, and other actions.

RICNU Plan uses an efficient interrupt-driven firmware architecture to offload communication tasks to its microcontroller's peripheral hardware, focusing the CPU efforts on computationally-intensive processes. The firmware utilizes the microcontroller's powerful Nested Vector Interrupt Controller (NVIC) to schedule tasks without a dedicated operating system.

Data logging is performed using a FAT file system module, which enables the user to easily access data logged onto the microSD card using a personal computer. The logging is maintained using a queue implementation, such that no log entries are lost due to SD Card waiting periods. Each log entry is time-stamped and given a unique ID, so the researcher is able to know exactly when certain events occurred.

The Bluetooth module, USB-to-Serial converter, main MCU, and the SD Card all operate independently of each other, and only exchange data when needed and available. Thus, the board can be put into two modes: normal operation, and Bluetooth module programming. The switch is made using a hardware slide-switch that directly controls the serial bus driver on the board and puts the Bluetooth module in programming state after it is reset using a hardware push-button.

The current microcontroller firmware defines three states for the system, controlled by the RICNU User app: active, idle, and error. The system state is indicated using LEDs of different colors. During the active

state, the system performs various tasks chosen by the user in the User app, like data logging, active device control, and others. In idle state, the controller relaxes the leg and goes into deep sleep to conserve power. Error state is very similar to idle, except the user is notified of the state both through User app and an on-board LED.

As a result of all of the above, the RICNU Plan board can communicate data to any Bluetooth LE or Classic devices, gather and write data to any standard serial communication devices, log data onto high-capacity SD Cards for easy "plug and play" data analysis, be easily reprogrammed by the user, and is very tolerant of user errors in a lab environment. More importantly, it is able to build an internal representation of the device control finite state machine (DCFSM) based on information written by the user using an intuitive XML specification and hosted on the device's microSD card. This effectively insulates the researcher from low-level firmware code. The real-time wireless communication between RICNU Plan and RICNU User allows the researcher to easily get started with testing various state machine without much engineering intervention.

2.5 RICNU User Overview

The RICNU User application is a graphical user interface to the RICNU Plan board and is hosted on a hand-held mobile device. In its current state, it acts as a data display and a system state control panel for RICNU Plan; with future development of the system, it has the potential to also become a direct interface to the DCFSM maintained on RICNU Plan in real time. The application has been designed to be simple, intuitive, and compatible with a variety of Android devices.

2.5.1 System-Level Requirements

As a graphical user interface to the RICNU Plan board and thus all of the downstream system components of prosthetic device, it was decided that RICNU User must:

- Provide a seamless and natural interface to RICNU Plan's data communication protocol
- Enable the user to send system state commands to RICNU Plan
- Provide a seamless wireless data link to RICNU Plan's custom Bluetooth profile
- Allow for unimpeded throughput of data to the user in real time
- Provide a means for maintained wireless control over the entire prosthetic control system
- Be compatible with a large variety of mobile hand-held devices

2.5.2 System Design

The RICNU User application links to RICNU Plan wireless using a Bluetooth Low Energy (BLE) link and is hosted on an Android platform. BLE is available on a large majority of smartphone devices, and Android phones are widely popular and inexpensive. At this early stage in the design, the application separates user tasks into Bluetooth connectivity tasks and data monitoring. Each group of tasks features a dedicated Android activity that has a unique layout and interface features. In the background, the application runs a BLE service, which maintains stable connection to the RICNU Plan board and takes care of all of the low-level BLE tasks. Figure 8 depicts the RICNU Plan task organization.

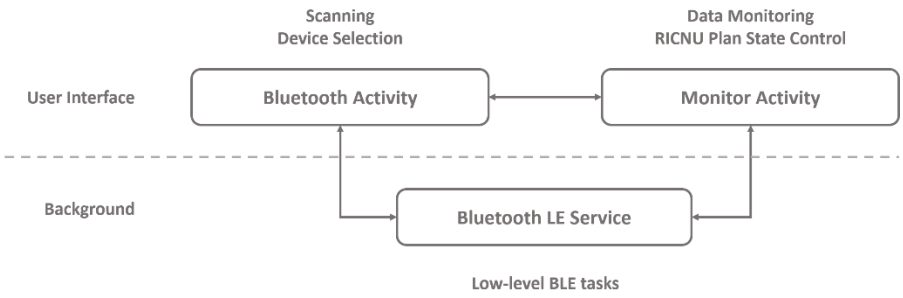


Figure 8: RICNU User task design

2.5.3 User Workflow

The Bluetooth Activity is started on app startup. It is used to scan for nearby BLE devices, select RICNU Plan among them, and connect to the board. Upon successful connection, the Monitor Activity is used for real time data monitoring and RICNU Plan state control (idle, active, calibration, etc.).

2.5.4 Software

To navigate all of the various states of Bluetooth connectivity, application flow and user interface, the application utilizes multiple state machines. The three main components communicate with each other using Android broadcasting methods. Custom classes are used to process sensor and state data coming from RICNU Plan and generate control commands sent to RICNU Plan.

The software has been developed using the free and easily-accessible Android Studio and is compatible with any Android device using Android Version 4.4 (KitKat, API 19) and higher. At this time, KitKat has already become an obsolete platform, meaning that the RICNU User application is compatible with all modern Android OS versions.

2.6 Design for Research

Together, RICNU Plan and RICNU User enable the researcher to interact with the patient's prosthetic device directly and wirelessly. No software is required to be run on a desktop computer during real-time control, so patient testing and data acquisition can take place outside of clean lab spaces and into rugged, real-life environments, such as street sidewalks and the patients' home areas. Perhaps with more software development, the RICNU User application can be used as a training tool for both patients and researchers.

The design files, documentation, and software for both RICNU Plan and User are available publicly in an open-source format. Instead of spending months and, perhaps, years on developing a new prosthetic

device, research labs worldwide are able to access the design files, build the system, and quickly move on to continuing the scientific inquiry on human movement and physical disability.

Finally, the RICNU system has been designed with the researcher and an entry-level research engineer in mind – not a specialized engineer versed in electrical and software engineering concepts. Besides this detailed document, documentation for the project also includes a concise, easy-to-understand Quickstart manual that is publicly accessible on the internet.

While the developers of the FlexSEA motor control boards have tried to address the lack of a universal open-source prosthetic control system in the research environment, the RICNU System fully closes the control loop and provides the researcher with an easy means to interface with the FlexSEA boards. The author hopes that the RICNU/FlexSEA system will become a standard research device across multiple laboratories, accelerate the search for understanding of the human movement, and ultimately, aid in development of better and better prosthetic device control algorithms.

3 RICNU Plan

Now that the system-level requirements and the general design of the RICNU System have been introduced, RICNU Plan will be discussed in this chapter separately. RICNU User will be discussed in Chapter 4. Aside from serving as an academic thesis, this document will also be the first point of reference for potential users. To satisfy both audiences, Chapters 3 and 4 introduce general concepts and decision-making rationale first, followed by a detailed discussion of the design. Consequently, the design discussion sections resemble a technical reference manual more than an academic document.

3.1 Design Philosophy

Referring back to the requirements listed in Section 2.4.1, the reader may be able to catch multiple underlying trends. As in any design, decisions are made based on compromises, for example between power consumption and component size or between user-friendliness and functionality. This section describes the general guidelines and used to evaluate and decide on various design choices.

First, the board's main purpose is to act as a high-level controller for the RICNU robotic leg prosthesis. Consequently, the physical size of the board needed only to be small enough to be conveniently mounted to the RICNU leg. Subject to the overall size constraint, components with lower power consumption were preferred. This allows for battery power to be conserved so that the patient may walk with the leg longer. Next, user-friendliness was prioritized. Since the board is intended for use by various researchers who do not have a strong engineering or software background, robust protection circuitry was incorporated to prevent damage in case of misuse (i.e. overvoltage, wrong polarity, etc). System-level decisions were made to separate engineering support from the researcher as much as possible. Even if the initial hardware setup may require some effort from a relatively inexperienced engineer, the actual patient testing and software setup does not require an engineer at all – even the device control state machine

configuration and board firmware updates can be performed by non-engineering personnel – just by following documentation instructions. In some sense, the board was designed as a prototyping board, as well as a research device. Being able to quickly switch between testing a control algorithm in a simulated benchtop environment and testing that same algorithm on a prosthetic device is a great way to accelerate the research process.

When designing the "flexible" part of the board – the multiple configurable communication interfaces – choices that offered the highest diversity and number of available communication peripherals were prioritized, provided that the overall board remained small enough to be mounted on the leg. Similarly, the choice of on-board memory storage and its system role was devised based on ease of access to data (so that MATLAB or Python based analytical tools could be used), rather than minimizing the board area and other factors.

Thirdly, compatibility with the FlexSEA system was maintained despite some drawbacks. As one example, FlexSEA boards use specific inter-board connectors. RICNU Plan was still designed to use the same line of connectors even though they are somewhat expensive and difficult to use. The benefits of using the same debugging interface and becoming familiar with only a single set of tools outweighed the aforementioned drawbacks.

Generally, compatibility was held in high esteem. For instance, selection of the wireless communication technology was heavily influenced by what technologies are most-widely used in hand-held mobile devices. The power stage was designed to be compatible with most standard power supplies, including 5V USB, present in modern computers and mobile devices.

Next, firmware libraries, development tools and hardware components that are free and/or open-sourced were chosen over those that may have better capabilities but are restricted in use. This was done in the

spirit of the open-source initiative. To use the system, the users should not have to purchase anything other than specific third-party hardware components used in the board design. This philosophy was followed when making most design decisions, except for ones influenced by hardware compatibility with the FlexSEA system. Software solutions focused on ease of development time and external code libraries were chosen based on their interface simplicity, robustness, and code size – the smaller, the better. Processor firmware was created with power considerations in mind.

Finally, a counterbalance to *all* of the above factors and issues is time. Having little more than 9 months to design, build, and test both RICNU Plan and User, the author was forced to make some system-wide choices out of necessity to complete the project in a timely manner.

3.2 Hardware Design Considerations

When designing RICNU Plan, each part of the board underwent a studious decision-making process. The following system-level components of the board were considered in detail:

- Embedded processor to use as the main computing component on the board
- Wireless transceiver to interface with a mobile hand-held device
- Data storage memory unit, used for both data logging and device configuration
- On-board programming interfaces for ease of firmware update procedures
- Power supply design that would allow for compatibility and interchangeability of external power
- User interface features like pushbuttons and slide switches for mode selection
- Device protection features that would protect the board from user errors and electrical events

3.2.1 Embedded Processor

3.2.1.1 Requirements

Each embedded system can be thought of as a "black box" that takes in inputs, processes them, and affects another system by generating outputs. To process various inputs, small embedded systems usually utilize microprocessors. They are used for control, communication, data collection and computational tasks, and are essential in applications like this.

RICNU Plan has to communicate with a lower-level FlexSEA Manage board, relay data to and receive control adjustments from a hand-held device application, and log data onto a memory storage device for data analysis on a separate computer. To aid with all these tasks, the board needs a powerful processing unit that:

- Can communicate to external devices via various communication protocols
- Offers the developer *and* end-user a very flexible level of control over its hardware
- Consumes as little power as possible
- Offers the end user an easy method of reprogramming its firmware
- Is computationally-powerful enough to provide sophisticated finite-state machine control over a prosthetic device, collect and transmit data, and maintain control over other on-board devices

3.2.1.2 Microcontrollers vs Embedded Computers

When choosing a processing device, the developer is first faced with a decision between using a microcontroller and an embedded computer. The former is a bit easier to define than the latter.

For the purposes of this work, a *microcontroller unit* (MCU) is a device containing a microprocessor, volatile and non-volatile memory, various input/output (I/O) units, and analog circuits within one

integrated circuit (IC) chip. An MCU does not have a dedicated operating system (OS), and the developer has full control over the hardware units of the device.

Embedded computers (sometimes referred to as microcomputers) generally offer higher computational power at the cost of higher power consumption and hardware rigidity. An embedded computer is a System-on-a-Chip (SoC) device that has a powerful processor, separate memory units, and minimal number of inputs and outputs, usually used purely for communication interfaces. These devices run operating systems, which take care of hardware manipulation for the developer. Examples of these boards include systems like Raspberry Pi, Orange Pi, BeagleBone, and others.

In general, microcontrollers offer more flexible I/O control and lower power consumption. Instead of a having to reach down to lower firmware levels through an abstract operating system, the developer has full control over the device's hardware. However, microcontrollers offer little in the way of software abstraction – the firmware is usually written in a low-level language like C or assembly. Embedded computers, on the other hand, can run very sophisticated software and interpret various programming languages, but they tend to be very rigid in terms of their hardware. These devices usually also consume more power than small microcontrollers. For example, most power-efficient Raspberry Pi boards consume around 80mA of current at 5V supply according to some community-sourced application tests [14], while some small microcontrollers can consume as little as microamperes of current at supplies of 3.3V and lower [15].

In this application, power consumption and hardware access are key. Although RICNU Plan has to process some data, the computational burden is not sufficient to require a full embedded computer.

3.2.1.3 MCU vs. FPGA

Field-Programmable Gate Arrays (FPGAs) are powerful devices that let the developer build their own hardware logic. But, the tasks needed to accomplish by the embedded processor do not call for custom logic. There is no signal processing involved, and there is no benefit to be had from designing parallel computation logic. Furthermore, FPGAs are not optimized for low power consumption. Finally, for the same amount of logic, FPGAs are also more expensive than microcontrollers.

3.2.1.4 Microcontroller Type

While choosing a microcontroller, the designer is first must choose between using an 8-bit, a 16-bit, or a 32-bit architecture. 8-bit architectures offer lowest power consumption and fastest I/O operation, while 32-bit devices can perform certain software computations faster and have less restrictions on memory and hardware interfaces. Some 32-bit devices offer floating-point capabilities in addition to standard integer operations. While 8-bit and 16-bit microcontrollers are still popular among developers, the community is embracing the 32-bit devices because of their superior computational power. RICNU Plan deals with data arrays containing 32-bit values, as per FlexSEA communication protocol, and a 32-bit device is more efficient at these tasks than an 8-bit MCU. Within the 32-bit world, an important distinction is whether a particular microcontroller features a digital signal processing (DSP) unit. These types of microcontrollers are great for performing parallel computations on digital signals. In this application, the DSP capability is not needed.

3.2.1.5 Processor Architecture

The choice of a specific architecture is hard to make ahead of development time. There is a lengthy list of architectures to consider, but a safe choice is the very well-documented and widely-supported ARM architecture. ARM processors are based on RISC (Reduced Instruction Set Computing) architecture, allowing these devices to be smaller in size than their competitors and consuming power more efficiently.

ARM architecture MCUs include ARM Cortex-M0/M0+, M1, M3, M4, and M7 processors. Advancing in that order, these processors offer more advanced computation at the cost of higher power consumption. With a power budget of only 250mW for the entire board, an M0 device was initially used for development. Later, as constraints on memory and peripherals became clear, M3 devices were used.

3.2.1.6 Manufacturer

Many manufacturers license the ARM Cortex-M architectures and choosing a particular MCU family is often a matter of personal preference, as most have very comparable electrical and computational characteristics. Having very positive prior experience with low power 8-bit MCUs from STMicroelectronics (STM8L), the author settled for using the STM32 line of microcontrollers. STM32 devices enjoy very accessible development tools, a wide development community, and very good documentation.

Initial development was done on the STM32F0DISCOVERY board from ST, employing a 48-pin ARM Cortex-M0 microcontroller. As memory constraints started affecting development, the project moved onto a larger STM32L1 (low-power, Cortex-M3) controller on the 32L152DISCOVERY board. During the hardware design phase of the project, inter-board compatibility requirements emerged. To be used with other boards developed by engineers and researchers at the Shirley Ryan AbilityLab, CAN and RS-485 interfaces were added to the design, forcing the firmware development to move on to the next-tier series – the STM32F103xx series of microcontrollers.

3.2.1.7 Form Factor

STM32F103xx devices come in a variety of packages, including 48-, 64-, 100-, and 144-pin LQFP, BGA, and QFPN packages. For an open-source, “build/debug it yourself” design, LQFP packages fit best, as they offer gull-wing type pins, easily soldered by hand. In future designs, LQFP can be changed to a

QFPN, or even a BGA to reduce the overall size of the board. To satisfy all of the I/O requirements while conserving board real estate, a 64-pin device was chosen – STM32F103Rx.

3.2.1.8 On-Chip Memory

The differences between MCUs in this family lie mainly in the sizes of RAM and Flash memories available. During firmware development, it was determined that RAM needs to exceed 32 kB at minimum. Size of flash memory did not matter in selection since these devices have more than enough flash for this application.

3.2.1.9 Other Considerations

It is worth noting that the FlexSEA Manage board also utilizes a STM32 microcontroller, although a more powerful one from the F4 family (using ARM Cortex-M4 CPU). Debugging tools and code can be easily migrated between STM32F4 and STM32F1 devices, thanks to a well-documented firmware library developed by ST. Also, the wireless module used on RICNU Plan employs an ARM Cortex M0 architecture. Using an ARM MCU as the main processor means that the end-user only has to learn how to use one debugging mechanism for all three devices.

3.2.1.10 Final Decision

With the above considerations in mind, the final choice for the main microcontroller of the system is the STM32F103RDT6 MCU, which offers, among other things:

- ARM® 32-bit Cortex®-M3 CPU, 1.25 DMIPS/MHz (Dhrystone 2.1) performance at 0 wait state memory access, with single-cycle multiplication and hardware division
- 384 KB Flash memory, 64 KB SRAM
- Up to 2 × I2C interfaces (SMBus/PMBus), 5 USARTs (ISO 7816 interface), 3 SPIs (18 Mbit/s), CAN interface (2.0B Active), and USB 2.0 full speed interface

- 51 I/Os, all mappable on 16 external interrupt vectors and almost all 5 V-tolerant
- Various configurable timers, including watchdog timers and a CPU-native 24-bit down-counter
- Serial wire debug (SWD) & JTAG interfaces
- 12-channel DMA controller that supports timers, ADC, DAC, SDIO, I^2C, SPI, I^2 C and USART
- Low power modes, such as Sleep, Stop and Standby modes
- Power-on reset (POR) and power-down reset (PDR) circuitry
- Internal and external high-speed and low-speed oscillator support

3.2.2 Wireless Transceiver

3.2.2.1 Requirements

To communicate with a hand-held device during research experiments and benchtop testing, the RICNU Plan board needs to use a low-power wireless RF transceiver that:

- Uses a short-range protocol compatible with any hand-held device
- Consumes less than 30% of the total power budget in low-power operation
- Takes care of the communication protocol to reduce development time
- Offers the user an easy reprogramming and configuration scheme

3.2.2.2 Wireless Technologies in Hand-Held Devices

Modern hand-held devices are marvels of engineering and technology. Smartphones nowadays employ multiple wireless technologies to enable various kinds of networking, whether on the World-Wide Web, or locally, device to device. The most widely technologies include Wi-Fi, Cellular, Bluetooth, FM radio, and others.

However, most of these are intended for long-range communication, and long-range protocols are undesirable in this application, mainly due to patient data security concerns and Federal Communications Commission (FCC) compliance. Among short-range protocols for Internet-of-Things applications, the most popular technologies are ZigBee, Wi-Fi, Bluetooth (Classic and Bluetooth Low Energy), and WiMAX. ZigBee and WiMAX are not widely used in mobile devices, primarily because the former is outperformed in data throughput by Wi-Fi and Bluetooth, and the latter's development has seen a major downfall in the last decade [16]. Among the two remaining technologies, Bluetooth Low Energy typically offers much lower energy consumption than Wi-Fi, so BLE was picked for this application [17].

3.2.2.3 Bluetooth Technology

Bluetooth technology was first incorporated into commercial products in 2000. In 2004, the Bluetooth Special Interest Group (SIG) adopted the Core Specification Version 2.0: Enhanced Data Rate (EDR). In a quest for even faster data rates, specification 3.0, dubbed "High Speed" (HS), was adopted in 2009.

In improving the technology, speed was of utmost concern. Energy consumption, however, was left largely unoptimized. Bluetooth v3.0 and prior, or Bluetooth Classic, relies on relatively power-hungry technology, unsuitable for some battery-powered applications. With this in mind, researchers at Nokia developed a slightly different technology in the mid-2000s. Today, it is known as Bluetooth Low Energy (BLE), or Bluetooth Smart, and is part of the Bluetooth v4.0 and later specifications [18] [19].

Bluetooth Smart devices cannot directly communicate with Bluetooth Classic devices, due to differences in communication technology. Although there is currently a major shift from Classic to BLE in consumer electronics, some legacy devices can only use Classic, while some newer hand-held devices may soon be using only BLE technology. Middle-man hub devices like personal computers usually incorporate both technologies in one Bluetooth module. Such modules are dubbed Bluetooth dual-mode, or Bluetooth Smart Ready (BSR), and can communicate with both BLE and Classic devices.

As one of the core requirements of RICNU Plan is to be compatible with any modern and legacy hand-held device, the design of RICNU Plan incorporates a Bluetooth v4.1 Smart Ready module. This allows any Bluetooth-compatible hand-held device to host the RICNU User application.

Recently, Bluetooth v5.0 emerged as the cutting-edge Bluetooth technology. However, it does not yet have good community support, and only a couple of released modules actually utilize it. Bluetooth v4.1 and v4.2 were considered to be primary targets in this application.

3.2.2.4 BSR Module Market

When selecting a BSR module, two distinct device groups emerged – stand-alone BSR stack modules, and system-on-a-chip (SoC) BSR modules.

The modules rely on the design engineer to develop a custom antenna that interfaces with the module; the device itself merely analyzes the RF signals and communicates with the main processor. The SoCs provide a complete one-chip solution to wireless interface – they include an RF antenna and other components on a single printed circuit board. While the stand-alone modules are cheaper, SoC solutions significantly reduce development time and prototype reliability. RF antenna design was not within the scope of this project, and Bluetooth Smart Ready SoC were preferred in the selection process.

3.2.2.5 Final Decision

Bluetooth v4.1 dual-mode SoC solutions are not too common. Among them, BT121 from Bluegiga (part of Silicon Labs) was attractive because configuring this dual-mode module is very easy, documentation is good, and the module runs its own Bluetooth stack. External devices can communicate and control the module externally using an API system developed by Bluegiga, or via a UART interface. More importantly, it is running an ARM Cortex-M0 processor, meaning that the same programming mechanisms used by the main processor on the board can also be used to program this device.

3.2.2.6 System Interactions

Referring to back to the overview of RICNU Plan (Section 2.4), the communication between the main MCU and the Bluetooth module should be asynchronous to increase data throughput and maintain the system-level independence of the wireless transceiver from the main microcontroller. An obvious choice for the physical link between the two is the Universal Asynchronous Receive and Transmit (UART) communication. The communication lines are not clocked, and data can be exchanged at any given time.

3.2.3 Data Storage

Two of the major roles of RICNU Plan is to provide high level state-machine control to the prosthesis, as well as to log sensor data, state information, and other data in real time. It is hard to predict how a device will be used, but in general, experiments with prosthetic devices may be lengthy. At an extreme, the device may be given to a patient for a whole day or a week to analyze the patient's use of the device, and RICNU Plan needs to log data throughout the experiment. Clearly, this data can quickly reach Gigabytes in size. Keeping in step with the overarching goal of the project, this memory also needs to be quickly and easily accessible to the researcher for data analysis. Finally, the contents of this memory are likely to be changed very often, as the data is logged and erased for each experiment.

3.2.3.1 Requirements

The above considerations call for a memory storage hosted on the board that:

- Is sufficiently large to host multiple gigabytes of data
- Can be quickly written to in real time (but not necessarily bit-by-bit)
- Can be quickly and easily accessed and analyzed by the researcher
- Retains data while being powered off
- Is either extremely reliable over time *or* can be easily replaced in case of endurance failures

3.2.3.2 Memory Size

The board will periodically be logging sensor data (accelerometer x/y/z, gyroscope, x/y/z, multiple encoders, motor current, strain gauge, etc.) and state information as it comes in, along with log IDs and time when the log was created. The size of that data array would easily reach 100+ bytes in size. For calculations, each sample point will be considered a 128-byte data log entry.

Data is sampled by the FlexSEA system at 250Hz. Assuming that logging happens in parallel and no data is lost due to system latencies, the board would be logging at an average rate of 128 bytes at 250Hz, so 32000 bytes per second, and 1,920,000 bytes per minute. In a 24-hour period, the board will record as much as 2.5+ gigabytes of data.

Clearly, memory size needs to be large, and, at the same time, robust and configurable. Some laboratories may not need a lot of memory, while others may need gigabyte-size memory arrays.

3.2.3.3 Semiconductor Memory

The invention of semiconductor technology allowed development of dense digital memory storage devices that are used virtually in every modern embedded system. Semiconductor memory types are classified on data retention capabilities, data manipulation capabilities, and specific technology, as shown in Figure 9 [20].

3.2.3.4 Volatile vs. Non-volatile

Many memory types have been developed over time, but they can generally be broken up into two categories – volatile and non-volatile. Volatile memory can only store information while powered on and loses that data when powered off. Non-volatile memory types retain written information even when powered off. For this application, non-volatile memory types are essential.

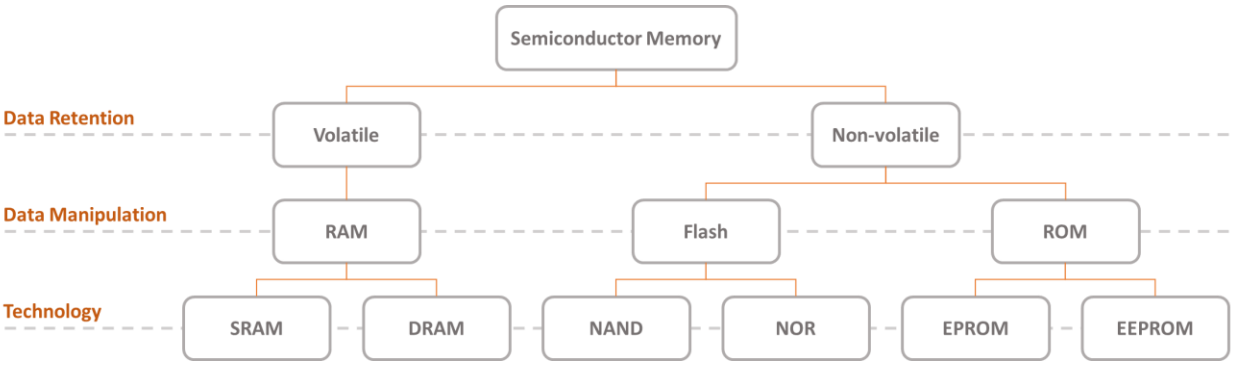


Figure 9: Semiconductor Memory Classification

3.2.3.5 Differences in Technology

Until the invention of Flash memory in late 1980s [21], Random-Access Memory (RAM) and Read-Only Memory (ROM) were synonymous with volatile and non-volatile memory, respectively. RAM can be read/written multiple times, while ROM is generally only written once and can only be read, unless it can be electrically or otherwise erased and written to again. Flash memory, while being non-volatile, can be read and written in bytes (NOR) or blocks (NAND) [22]**.** For this application, Flash memory types are most useful, since data logging needs to be continuous and data must be preserved then the memory is powered off.

Between NAND and NOR Flash, NAND technology offers much higher storage densities at the cost of ease of data access (can be accessed only in blocks), while NOR technology offers easy byte addressing but much lower data densities. Because the chosen microcontroller has a relatively large memory capacity, it is possible to buffer collected data and write it to memory in blocks, so byte-addressing is not essential. NAND Flash memory is most useful in this application [22].

3.2.3.6 Form Factor

With regards to user access and reliability, the choice is between having a dedicated on-board memory chip or a removable storage device. On-board memory chips often occupy less on-board space than

removable devices, but a large disadvantage of the former is endurance and maintenance. Flash memory cannot endure infinite write cycles, and an on-board chip could have to be replaced often enough to be a nuisance. A removable storage device would alleviate this problem, while also providing the end-user with an easy way to transfer data from RICNU Plan onto a desktop computer for analysis.

Among the most popular removable flash memory devices are USB Flash drives and SD Cards. Both behave very similarly, and the most important differences amount to the form factor. A microSD socket and a USB A connector would use very similar board space, but a USB drive would occupy a lot of space outside of the board edge, while a microSD card could be fully retained within the board margins.

3.2.3.7 Final Decision

Considering that RICNU Plan may be mounted in a size-restricted enclosure on a prosthetic device, the smaller form factor of a microSD card is preferable. Furthermore, modern microSD cards can reach up to hundreds of GB in capacity, which is more than plenty for this application.

The unavoidable disadvantage to using a removable flash memory of any kind is high power consumption during memory access operations. However, when not needed, a microSD device can be put in a standby mode, reducing power consumption significantly.

3.2.3.8 System Interactions

SD Cards feature two communication interfaces – Serial Peripheral Interface (SPI) and Secure Digital Input Output (SDIO). SPI is more widely available to a variety of microcontrollers, but it only features one communication line. SDIO, on the other hand, is not well documented but offers 4 parallel communication lines.

In this application, fast write rates are not required. Taking as an example the calculations in Section 3.2.3.2, 128-byte packets would be generated at 250Hz, and optimal write operations for high-capacity

SD cards require 512 bytes to be written at once. So, a block of data would have to be written to the SD Card once per 4 packets, or once every 16ms. Assuming an average “busy” times for the SD card could reach up to half of that frame, 512 bytes of data would need to be sent to the SD card every 8ms. The lowest acceptable writing rate is then 64 Kbyte/s, which is easy to achieve with SPI communication. An argument against using the SDIO interface is that it would take more data lines to achieve, and so the microcontroller’s I/Os that could be used for other purposes would have to be used for SDIO unnecessarily.

3.2.4 On-Board Programming Interfaces

3.2.4.1 Requirements

To do any useful work, a processing unit has to receive computation instructions from a memory unit. To update firmware stored in a microcontroller’s non-volatile memory, it has to be programmed using a communication interface, which varies among the different MCU architectures. Often, components and traces related to programming occupy a significant portion of PCB area and force the designer to make significant sacrifices in terms of both PCB size, BOM, and routing complexity. The following sections consider different methods that can be used to program STM32 microcontrollers and the BT121 module, used for wirelessly interfacing with a hand-held device. Optimally, these programming interfaces should:

- Occupy as little board space as possible
- Make it easy for the end-user to perform device firmware update (DFU) procedures, preferably using graphical tools on a desktop computer
- Minimize programming errors

3.2.4.2 Programming the Main Microcontroller

In general, STM32 devices can be programmed by either an in-circuit programming method (ICP) or various in-application programming (IAP) methods. In-circuit programming is useful for debugging as well as upgrading, while IAP is used in products that only need a DFU.

In-circuit flash programming for ARM Cortex-M microcontrollers is done via Serial Wire Debug (SWD) interface. Briefly, SWD is simply an ARM Cortex-M version of the widely-used JTAG protocol; SWD uses less interface pins, but its functionally similar. Besides a bi-directional data pin and a clock pin, SWD offers an optional signal trace output interface via a third line. To use SWD, the developer needs to connect an SWD debugging device between the host computer and the microcontroller, as shown in Figure 10. Besides simple DFU process, SWD also offers debugging capability.

Figure 10: Programming STM32: SWD

In-*application* programming methods utilize special bootloaders to update the device firmware using hardware communication peripherals available on the microcontroller, such as UART, CAN, or USB.

To be able to program a microcontroller using an IAP method, the chip must first be flashed with the appropriate bootloader using an in-circuit method (SWD in case of ARM Cortex-M microcontrollers). ST does this for all of their STM32 chips – the factory pre-programs its STM32 devices with a UART-based bootloader before release. For ARM Cortex-M devices, the selection between SWD- and UART-based DFU is done by manipulating a special-function pin (BOOT0).

This type of IAP requires the developer to put a USB-to-Serial interface connected between the USB interface of the host computer and the UART pins of the microcontroller, as shown in Figure 11.

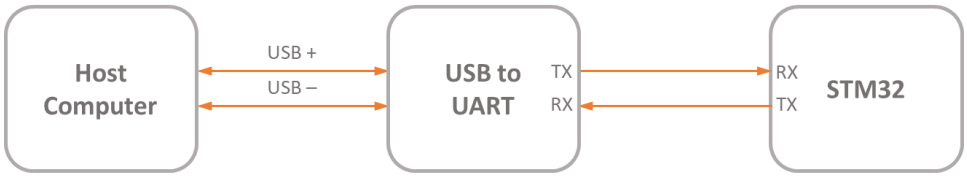


Figure 11: Programming STM32: UART bootloader

STM32 devices furnished with USB On-The-Go (OTG) capability can also be pre-programmed with a USB-based bootloader. This is great for applications where USB is already in use for data exchange, and in instances where firmware security is not a concern. This method does away with having to break out programming-specific MCU pins. As shown in Figure 12, no intermediate hardware is necessary.

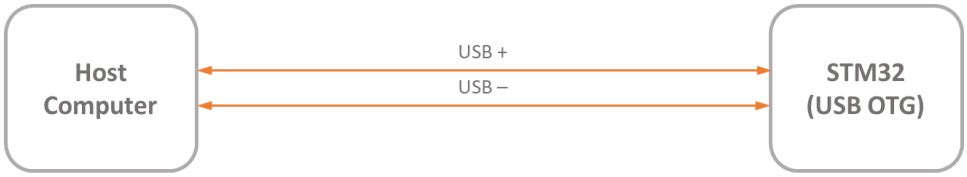


Figure 12: Programming STM32: USB Bootloader (USB OTG only)

With all of this in mind, which of the three would be best for RICNU Plan's microcontroller? The first version of RICNU Plan board is intended to be not just a research and active-use device, but also a development board for the device's firmware. As such, the debugging capability of the SWD protocol is essential. As unfortunate as it is for the board's form-factor, an SWD interface to a stand-alone external debugger should be present on the board. This, however, also means that the selected microcontroller does not have to be compatible with the USB-OTG protocol (and the selected STM32F103RD is not). As firmware becomes more stable and developed, future work on the project may entail migrating to a USB-OTG microcontroller and exclude the SWD interface from the printed circuit board.

3.2.4.3 Bluetooth Module

The DFU process of the BT121 module can be accomplished in three ways.

At its core, the BT121 module is an ARM Cortex-M0 processor, and hence can be programmed using the same methods as the main STM32 microcontroller on the board. The user may choose to use either SWD or UART to flash the chip.

However, this is where similarities end. The debugging features of the SWD interface cannot be used here because BT121 is pre-programmed by Bluegiga with proprietary firmware. As mentioned before, this firmware can interpret Bluegiga's BGScript language and BGAPI commands. In a way, it can be considered to be a rudimentary operating system – it takes care of the direct hardware control on its own,

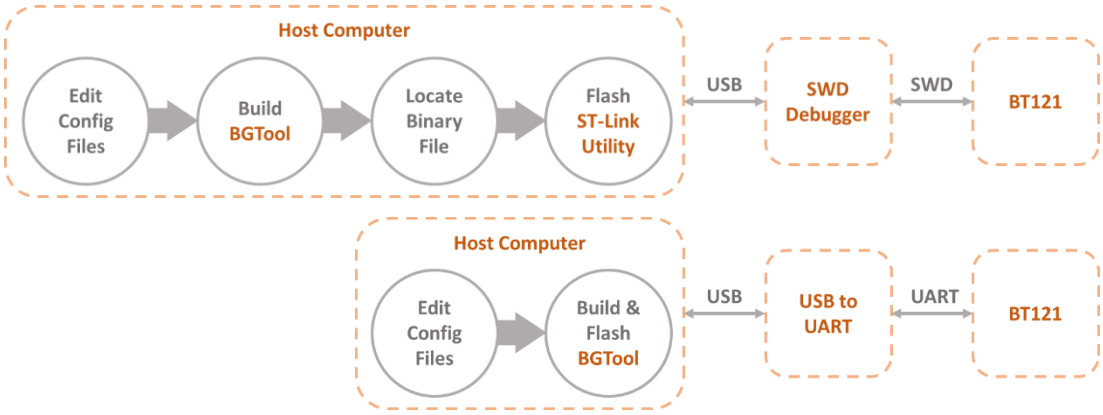


Figure 13: Programming BT121: SWD (top) vs UART (bottom)

while providing the user with the ability to control the operation of the device externally using a high level "language" protocol.

With this in mind, the programming process takes on the form of building a binary file from multiple BGScript and XML configuration files using Bluegiga's free BGTool software and uploading this binary into the chip's Flash memory using either SWD (using a separate tool called ST-Link Utility, from STMicroelectronics) or UART (with the same BGTool software). The options are shown in Figure 13.

From the end-user's perspective, using the desktop-based BGTool software to both build the firmware and program it into the module is much faster, easier, and less prone to errors.

Finally, aside from using BGTool to interface between a desktop computer and the chip directly, the host microcontroller can be used to program the chip using the BGAPI protocol, also via UART. In that case, the MCU is used as the primary programmer, and has to have the BGAPI stack *and* the full firmware binary for BT121 installed in its own Flash. This unnecessarily wastes memory.

Out of the three options, the ST-based UART DFU via BGTool offers the easiest setup for the user. BGTool is a graphical interface that takes care of all of the low-level steps in the process and offers the user the ability to program the chip "at a click of a button" [23].

To use the UART-based DFU method for programming BT121, communication between the host computer and BT121 must be translated from USB to UART. This can either be accomplished using the board's main microcontroller, or a discrete USB-to-UART interface IC.

Doing this on the main microcontroller would require development and appropriation of a full-speed USB stack, in addition to the actual USB-to-UART translation mechanism, not to mention the fact that in order to use the USB peripheral on the STM32F103 devices, an external high-speed oscillator is required. Having a dedicated USB-to-UART device taking care of this decreases the computational and memory burden on the main microcontroller. The designer is then faced with another decision – to design a USB-to-UART interface on the board, or make the researcher use an off-the-shelf external translator.

3.2.4.4 Final Decision

To summarize the above, it was determined that the board design needs to accommodate:

- SWD-based programming/debugging interface for the STM32 microcontroller

- UART-based programming interface for the BT121 Bluetooth module, using a USB-to-Serial translator (on-board or off-board) to interface with the user's computer

3.2.5 Power Supply Design

3.2.5.1 Requirements

The RICNU Plan board is intended for use in research settings primarily. This means that it needs to accommodate a variety of powering schemes and user errors, while at the same time providing the user with an easy way to switch between benchtop testing and untethered experimentation.

Whether an external power supply or battery power is available or not, the board needs to be powered using a desktop computer for development of control algorithms and benchtop testing. For this, a USB interface is very useful, as it can provide a steady 5V supply. As mentioned before, a USB connection is also needed to re-program the BT121; using USB for power would serve both purposes.

For quick transitioning between benchtop code development and untethered experimentation, the user should be able to connect the board to a desktop computer *and* a battery/external source at the same time. Upon disconnect from the computer-sourced USB power, the system should work without a power-out interruption. So, a USB port should be available *in addition* to an external supply input. This feature comes with the problem of supply arbitration. Both inputs must be furnished with backflow protection, or selected in a mutually exclusive, "break before make" fashion. When USB power is connected, the board should select it over the external supply to conserve device battery energy.

Low power consumption being one of the ultimate requirements for the project, RICNU Plan utilizes standard 3.3V logic in all of its digital circuits and should be able to work on at least 3.3V supply. Thus, the board should accept and selectively arbitrate two power supplies, one of which can be a computer-sourced or improvised USB bus, and another – a 3.3V-5V source, which can be a low voltage battery, 5V

output of FlexSEA Execute, a desktop power supply, or another source. After selection stage, the board must condition the supply down to 3.3V for use by the rest of the on-board circuits.

Finally, the first version of RICNU Plan is not a commercial device. Quite the contrary – its primary purpose is to serve in lab settings and as a development board for the continuing improvement of the design. It is inevitable that the board will be mishandled. Mistakes will be made - power will be occasionally plugged in backwards; battery voltages unthinkable for this application will be used, and even the USB input may experience improper external wiring and out-of-spec abuse.

In summary, the RICNU Plan's power design must:

- Be compatible with USB 2.0 and another external supply in the range of 3.3V-5V
- Select USB power preferentially, in case both supplies are connected
- Protect both of the external power supplies from short circuit conditions and current backflow
- Protect internal circuitry from overvoltage conditions in the range of up to 20 V, reverse power installation, and ESD events per IEC 61000-4-2 standard
- Condition the selected input supply down to 3.3V for use by the board's internal circuits
- Source up to 500mA of peak current

Figure 14 shows a simplified graphical representation of the required power system.

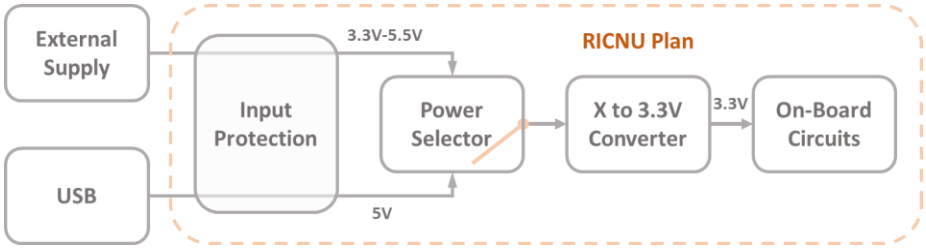


Figure 14: General Power Design

3.2.5.2 Power Selection

To achieve the dual-supply requirements above, the circuit must provide backflow protection to both supplies. In addition, the design should conserve as much battery power as possible.

One way to isolate the two supplies from one another is to put regular diodes in series with the two supplies in an arrangement is shown in Figure 15.

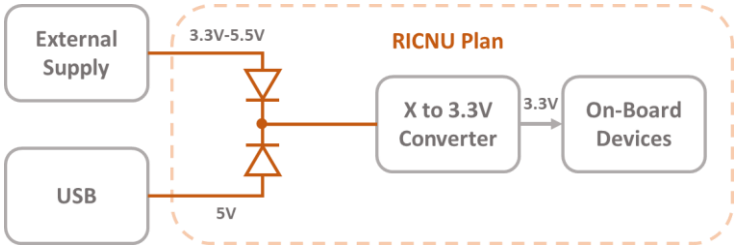


Figure 15: Backflow protection diode arrangement

This indeed provides adequate back-flow protection but results in unnecessary waste of energy. To illustrate, let's assume that only the battery supply is connected. Assuming total board power consumption reaches 250mW (maximum power budget for default mode operation), at 5V the board would draw 50mA. An average forward voltage drop over a standard diode is about 0.7V. Without even thermal considerations, the power dissipated as heat by the diode would then be $P = IV = 50mA \times 0.7V = 35mW$. That is *14%* of the already-limited budget – completely unacceptable. At lower voltages, the power waste will increase further.

An alternative would be to use Schottky diodes in the same configuration. Some of them can have much lower forward drops of 0.2V and lower. This decreases heat dissipation, but only down to ~10 mW, wasting 4% of the budget. To add to that, Schottky diodes usually have much higher reverse leakage currents than standard diodes, which partially defeats their purpose as backflow protection in the first place. Most importantly, these designs allow for both supplies to be drained with no selective control.

Without delving into complicated analog designs, an alternative to this is to use a designated power multiplexer (PMUX). The TPS211x power multiplexers from Texas Instruments are specifically designed to switch between two power supplies in the range of 2.8V-6V. These devices perform supply selection

preferentially, based on external circuit setup. These PMUXs utilize current limiting, manual/automatic switching modes, and status output functionality in various combinations.

The specific device chosen, TPS2113, has a switching $R_{ds(on)}$ of only 84 mΩ (typ.), and boasts an operating current of only 55µA. Taking the example above, at 50mA of current drawn by the rest of the system and 5V supply, power dissipation at this stage is only $P = 5V \times 55\mu A + 50mA^2 \times 84m\Omega = 0.485\ mW$, which is only 1.9% of the power budget. Also, TPS2113 was chosen because it handles switching automatically based on a voltage appearing on one of its pins – no external control is necessary. The chip also offers a current-limiting function and a status output. TPS2113 can be setup in a way that when USB power is available, only USB is used to power the device, even when the secondary power supply is connected. When USB is not connected, the secondary power supply is used, as would be the case during patient experimentation.

3.2.5.3 Buck Converter

As mentioned above, RICNU Plan is a 3.3V board in its entirety. However, because the design can be powered by USB (and in general, 3.3V – 5.5V), the board design is using a step-down power supply.

In designs where low power consumption is not a requirement, linear low-dropout regulators (LDOs, or DC linear regulators) are used because they are very easy to set up and do not introduce any noise into the supply line. However, they are highly inefficient, dissipating energy as heat, and thus they also require thermal design features.

A highly-efficient alternative device is a switch-mode power supply (SMPS). Although they cost more than linear LDOs, some claim efficiencies in the mid-90% by boosting or bucking input potentials using high-frequency switching circuits.

The selection of an appropriate supply was made using the WEBENCH Power Designer software from Texas Instruments. Using various parameters as inputs, the database selects appropriate SMPS devices based on preferences on efficiency, cost, component size, and others. The selection was made by choosing a stable 5V as an input (whether USB or from FlexSEA Execute), 3.3V as output, and up to 500mA of output current; among various suggestions made by WEBENCH, TLV62568 was chosen to power the board because of its small size, high efficiency, and low cost.

3.2.5.4 High Power Device USB Enumeration

USB 2.0 protocol states that a USB host may only provide up to 100mA of current, with an absolute maximum of 500mA [24]. Theoretically, a USB device can just ignore the 100mA software limit and try to draw up to 500mA from the host, but this may not work on all host systems. Instead, to be able to draw more than 100mA, a USB device must present itself to the host as a High-Power USB device.

For this, one can either design a USB enumeration stack as part of the board's firmware or use a dedicated device to perform this function. Connecting the dots further, a USB-to-Serial converter is needed for UART-based IAP of the Bluetooth module.

Fortunately, FTDI Chip makes a device that can perform both functions, FT232R (among others). Besides translating incoming USB communication into UART, it also performs USB enumeration when powered on, and can be programmed to display itself as a High-Power USB device.

3.2.6 User Interface Features

The required functionality of this first version of RICNU Plan is two-fold: to perform in patient testing, and to perform as a development board for the RICNU Plan's firmware. Visual indication and mechanical switches for hardware control over the board are important to have at this stage in the design.

3.2.6.1 Visual Indication

As a high-level control board, RICNU Plan must indicate to the user its state. At minimum, the user should know whether everything is working normally, whether an error has occurred, whether the board is logging data or not, and whether the board is currently asserting active control on the prosthetic device. This indication can be achieved with, at minimum, two LEDs (both off, both on, on/off, off/on). However, these four states are not mutually exclusive; for example, the device will be working normally while asserting active control and data logging at the same time.

The indication should be natural to the user. For instance, blue or green colors are usually associated with "normal operation" in most devices. Red, on the other hand, is associated with erroneous operation. Data logging and active control can be indicated by other colors. Four most common LED colors were chosen – blue, green, red, and yellow.

3.2.6.2 Mode Control

Let's refer back to Sections 3.2.2.6 and 3.2.4.3, dealing with system interactions and programming of the Bluetooth module, respectively. It becomes clear that both processes use the module's only UART peripheral, mostly for similar reasons. It is then apparent that there must be a mechanism to arbitrate the control of the bus between the board's main microcontroller and the USB-to-Serial converter.

This can be achieved in a variety of ways. The user can have a double-pole-dual-throw (DPDT) switch that connects and disconnects the RX and TX lines of the Bluetooth module between the microcontroller and the USB-to-Serial converter, or the switching can be done using a tri-state bus driver.

It is also important to remember that in order to be programmed via the UART interface, the Bluetooth module's BOOT0 pin has to be pulled up while its reset line pulled low.

In the DPDT switch scenario, the board would have to have two other mechanical or electrical switches to control those lines. The bus driver, on the other hand can be controlled *together* with the BOOT0 pin by a single-pole-dual-throw (SPDT) switch to achieve the same functionality but requiring less actions from the user.

Besides this circuit, both the Bluetooth module and the main microcontroller need to have mechanical interfaces to their reset lines, so the user can reset them at will. This can be achieved with push-buttons.

3.2.7 Communication Interfaces

In step with the FlexSEA system, RICNU Plan should be able to communicate with a variety of peripherals for complex system implementations. Per the requirements listed in the introductory Section 2.4.1, Plan has to communicate with FlexSEA Manage and RICNU User at the minimum, with support of a variety of other communication interfaces, as well.

3.2.7.1 SPI with FlexSEA Manage

The communication with FlexSEA Manage happens via SPI, as defined by FlexSEA. The SPI bus on FlexSEA Manage is accessible through a single-row 8-pin connector from the Molex Pico-Clasp series. Not all pins are required for the SPI bus; Manage is also powered and can be reset through this connector. All-in-all, only 6 lines are required – the regular 4 SPI lines (chip select, master-in-slave-out, master-out-slave-in, clock), ground, and reference voltage for Manage to translate logic correctly. Hence, only a 6-pin connector is needed on RICNU Plan.

To maintain compatibility, the same line of connectors was chosen – Molex Pico-Clasp. It would require the same wire crimps and wire crimping tool as the connectors on Manage and Execute; the user would not have to purchase any other tools.

3.2.7.2 Bluetooth Communication

A wireless transceiver is necessary to communicate to the RICNU User application. However, in multi-limb systems, wireless communication between two RICNU/FlexSEA systems may be required.

This can be achieved in multiple network topologies. For an example, let us assume the system consists of RICNU User and two RICNU Plan/FlexSEA Manage/FlexSEA Execute trees.

The communication can occur between the two control trees without involvement of RICNU Plan. In that case, one of the RICNU Plan boards has to take over the role of the server, and another – of the client, including initiating connection, and the like. With Bluetooth LE or BR/EDR as the wireless technology, this topology would be hard to implement, as both trees would also need to communicate with RICNU User as servers.

A great solution to this is to maintain a star topology, with RICNU User as the center node. With Bluetooth v4.1 specification, it is possible to connect two devices to a host at the same time. RICNU User can analyze data and state information between the two devices and generate downstream commands appropriately.

3.2.7.3 Controller Area Network (CAN)

Due to its robustness against external noise and simple physical link implementation, the Controller Area Network protocol is widely used in automotive applications and is ideal for industrial environments where multiple devices have to communicate with each other. It is also used on all RIC/SRALab electronic boards for the same reason. For compatibility, a CAN interface was considered to be a logical addition to the design.

3.2.7.4 RS-485

Communication between Manage and Execute happens via asynchronous half-duplex RS-485, which requires a single twisted data line pair and a ground reference. In systems where Manage is simply used as a data-translator, FlexSEA Manage should not be necessary, as it takes unnecessary volume and area over the prosthetic device. By designing an asynchronous half-duplex RS-485 interface on RICNU Plan, the design would make it possible for FlexSEA Execute to communicate with RICNU Plan directly. RS-485 is a very robust and widely-used protocol, and it can also be used to communicate with other user-defined devices.

3.2.7.5 Other Serial Communication Protocols

Following the requirements in Section 2.4.1, the board must be compatible with a variety of serial communication interfaces, including USART, extra SPI busses, I2C.

One option would be to assign certain microcontroller pins to certain peripherals and have separate connectors built for each. However, that destroys concept of flexibility for this system.

Instead, the chosen microcontroller STM32F103RD, like other devices in its class, is able to map its 51 I/Os to different communication peripherals based on its firmware. This is very useful, as the same pins can be used for different peripherals at different times (for example, there are multiple pin pairs that can behave as either I2C busses or UART busses). Also, instead of having an individual connector for each interface, the design in this case would include a large multi-pin connector that can serve various functions based on user's intent. This concept follows from the FlexSEA Manage and Execute expansion connectors, which feature multiple pre-set and configurable I/Os connected to the boards' main microcontrollers. By using the same series (but different number of circuits) dual-row Molex Pico-Clasp connector as both FlexSEA boards, system compatibility would be supported, as well.

3.2.8 Device Protection

The RICNU Plan board is sure to be misused in a lab environment. Whether a tired engineer plugs in the power in reverse, or a researcher unfamiliar with the system tries to plug the prosthesis' battery straight into the board, RICNU Plan must withstand these events, as well as electrical stresses on its communication inputs and outputs.

3.2.8.1 Power Input Protection

Among other things, the design must clamp the input voltage at a safe level for the TPS2113 power multiplexer (-0.3V to 6V), provide fault-safe reverse polarity protection, and direct ESD pulses away from the board's components. Discussing all of the options that would satisfy these requirements is out of the scope of this thesis; however, for detailed description of the design, refer to Section 3.4.2.

3.2.8.2 Microcontroller I/O Protection

The board's microcontroller is the single point of contact for all inputs and outputs on the board, except for the USB and non-USB power connections. Besides ESD events endangering internal circuitry of the main microcontroller and the USB-to-Serial converter, their I/Os must also be protected from conflicting logic configuration and general misuse. Each I/O must be protected from overvoltage and overcurrent, besides ESD and surge events.

3.2.9 Other Considerations

A complex system like the RICNU/FlexSEA system is prone to run into grounding-related issues. Multiple boards sharing multiple ground return paths can introduce noise into the system and surrounding environment. On a prosthetic device, grounding can be implemented in various ways. For instance, everything can be referenced to the battery's negative terminal, while nothing is connected electrically to

the prosthesis itself. Or, a metallic shell of the prosthesis can serve as a ground reference for all components and can serve a heat sink for the system.

For this purpose, RICNU Plan has been designed to have a single return path to the prosthetic device through a mounting hole on the board. Two mounting holes are used (for minimum mechanical stability): one plated, and one non-plated. The plated mounting hole would provide a singular return path to ground if the rest of the system components are not connected to the metallic shell electrically. This way, no ground loops are introduced. Otherwise, if there is another ground connection to the prosthesis somewhere else, the plated hole can be interfaced with a plastic screw.

3.3 Software Design Considerations

RICNU Plan microcontroller firmware structure underwent a studious decision-making and consideration process. The following system-level components of the firmware were considered in detail:

- Choice of primary language and development tools
- DCFSM implementation
- Memory card interface
- Task scheduling and priority
- System relationship with the Bluetooth module
- Version control and Open Source Initiative

3.3.1 Choice of Primary Language and Tools

RICNU Plan's microcontroller is a powerful ARM Cortex-M3 processor and can run complicated algorithms and even some interpreted languages. But, this application requires strict low-level hardware control, and only low-level languages were seriously considered as candidates. At the same time,

extremely low-level assembly was not considered for its low portability across different platforms. The two languages most widely used in embedded systems other than assembly are C and C++, which is an object-oriented extension of C. Either one could have been used for this application, but it was decided that object-oriented software was not necessary for this application, and so the board's main microcontroller has been programmed in C99.

Among the various integrated development environments built for debugging STM32 devices, only free and open-sources tools were considered, in step with the open-sourced nature of the project. Drawing on prior experience, the author opted for using IAR Embedded Workbench for STM32 Kickstart Edition, which has all of the major debugging capabilities required. However, its code-size restriction came into play towards the end of the project, and the development switched over to the Atollic TrueSTUDIO for STM32, which is a free Eclipse-based tool that offers even more professional capabilities without a code-size restriction.

3.3.2 Device Control Finite State Machine

RICNU Plan asserts high level control on the FlexSEA system using a DCFSM. Referring back to Section 1.2, the DCFSM is a complex finite state machine that defines various discrete behaviors of the system for real-time prosthetic control. It can be implemented using multiple techniques.

For one, it can be a hardcoded part of RICNU Plan's firmware. Each time the state machine would need to be changed, the microcontroller could be flashed over and over again. It is immediately apparent that this type of implementation would be not only cumbersome for the user but would also decrease the lifespan of the microcontroller's flash memory.

Another way to implement this is purely in the graphical user interface. The researcher could use the RICNU User application to define the DCFSM and send the configured structure over a certain API to

RICNU Plan. While this would be ideal for the user, this process would involve complex point to point networking and detailed app development; both are out of the scope of this project and academic degree.

As a compromise, the DCFSM could still be configured externally on a desktop computer using a markup language or a GUI and then ported over to RICNU Plan while the device is already running. RICNU User, in turn, can be used to adjust the various settings of the DCFSM during normal operation.

The porting process could take multiple forms as well. For one, RICNU Plan could feature a "bootloader" module, which would download the DCFSM information from the desktop computer to the microcontroller using one of the board's communication interfaces (including Bluetooth). Another simpler solution would be to use the microSD card already present on the board for data logging purposes as a medium for the DCFSM file hosting.

While the former would have been a nice solution from the user's point of view, it was decided to follow through with the latter due to the project's short schedule. Perhaps in future versions of the RICNU system, DCFSM configuration could happen wirelessly via a Bluetooth link with a desktop computer.

3.3.3 Memory Card Interface

Modern microSD cards feature two communication interfaces – Secure Digital I/O (SDIO) and Serial Peripheral Interface (SPI). SDIO is a parallel data communication interface, while SPI only has one data line. Consequently, for a given clock speed an SDIO interface can achieve much higher data transfer rates than the SPI interface. However, following the discussion in Section 3.2.3, it was decided that the low requirement on communication speed did not justify the effort to achieve communication with the card using the SDIO interface. Instead, SPI was used.

Importantly, SD cards employ a special protocol on top of the physical SPI link. The host device (RICNU Plan) has to send commands and the card takes some time to respond. These busy states can range from

microseconds to tens of milliseconds. For lossless data logging capability, it was decided to implement long-term log entry buffering in the microcontroller's memory and to design the data logging process such that it functions asynchronously to other tasks.

General compatibility with common devices is highly desired in this application. SD Cards fall into three capacity families (standard, high, and extended), and within multiple speed classes within those families. The RICNU firmware was developed to accommodate for these various classes to offer a single interface to any SD card available in the laboratory.

3.3.4 Task Scheduling and Preemption

RICNU Plan is part of a prosthetic control system and is required to offer real-time computational and communication performance. As new data becomes available from downstream control sources, it must be promptly unpacked, processed, and used in high-level control decision making. Fortunately, the FlexSEA system offers a relatively stable data re-fresh rate of 250Hz at maximum. All of the data processing and communication tasks may happen in a cyclical fashion, repeating at each consecutive 4ms frame. However, as explained in the previous section, data logging can be a slow and non-deterministic process, which directly opposes the repeating nature of all the other tasks.

One solution to this is to design a full operating system for task scheduling. The project could use open-source software like FreeRTOS to schedule all of the communication and data processing tasks, but the cyclical nature of the required processes did not justify spending development time on implementing a full operating system. Instead, another solution is to use the microcontroller's own interrupt controller and assign preemption priorities to all tasks other than data logging such that they preempt the data logging process and retain their deterministic cyclical nature. So, it was decided to handle all communication and processing tasks besides data logging in various interrupt handlers, triggered by timer and communication interface interrupts.

Another important consideration to make is the CPU's involvement in communication procedures. Embedded systems without tight schedule constraints may opt for data transferring using the CPU itself; more time-constrained systems usually employ other means of data transferring, allowing the CPU to tend to other tasks. To maintain the real-time performance of the system, communication-related data transfers are offloaded to the microcontroller's direct memory access (DMA) peripheral.

3.3.5 Bluetooth Module

RICNU Plan hosts two processing systems – the board's main planning microcontroller and a Bluetooth SoC, which also employs a processor. Each of these components performs drastically different tasks. The goal of the Bluetooth module, for instance, is to maintain a BLE link with the RICNU User application and route data between the FlexSEA system and RICNU User. The goal of the board's main processor is to maintain high-level control over the prosthetic device. It was decided that one must not affect the other's normal operation. For instance, a reset experienced by the board's main processor should not break the Bluetooth link between Plan and User, and communication events experienced by the Bluetooth module should not disturb real-time control asserted by the main processor on the prosthetic device. So, it was decided to design the system in such a way that the two processors only share a protocol-less, asynchronous serial communication link. Although not great for data integrity (absence of acknowledgements and data clocks), this type of interface is ideal for decoupling the two subsystems.

Also, to maintain high data throughput to RICNU User, it was decided that the communication protocol between Plan and User should be simplified as much as possible and reduced down to raw data. This way of data transferring requires both Plan and User to perform exactly the same processing procedures on the raw FlexSEA data, but it allows for much greater data throughput.

3.4 Hardware Design

This section is intended to be a detailed technical reference on the hardware design of the RICNU Plan 1.0 board. Rationale behind major technical decisions is kept to a minimum, as it is mostly covered in the previous section. Each sub-circuit is explained in theory and supplemented with hardware layout information, as necessary. Appendix C contains all of the hardware schematics. The Quickstart document included in online documentation for the project explains how to easily interface with the board, and Appendix D: Bill of Materials can be referenced for specific component models.

3.4.1 Hardware System Overview

Figure 16 provides the complete system-level diagram of the board's hardware, including hardware interfaces to external subsystems like debugging devices and the FlexSEA boards. Besides digital signals, the diagram also shows the power supply architecture. Within the RICNU Open Source Leg ecosystem, FlexSEA Execute uses the battery power to supply both the FlexSEA Manage and RICNU Plan boards.

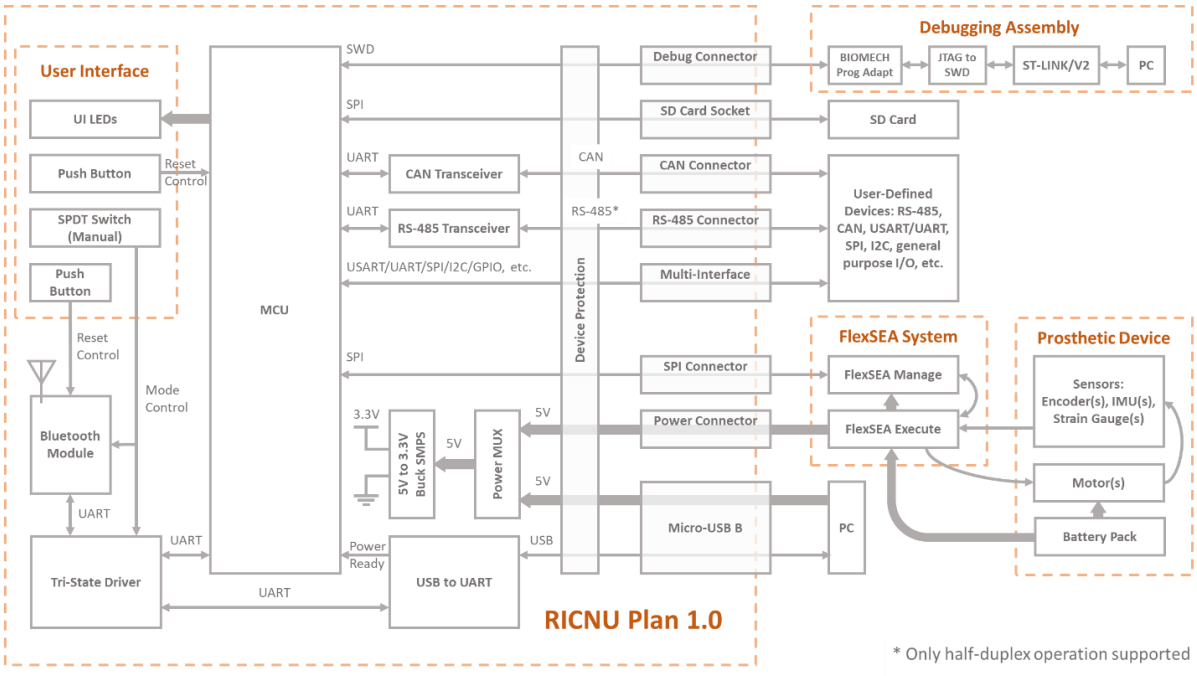


Figure 16: Hardware System Diagram

Figure 17 displays the physical locations of all of the major components on the RICNU Plan Board. The layout has been organized by functional areas. For instance, the top left corner of the board (Figure 17, top) is dedicated to power inputs and conditioning, and the bottom edge of the board is dedicated to interfacing with other devices. RICNU Plan is a four-layer board, designed in a Signal-Ground-Power-Signal configuration (top to bottom layer).

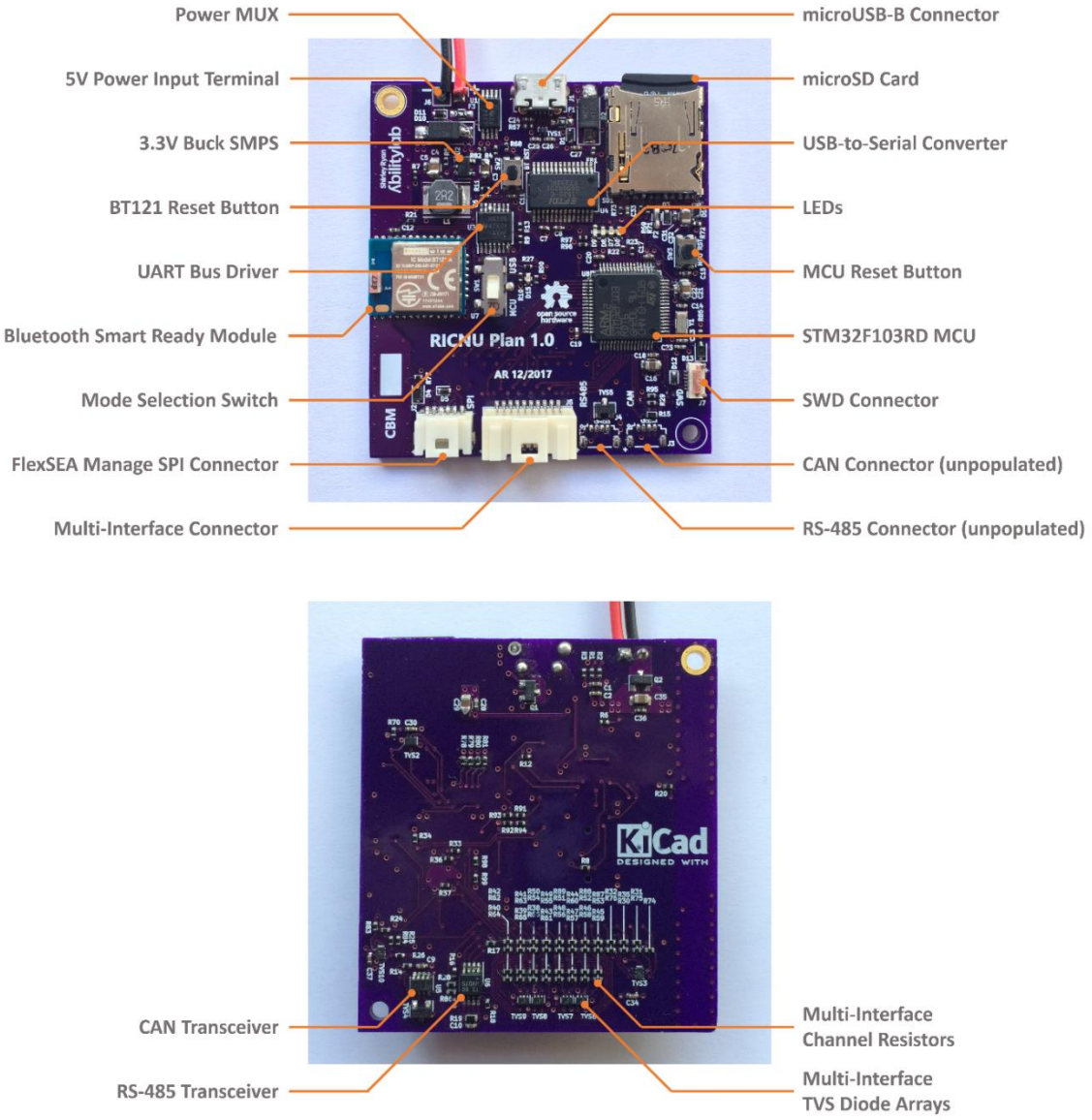


Figure 17: Front (top) and back (bottom) views of the RICNU Plan Board

3.4.2 Power Stage

RICNU Plan offers two inputs to supply the board. The user can choose to power it either via USB 2.0 (microUSB-B cable), or with another user-defined 5V supply. The input stage designs for both inputs are very similar, with some additions to the USB input stage.

3.4.2.1 Non-USB Power Input

Figure 18 presents the input protection circuit of the non-USB power input. Starting at the supply side, a resettable fuse F3 (FEMTOSMDC020F-2 from Littelfuse) protects the external supply from on-board overcurrent conditions (and vice versa). The fuse is rated at hold current of 200mA and trip current of 450mA; Within that range, the time it takes for it to trip depends on its thermal environment. Besides brief current spikes generated by the SD Card and the Bluetooth module, the board normally never consumes anything close to this much current. The fuse becomes active only in case a large voltage is applied across the input terminals, or if a catastrophic component failure occurs either on the board or externally. Otherwise, the fuse's low resistance generates a voltage drop of only ~0.1V even in worst-case scenarios (high current consumption, multiple prior fuse trip events).

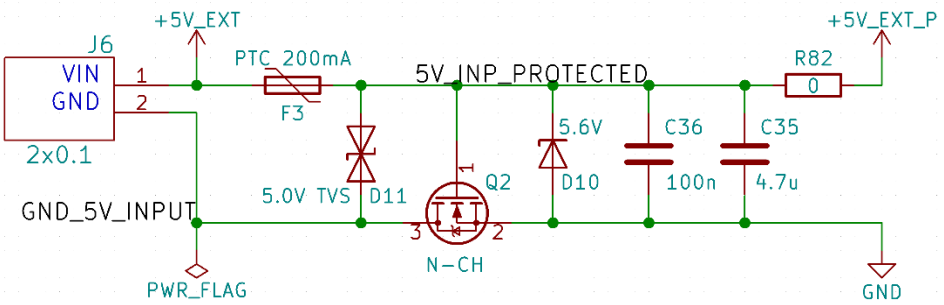


Figure 18: Non-USB power input

A low-capacitance bi-directional TVS diode D11 (PESD5V0S1BB,115 from Nexperia) provides IEC 61000-4-2 Level 4 ESD protection (contact ±8kV, air gap ±15kV) and IEC61000-4-5 surge protection to the rest of the board. The board has not been tested for ESD and surge survivability, but this device should provide ample protection. The diode has a typical breakdown voltage of 5.5V to 9.5V and can be used to clamp input voltages at 14V (per datasheet), but D11 is not used for this purpose here – only for ESD and surge protection.

The low-R_{dson} N-Channel MOSFET Q2 (SI2312CDS-T1-GE3) provides the rest of the board with effective reverse-polarity protection. In this design, its gate is connected to the input potential, its drain – to supply ground, and its source – to the board's ground. When a negative voltage is applied across the terminals of the input, the MOSFET finds its gate at a lower potential than its source, and almost no current is let through. So, the board is safely disconnected from the power supply *by default*. To turn on SI2312CDS fully, V_{IN} must reach a threshold of, typically, 0.45V or higher. When supply is at full 5V, MOSFET's R_{dson} reaches only a couple of mΩ, which results in a fraction of energy loss when compared to series diode/Schottky diode designs.

A 3W 5.6V Zener diode D10 on the source side of Q2 protects the next stage from overvoltage conditions. When the input voltage exceeds its Zener voltage, the diode starts limiting voltage by shunting current; shortly after this current exceeds the I_{hold} of the PPTC fuse F3 (200mA), fuse limits the current as well. Together, D10 and F3 safely limit the voltage seen by the next circuit stage to no higher than 5.89 V with inputs of up to 20V. In fact, the board is able to operate normally with inputs of up to 15V; after that, the limited current is not guaranteed to power the SD Card and on-board MCU devices.

Capacitors C36 and C45 function as bypass capacitors, and the 0Ω resistor R82, when not populated, is used to isolate the input stage from the rest of the board for testing purposes. The protected power output (+5V_EXT_P) is fed into the power multiplexer stage described in Section 3.4.2.3.

Figure 19 shows the PCB layout of the non-USB power input stage. The non-USB power connection is made via a 0.1” standard DIP-style connector, if desired. The PPTC fuse F3 and the bi-directional TVS diode D11 have been placed as close to the 0.1” thru-hole connector as possible. As a result, ESD and surge events should safely traverse the small loop formed by J6 (input connector), F3, and D11 without coupling to other components – all on the top layer.

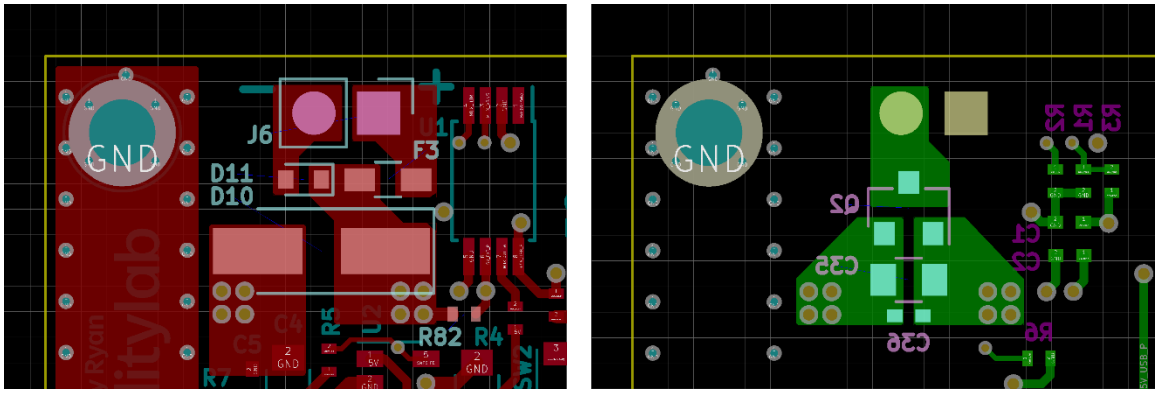


Figure 19: non-USB input stage layout
Left: top layer, right: bottom layer

Zener diode D10 is placed close to the fuse, as well. The intention here is that as the diode heats up while shunting current during an overvoltage condition, the fuse will heat up as well by proximity, and its current-limiting properties will come to life faster than if it was far away from the diode. Copper pours on both terminals are good heat sinks; the ground pour is stitched to the ground layer (layer 2) with four large vias, which further enhances the heat sinking capability.

MOSFET Q2, placed on the bottom layer close to the input connector, isolates the external supply ground away from anything else. Having D10, C35, and C36 in parallel (along with the gate and source of the MOSFET) presented a great design opportunity to have them all share stitched power and ground copper pours on the top and bottom layers. While providing heat sinking functionality, this also reduces the space taken by the circuit.

3.4.2.2 USB Input

Figure 20 presents the input protection circuit built for the USB input. The USB bus is used to communicate with the USB-to-Serial converter IC from FTDI Chip; for best performance, this design was developed with reference to the *USB Hardware Design Guidelines for FTDI ICs* (AN_146) document developed by FTDI.

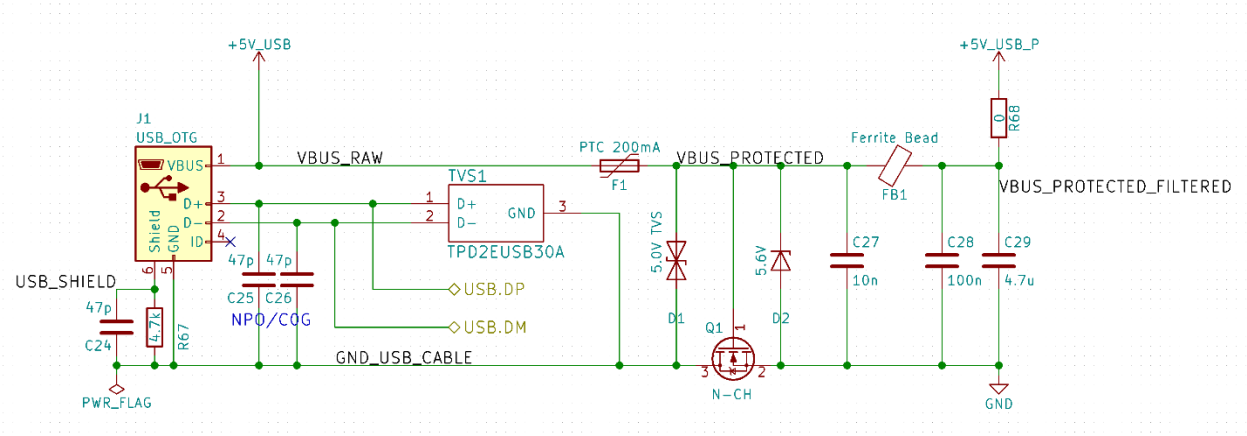


Figure 20: USB input

The design of the input protection circuit here is very similar to that of the non-USB input, with an addition of supply filtering and choking accomplished by a combination of a 10nF capacitor C27 and a 30mΩ DC/10Ω@100mHz ferrite bead FB1 (per recommendation from AN-146). To save board space, a microUSB-B connector (Molex 105017-1001) is used in the design. The metallic shield of the connector offers four thru-hole pins that robustly anchor the connector in place on the PCB. Working with the FlexSEA boards, it became apparent that simple surface-mount connectors often do not withstand repeated use.

Conveniently, according to the USB 2.0 protocol, the USB host can provide, at maximum, 500mA of current on VBUS. The same fuse used on the non-USB input is used here to accomplish effective current limiting way below the 500mA threshold. Because the supply is 5V, other components are also the same.

The D+ and D- inputs are protected from ESD events by a specialized TVS diode array from Texas Instruments specifically designed for USB applications, TPD2EUSB30DRTR. Just like PESD5V0S1BB,115, this device protects the USB data lines according to the IEC 61000-4-2 Level 4 and IEC61000-4-5 standards, but it offers exceptionally low line capacitance (on the order of 0.7pF), which is essential for USB communication. The design also has 47pF NP0/C0G capacitors (C25 and C26) on the differential data lines, again, following recommendations from FTDI regarding recommended line capacitance for their FT232R chip [25]. Finally, the board ground is connected to the USB shield by a 4.7kΩ resistor and a 47pF capacitor [26].

The above circuit layout is shown in Figure 21. Just like in layout of the non-USB power input, the PPTC fuse and TVS diode (in this case, F1 and F1) are placed close to the connector and the current travelling through them has a direct path to input supply ground. A MOSFET Q1 is placed in the vicinity as well, to isolate any events that may happen on the USB ground. USB supply filtering is accomplished by FB1, C28, and C29 (bottom left corners). The resulting filtered supply is channeled to the rest of the board by a thick 0.5mm trace (bottom). The differential data traces are length matched and pass through the two terminals of the TPD2EUSB30DRTR diode array (TVS1). C25 and C26 are also mirrored across the differential bus. C24 and R67 provide short AC and DC paths to USB ground for events happening on USB shield (cable or connector shell).

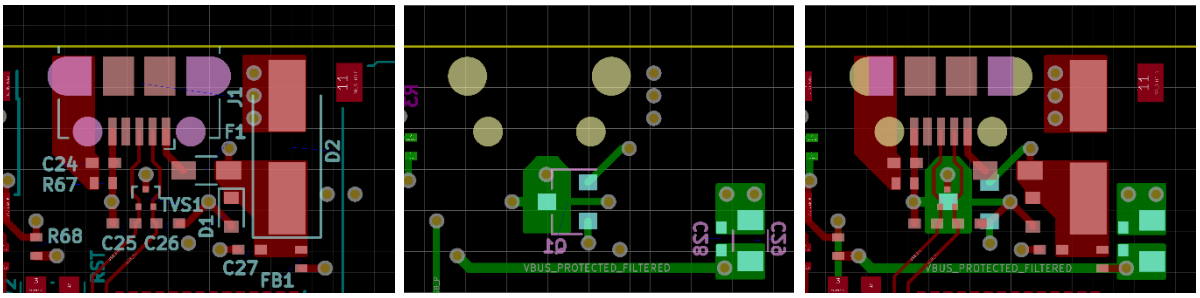


Figure 21: USB Input Stage Layout
Left: top layer; middle: bottom layer; right: both layers

3.4.2.3 Power MUX and Buck Converter

The two power inputs described in Sections 3.4.2.1 and 3.4.2.2 are arbitrated and mutually protected from each other by a power multiplexer, TPS2113. This device selects one of the input channels and routs it to a 3.3V buck converter, TLV62568. This power conditioning circuit is shown in Figure 22. The power MUX circuit is set up in such a way that TPS2113 preferentially selects the USB power input over the non-USB input, even if both are present at the same time.

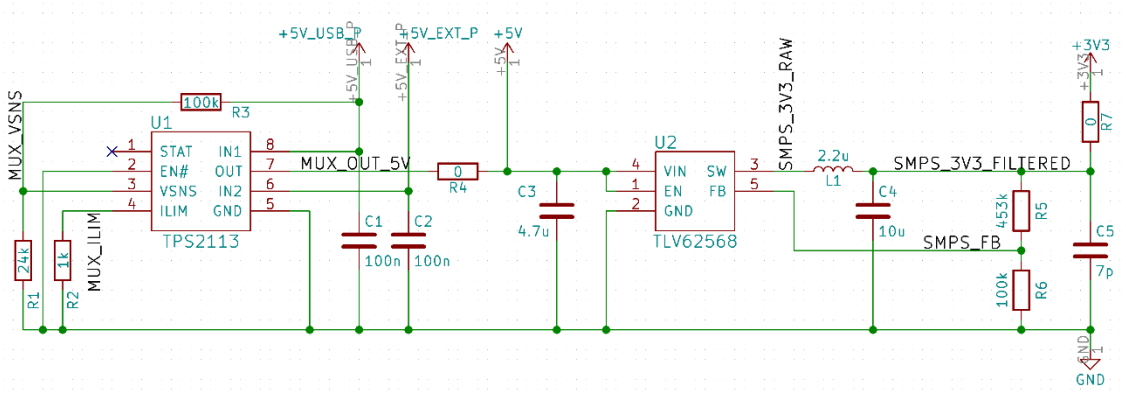


Figure 22: Power MUX and 3.3V Buck

According to the truth table from its datasheet seen in Figure 23, if potential at VSNS exceeds 0.8V and #EN is low, TPS2113 chooses the supply at present at IN1. In the circuit, #EN is tied to ground, and IN1 is the USB VBUS line. Resistors R1 and R3 form a voltage divider between the VBUS and ground; the resulting voltage is fed into the VSNS. So, when USB is present, VSNS exceeds 0.8V, and TPS2113 selects the USB input, even if the non-USB supply is present; otherwise, the non-USB supply is chosen. C1 and C2 are used as simple decoupling capacitors. The status pin on TPS2113 can be used in future designs to indicate supply availability. R4 is used for testing purposes.

TRUTH TABLE

$\overline{EN}$	$V_{I(VSNS)}$ > 0.8V	$V_{I(IN2)}$ > $V_{I(IN1)}$	STAT	OUT(1)
0	Yes	X	0	IN1
	No	No	0	IN1
	No	Yes	Hi-Z	IN2
1	X	X	0	Hi-Z

(1) The under-voltage lockout circuit causes the output OUT to go Hi-Z if the selected power supply does not exceed the IN1/IN2 UVLO, or if neither of the supplies exceeds the internal V_{DD} UVLO.

Figure 23: TPS2113 configuration truth table

Even though TPS2113 does not normally disrupt power supply when switching between the two inputs, a 4.7uF bulk capacitor is used on the input stage of the TLV62568 buck converter as a precaution.

The output stage of TLV62568 has been designed in accordance to the datasheet recommendations for this device. A combination of a 10uF bulk capacitor and a 2.2uH, low-ESR inductor is a standard recommendation for most applications. Testing has shown that this design results in a ripple of only ±60mV, with a peak-to-peak maximum of ±80mV at 3.3V output – that is far from causing any sort of brown-out conditions for either the microcontroller, the Bluetooth module, or even the supply-sensitive SD card. The voltage divider formed by R5 and R6 simply forces the converter to output 3.3V. R7, just like R4, is used for testing.

The hardware layout of the power conditioning stage is presented in Figure 24. To save board space, all TPS2113-related passives (R1, R2, R3, C1, and C2) are situated on the bottom layer, as they are simply configuring the chip or providing extra decoupling. On the other hand, most of the TLV62568-related components are placed on the top layer along with the buck converter itself.

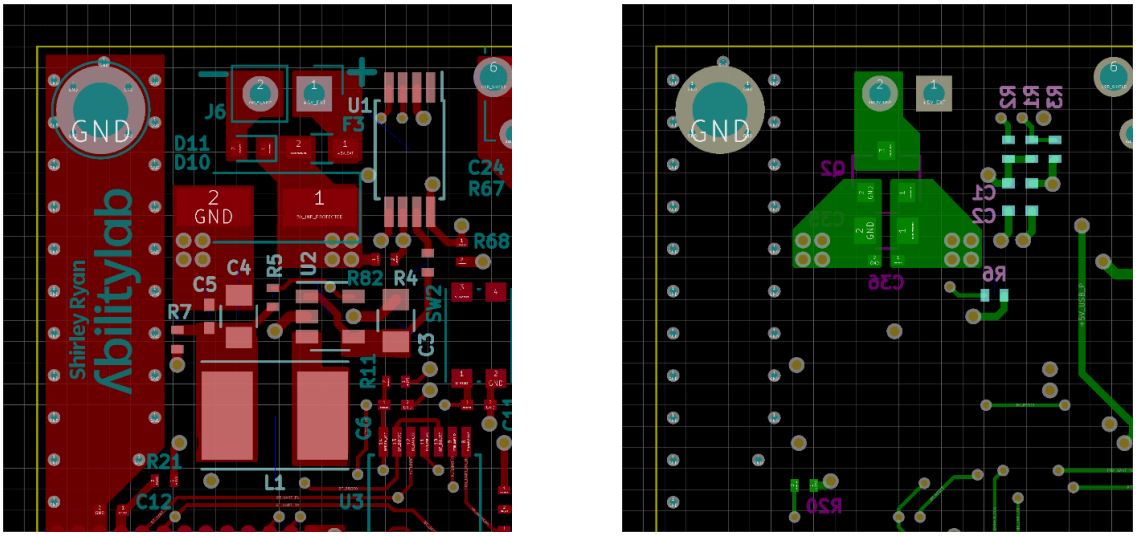


Figure 24: Power MUX and buck converter layout
Left: top layer; right: bottom layer

The layout of the TLV62568 circuit was designed in accordance to a layout recommended in its datasheet. To minimize the area of the high-frequency output node of the TLV62568 module, the large filtering inductor L1 is placed as close to the output pin as possible. Bulk and decoupling capacitors C4 and C5 are in the vicinity as well and share a convenient copper plane with the inductor. The two capacitors also share the ground pour used by the Zener diode protecting the non-USB input stage.

The total power stage layout is shown in Figure 25; both input stages, multiplexing, and conversion happens within the area marked with a yellow dashed line. The design makes sure that the power stage occupies as little space as possible, only carries power traces in one isolated area (the 3.3V output of this stage is fed through 0Ω R7 to layer 3, which is a dedicated 3.3V power plane. The four 0402 resistors highlighted in blue are the 0Ω resistors used to isolate each stage for testing. Their proximity and open placement makes it easy for the user to solder and de-solder these by hand. A wall of vias in the ground plane used for BT121's antenna reference prevents any noise coming from the power stage from reaching the signal reference ground.

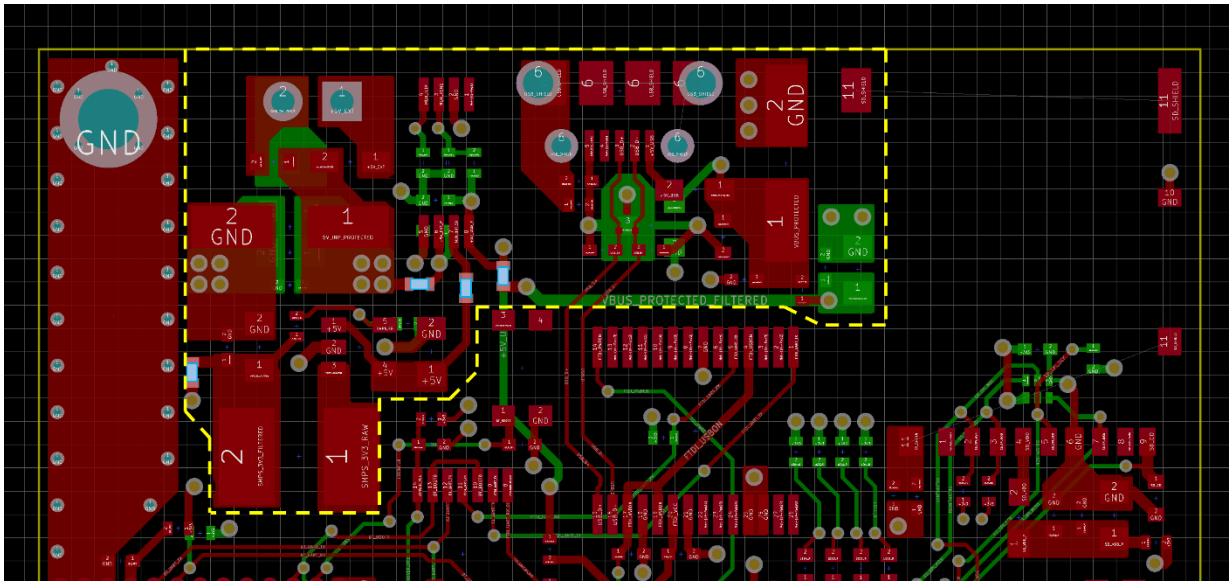


Figure 25: Power stage layout area

3.4.3 Main Microcontroller and Related Circuits

The main microcontroller used on RICNU Plan is an ARM Cortex-M3 microcontroller from ST Microelectronics, STM32F103RD. ST markets its STM32F1 devices as “Mainstream Performance line” microcontrollers. While the design could have easily used their low-power ARM Cortex-M3 series, STM32L1, having a CAN interface emerged as one of the requirements in the design. Looking at MCUs comparable or slightly more advanced then the L1 line, the STM32F103 series emerged as the victor. The particular microcontroller chosen, STM32F103RD has 384 kB of flash memory and 64 kB of SRAM.

3.4.3.1 Microcontroller Pinout

The 64-pin package of the microcontroller is utilized to the fullest extent – only five pins (three of which are special pins) are not used. Figure 26 shows the full pinout of the microcontroller.

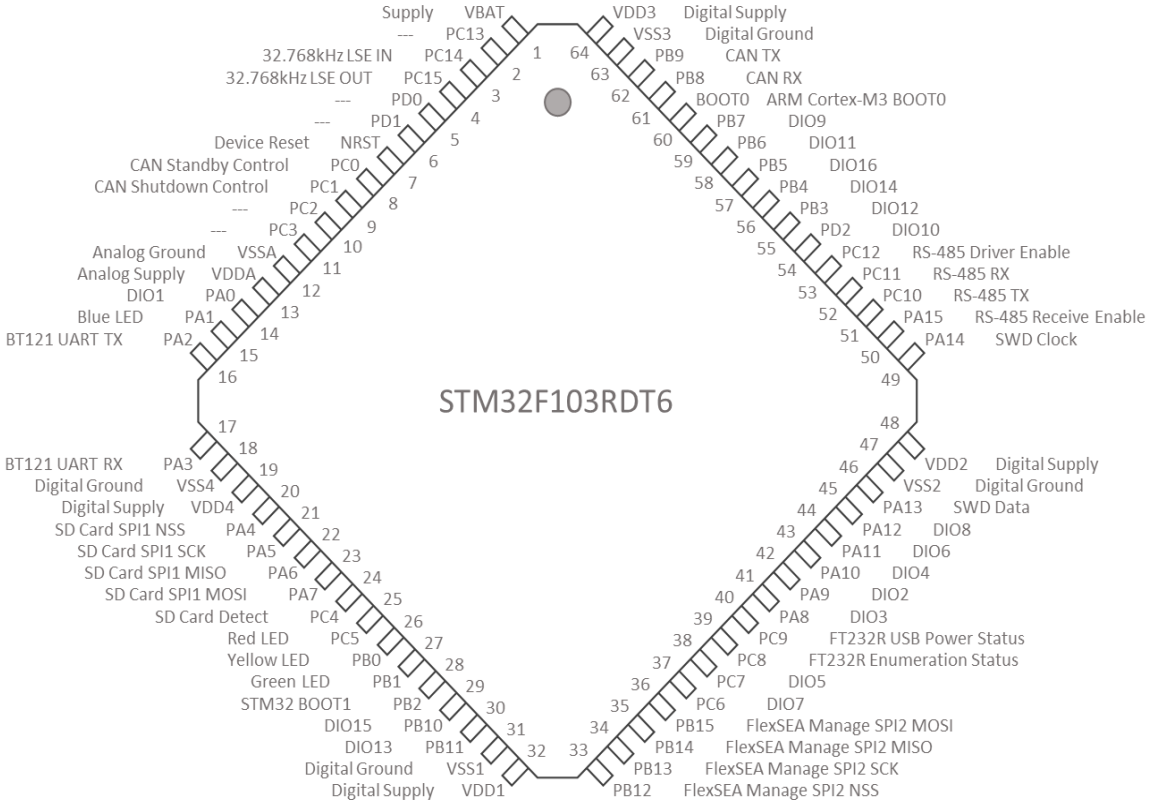


Figure 26: Microcontroller Pinout

3.4.3.2 Microcontroller Setup

The MCU uses a high speed internal (HSI) 8MHz RC oscillator for its system and peripheral clocks. An external 32.768 kHz crystal is used to clock the real time clock (RTC) peripheral for real-time data logging, time keeping, and deep sleep functions. All supply/ground pin pairs are furnished with 100nF decoupling capacitors, and a 4.7uF bulk capacitor is placed between VDD3/VSS3, as recommended in the datasheet. No analog functions are used on this board, so the analog supply and ground pins are tied to digital supply and ground, respectively. Still, following the datasheet, analog supply pins are furnished with a 10nF/1uF capacitor combination instead of a 100nF/4.7uF array.

Most pins that are externally driven feature series 180Ω resistors, the only exception being the CAN RX line (470Ω instead). These pins are prone to being configured as output pins in software and can suffer overcurrent damage when the external drivers attempt to drive the shared lines to an opposite logic level. The resistors limit the resulting current to only 3.3V/180Ω = 18.33 mA (and in case of CAN, to 7.02mA), which is below the absolute maximum ratings of both the MCU and all of the external drivers it communicates with. The SPI MISO, UART RX, and most DIO lines are pulled up by 10kΩ resistors. This value is a somewhat of a compromise - when driven low, the resistors only pass 0.33mA of current, and at the same time do not affect digital signals at most usable communication speeds.

The MCU can be reset via SWD debugging interface or using the provided push-button.

STM32 MCUs come with a great feature for boot mode selection. The two boot pins (BOOT0 and BOOT1) can be used to choose between three different boot modes. The three conditions are summarized in Figure 27. Because this is an open-ended, flexible board, the two pins are furnished with both pull-up and pull-down resistors. The

Boot mode selection pins		Boot mode
BOOT1	BOOT0	
x	0	Main Flash memory
0	1	System memory
1	1	Embedded SRAM

Figure 27: STM32 Boot Mode Selection

default design has the microcontroller boot from flash – both pins are pulled low by 470Ω resistors, and the pull-up resistors are not populated.

The power supply layout for the microcontroller was derived from recommendations from the device's datasheet and common techniques for good microcontroller circuit design. Figure 28 shows the basic layout. Most of the setup components are situated on the top PCB layer – that includes decoupling and bypass capacitors, along with a 32.768kHz crystal oscillator and associated load capacitors. Because the microcontroller interfaces with a variety of signals, the MCU's power layout had to be carefully designed. The layout resembles a star arrangement. At the center, the 0Ω resistor R37 is placed on the bottom signal layer. It connects the 3.3V supply plane on layer 3 to an intermediate plane on layer 4. From there, multiple vias carry the current back up to layer 3 and up to the top layer via thick traces (shown in white in Figure 28. Bypass and decoupling capacitors are placed between the MCU's VDD pins and the supply trace vias to ensure that the 3.3V supply is actually bypassed. This star arrangement ensures that even though layer 3 (3.3V supply) is traversed by multiple branches, return currents connecting multiple areas of the board can still flow freely through the MCU area on layer 3.

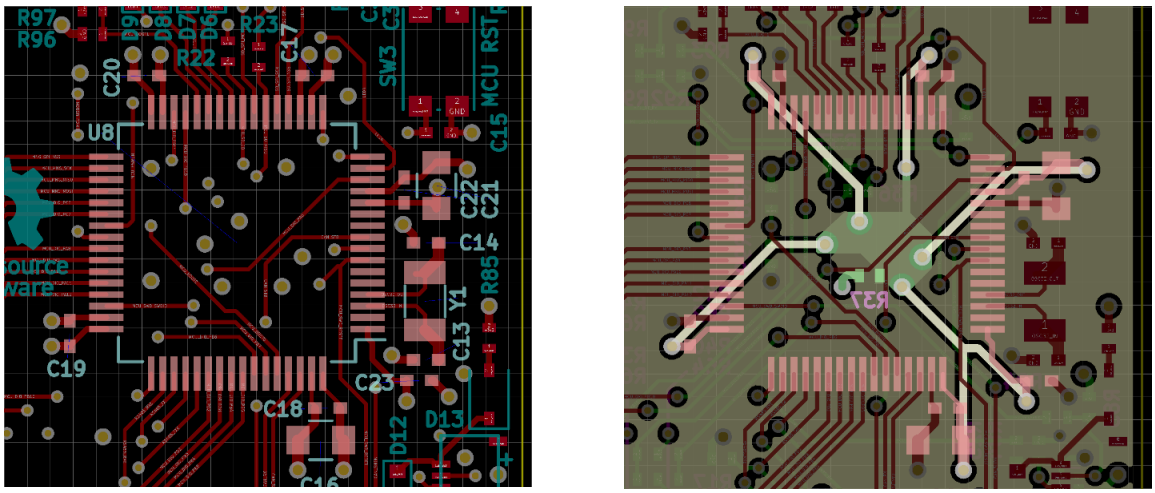


Figure 28: STM32F103RDT6 layout

Left: top layer component layout; right: power supply layout; green – layer 4 (bottom signals), light yellow and white – layer 3 (3.3V supply), red – layer 1 (top signals)

3.4.3.3 SWD Interface

The microcontroller can be debugged and re-programmed using the ARM Serial Wire Debug (SWD) interface, a close relative of the popular JTAG interface. SWD only needs a clock and a data line to perform most of the programming and debugging functions. The ST-Link/V2 debugger used to program the microcontroller also needs a reference supply voltage, a reference ground, and access to the reset pin of the MCU. Optionally, a Serial Wire Output (SWO) pin can be used for tracing internal signals; it has not been made accessible on RICNU Plan 1.0, as it is not essential for most debugging purposes. Figure 29 shows the circuit implementation of the SWD interface.

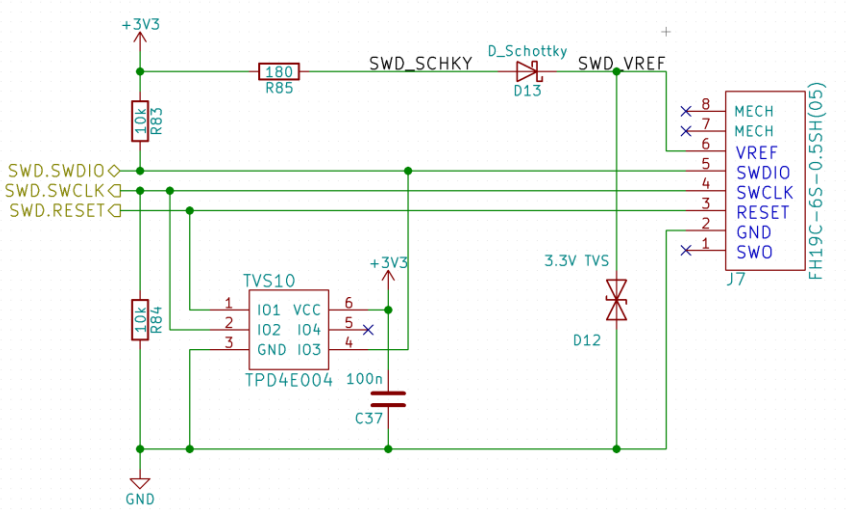


Figure 29: SWD Interface

Besides the already-familiar bi-directional TVS diode on the reference voltage, the design includes a low-forward-drop Schottky diode to protect the board from supply backflow. A 180Ω resistor is used to limit the current consumed by the debugger in case it malfunctions. The clock line is pulled down, and the data line is pulled up – both to define the logic on these lines when the debugger doesn't drive them. The SWDIO, SWCLK, and RESET lines are all protected from ESD events using a TPD4E004 TVS diode array. This device claims ±8-kV IEC 61000-4-2 Contact Discharge and ±12-kV IEC 61000-4-2 Air-Gap Discharge protection. Connected to 3.3V, it clamps voltages between -0.8V and +4.1V for the three lines.

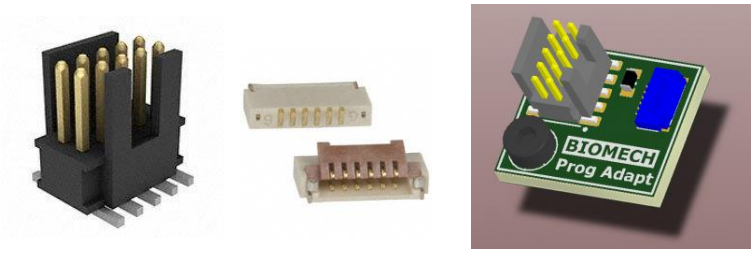


Figure 30: Connectors and Adapter for SWD
10-pin 0.05” (left), 6-pin FPC (middle), FlexSEA ProgAdapt (right)

SWDIO and SWCLK pins of the MCU are 5V tolerant (see datasheet), and all three pins are protected by series 180Ω resistors.

Usually, SWD utilizes a 10-pin 0.05" connector, shown in Figure 30 on the left. However, the FlexSEA boards use a 6-pin FPC connector from Hirose, FH19C-6S-0.5SH instead (Figure 30, middle); it is much smaller than a standard Cortex SWD connector. RICNU Plan would be used together with the FlexSEA system, and the same connector has been used in this design. FlexSEA boards come with a small 10-pin-0.05”-to-6-pin-FPC adapter called FlexSEA ProgAdapt (Figure 30, right).

3.4.3.4 LED Indication

RICNU Plan uses four LEDs for software-defined indication. They are all different colors (blue, green, yellow and red), and each is turned on when the microcontroller starts sinking current on a respective pin. During hardware design validation, it became apparent that the intensity balance between the four LEDs has not been properly equalized (see Chapter 7: Future Work).

3.4.4 Bluetooth Module and Related Circuits

The Bluetooth module used on RICNU Plan is the BT121 SoC from Bluegiga, now owned by Silicon Labs. The main microcontroller on the board uses BT121 as a UART passthrough port – it uses the module to transmit packets of serial data to a mobile device application wirelessly.

The module can be reprogrammed in-application via the board's USB port. For this procedure, the board utilizes a USB-to-Serial converter chip, FT232R, and a serial bus driver, needed to arbitrate the UART bus between the FT232R and the main microcontroller. The selection is chosen by the user via a mechanical slide-switch.

Figure 31 may help the reader visualize the data flow associated with the Bluetooth transceiver. USB data is transmitter to the FT232R USB-to-Serial converter; the resulting UART packets are transmitted to BT121 via the bus driver. When the slide switch selects the USB data route, data is received by BT121 from FT232R. Otherwise, BT121 receives data coming from the microcontroller.

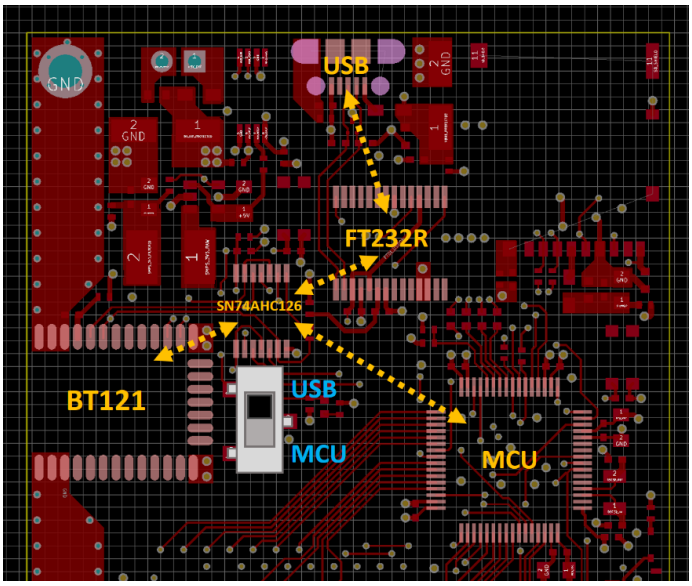


Figure 31: BT121 - data flow and mode selection

3.4.4.1 Bluetooth Module

While the module has a variety of serial communication peripherals, only its USART peripheral is used, asynchronously and without flow control. Figure 32 shows the circuit setup for the module. The only pins used besides supply and ground are UART RX and TX, BOOT0, and RESET.

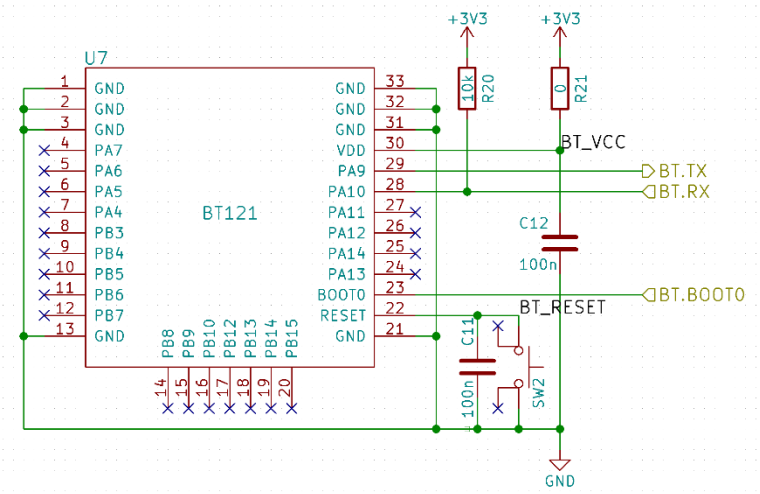


Figure 32: BT121 circuit setup

The UART RX line is pulled up with a 10kΩ resistor R20, and a decoupling 100nF capacitor C12 is used merely as a precaution, because Bluegiga indicates that the SoC includes all of the appropriate decoupling and bypassing components on-board.

The module is based on an ARM Cortex M0 MCU; to boot the device into UART-based IAP mode, the chip's RESET pin needs to be pulled low as the BOOT0 pin is held high. According to the datasheet, BOOT0 is internally pulled low and RESET is internally pulled high, resulting in normal operation. So, no PU/PD resistors are needed. Switching both lines is accomplished using user-manipulated mechanical switches. A normally-open push-button SW2 pulls RESET to ground when pressed, and BOOT0 logic level is controlled by the slide switch SW1 shown in Figure 37, Section 3.4.4.3.

While the circuit is simple, the challenge comes with the hardware layout. For optimal communication range and signal strength, Bluegiga recommends having ample (presumably, component-free) ground plane area around the chip (Figure 33).

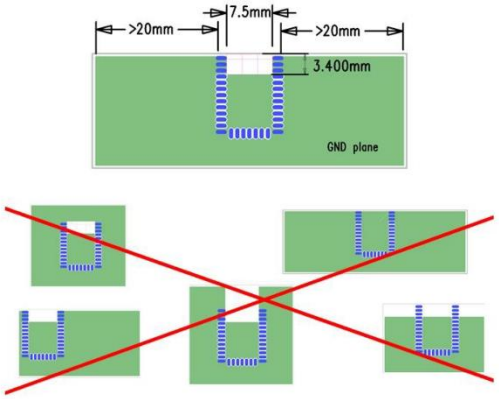


Figure 33: Ground Plane Recommendation

That comes with its own challenges – one of the goals of the design is to make the PCB as small as possible, but the datasheet confusingly shows a component-free ground plane around the entire chip. But, since the antenna reference ground only needs to align with the antenna itself, the design shown in Figure 34 achieves excellent RF properties nonetheless.

The left part of Figure 34 shows the antenna reference planes on either side of BT121. They are both stitched to the internal ground plane (second layer from the top, shown in middle part of Figure 34). Notice that all layers respect the antenna's keep-out area, following Bluegiga's recommendations. The right part of Figure 34 shows the third layer – 3.3V copper pour. The decision to keep the 3.3V power away from the reference ground plane area vertically was based on the fact that no components actually need the 3.3V supply in those areas. One may notice that the large mounting hole in the top left of the PCB is plated and connected to board ground. It provides a single ground reference for all on-board signals when the board is mounted onto a prosthetic device.

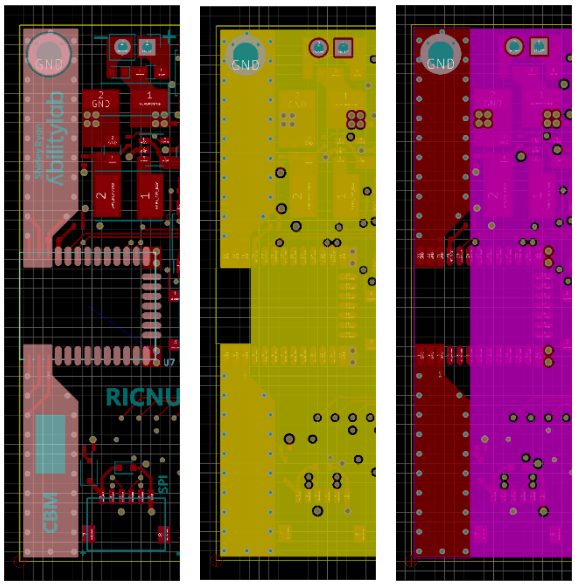


Figure 34: BT121 layout
Left: top copper (reference GND planes and BT121 pads); middle: layer 2 (GND); right: layer 3 (3.3V)

3.4.4.2 USB-to-Serial Converter

UART-based IAP of the BT121 module is accomplished via a USB connection to the user's computer. To aid with translation between USB and UART, the board incorporates a USB-to-Serial converter, FT232R from FTDI Chip. On the host end, it acts as a COM port device and provides an easy interface to BT121.

Figure 35 shows the circuit setup for the FT232R IC. While the chip has extensive functionality, only its USB, UART, 3.3V output, and the PWREN capabilities are used in the design. The circuit setup for this device was heavily referenced from the device's datasheet.

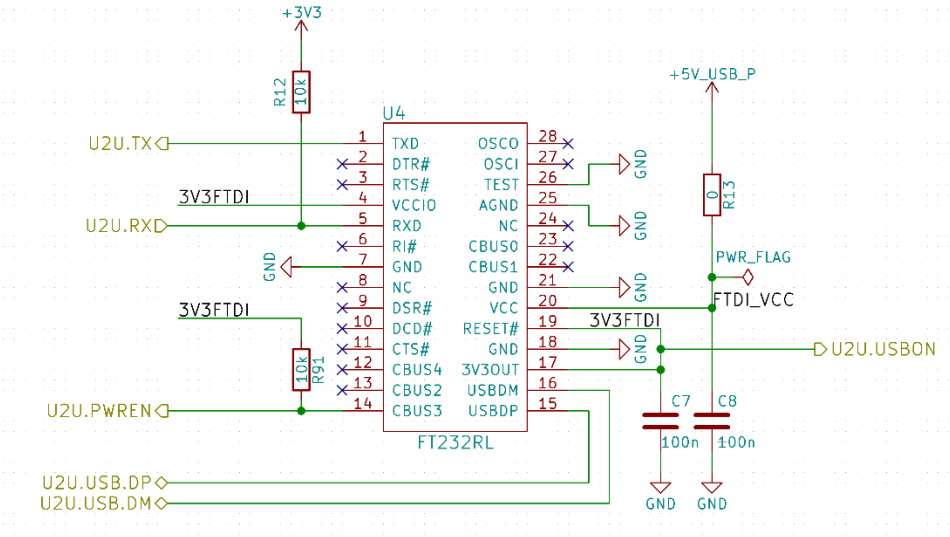


Figure 35: FT232R Circuit Setup

In this first version of the board, this chip is only used to program BT121, and programing BT121 is only possible when the board is connected to a PC via USB. So, to conserve power when USB is not available, the chip is powered directly with the protected USB supply, *not* with through the power multiplexer. This way, when USB is not available, the chip is powered down and does not consume any power.

FT232R has an integrated linear 3.3V regulator that uses the 5V power as input. In this design, the regulator output is used to force the chip's digital logic to follow 3.3V logic levels (VCCIO is tied to 3.3V) and, thus, be compliant with the logic levels accepted by BT121. Since the chip should always be

ready to respond to USB communication coming from the user's computer, the reset pin of the device is tied to the output of the 3.3V regulator, and the device is always active when USB is present. The same output of the 3.3V regulator is also available to the main MCU as an indicator of USB power availability.

USB 2.0 protocol states that a USB host may only provide up to 100mA of current, with an absolute maximum of 500mA [24]. In case the board has to use auxiliary peripherals like CAN and RS-485, it may need more than a 100mA supply. Theoretically, a USB device can just ignore the 100mA software limit and try to draw up to 500mA anyway, but this may not work on all host systems. Instead, to be able to draw more than 100mA, a USB device must present itself to the host as a High-Power USB device.

FT232R can be configured to display itself as a High-Power USB device to the USB host. The chip's PWREN pin (by default, assigned to CBUS0) is low during USB enumeration, and high afterwards. This can be used to indicate to the main MCU that USB power is ready. Once PWREN is released or driven high, the MCU can enable the high-current devices without repercussions. Resistor R91 is used to pull the CBUS0 pin up externally so the MCU's power arbitration can work properly even when USB power is not available. Resistor R12 pulls up the UART RX line, and capacitors C7 and C8 are used for decoupling. Resistor R13, like other 0Ω resistors on this board, can be used for testing and circuit stage isolation. Following recommendations from FTDI, 47pF capacitors are put on the two USB data lines (Section 3.4.2.2).

3.4.4.3 UART Bus Driver

The SN74AHC126PWR quad tri-state driver array is used to arbitrate between UART communication coming from the board's main microcontroller and from the user's PC through FT232R USB-to-Serial converter. When BT121 is programmed, FT232R drives the module's TX line; otherwise, MCU takes control. The design makes sure that the two buses never interact with each other.

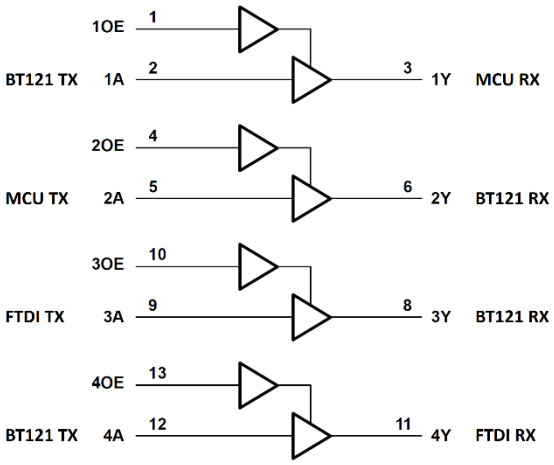


Figure 36: Tri-state buffer circuits

To be able to program the BT121 module without disrupting the MCU operation and vice versa, all four communication lines (MCU/BT RX+TX and FTDI/BT RX+TX) are controlled by tri-state buffers as shown in Figure 36. All output-enable (OE) pins are active-high. Buffers 1 and 2 relay data between BT121 and the MCU, while buffers 3 and 4 connect BT121 and FT232R. The full UART bus driver circuit is shown in Figure 37.

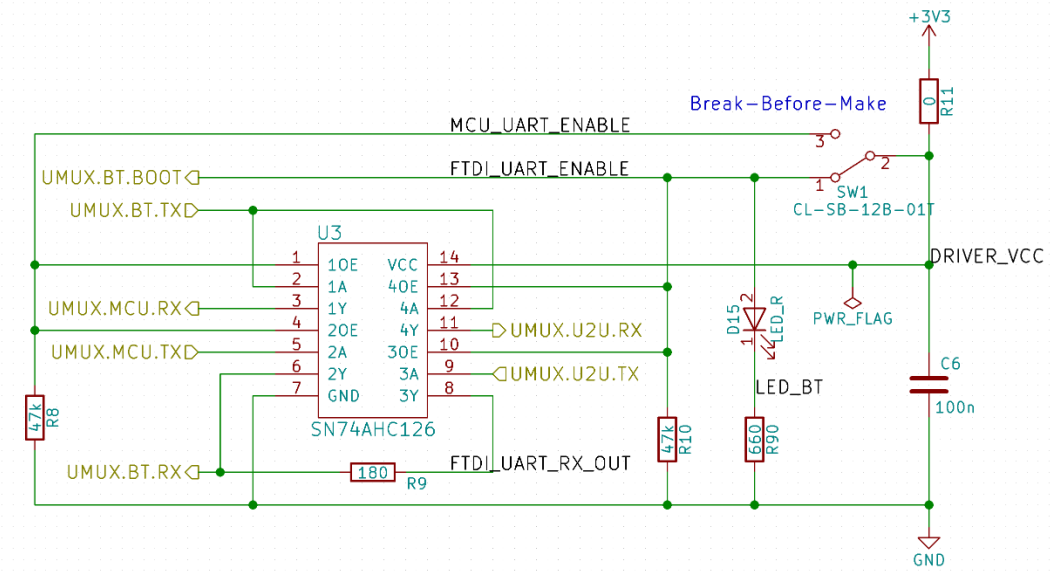


Figure 37: UART Bus Driver Setup

Bus selection is made using a mechanical slide switch SW1; in one position, it ties 1OE and 2OE pins of SN74AHC126 to 3.3V (making the MCU bus active), and in another – 3OE and 4OE (FTDI bus). Both pairs of selection pins are normally pulled low by R8 and R10. When the switch disconnects each pair from 3.3V, the buffers become inactive. Because the switch itself is a break-before-make (BBM) type, each pair of buffers turns off before the next pair is turned on.

Nonetheless, if the switch fails, mechanically or otherwise, the outputs of the two buses may become connected. In this case, when both devices try to drive their TX lines to different logic levels, R9 will limit resulting current flowing between pins 2Y and 3Y to 3.3V/180Ω = 18.33 mA, which is safely under the ±25mA maximum continuous output current rating of SN74AHC126.

In this first version of RICNU Plan, USB communication only has one function - reprogramming the BT121 module. As mentioned in previous sections, BT121 boots into programming mode when its BOOT0 pin is pulled high while the module is reset. This present a neat design opportunity – the slide switch that arbitrates the two communication buses is also used to pull BT121's BOOT0 pin high. This simplifies the programming procedure down to three steps:

1) Connecting PC to the board via USB
2) Sliding the SW1 to get the board into BT121-programming mode
3) Pressing the reset button for the BT121 module

To further ease the process, red LED (D15) lights up when the board is switched into programing mode.

3.4.5 SD Card

Data logging and finite-state machine specification are performed using an SD Card. To securely hold the card in place, the board has a push-push socket from Hirose, DM3AT-SF-PEJM5, which features a mechanical Card Detect feature. Figure 38 presents the setup circuit for the SD Card.

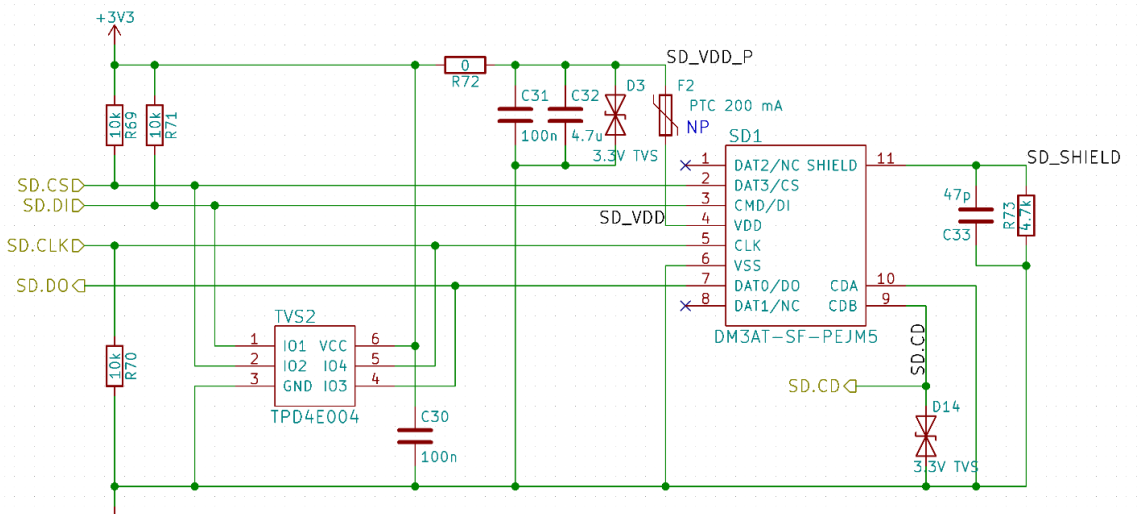


Figure 38: SD Card Socket circuit setup

All data lines are protected from ESD events using the same TPD4E004 TVS diode array. SPI Chip Select and MOSI lines are pulled up, while the clock line is pulled down to ensure proper SPI idle logic states.

SD Cards are notorious for their large power consumption in some high-speed read/write states. To prevent browning out the board's microcontroller and Bluetooth module, a 4.7uF ceramic capacitor is used on the supply line to the SD Card. A 3.3V TVS diode, PESD3V3S1UB,115, is used to protect the supply line from any ESD events originating from the card itself. It is a lower-breakdown version of PESD5V0S1BB,115 described in other sections.

The PPTC fuse F2 was originally designed on the power line to protect the circuit board from faulty cards. If the card itself develops a short-circuit path from VDD to card ground, the rest of the board may be negatively affected. However, testing has shown that the fuse raises the supply impedance to an unacceptable level for the card, and the SD Card is not able to get out of the Idle state when the fuse is populated. Besides, the supply input line fuses F1 and F3 would provide a similar protection mechanism anyway. A short-term solution for this is to bypass the fuse by soldering a bridge across its terminals, or

not populating the fuse at all – just a solder bridge between the dedicated PCB pads. The next design revision should eliminate the fuse from the design. The socket provides two card-detect connections – CDA and CDB. Upon card reception, CDA is connected to CDB. In this board design, CDA is tied to ground, while CDB is pulled up with a 10kΩ resistor. The microcontroller assumes the card is inserted when its Card Detect signal changes from logic high to logic low. Just like in the design of the USB input, the SD Card socket shield is connected to board ground through a 47pF capacitor and a 4.7kΩ resistor for the same reasons.

3.4.6 Communication Interfaces

RICNU Plan can be configured to use a variety of communication interfaces. While some are pre-defined, some can be chosen by the user at compilation time. For detailed information on pinouts and Plan-FlexSEA cables, please refer to the Quickstart document included in the online documentation.

3.4.6.1 SPI to FlexSEA Manage

Data communication with the FlexSEA system is done using SPI. For this, RICNU Plan utilizes a 6-pin Molex Pico-Clasp single row connector. FlexSEA Manage uses the same connector series, except an 8-pin version for some additional functions. The 6 lines used here are: reference voltage for FlexSEA Manage voltage translator, the four SPI pins (chip select, clock, MISO and MOSI), and a ground reference. The Manage protection circuit is very similar to the one featured in the SWD interface.

3.4.6.2 CAN and RS-485

For compatibility with other RIC/SRALab-developed boards and other applications, the board features a CAN interface and a half-duplex RS-485 interface. The connector of choice for both is a 4-pin DF-52 connector from Hirose, which is a simple, yet robust plug receptacle. DF-52 is used by some of the boards developed at RIC/SRALab and was picked out for compatibility reasons.

3.4.6.3 Multi-Interface

Keeping in step with the flexibility of the FlexSEA boards, RICNU Plan includes a multi-interface connector that can be used for USART, SPI and I2C communication, as well as other purposes like timer outputs, general digital input/outputs, etc. The multi-interface connector is a dual-row, 20-pin Molex Pico-Clasp connector (again, for hardware compatibility with the FlexSEA hardware). The middle four pins are all dedicated to ground connections for various interfaces. The other 16 pins have multiple functionality (see Figure 39and Table 1).

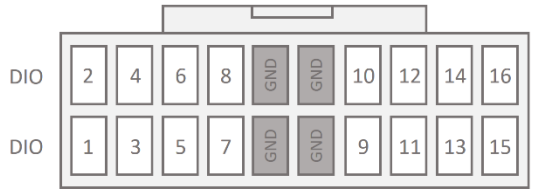


Figure 39: Multi-Interface connector pins: view from wire

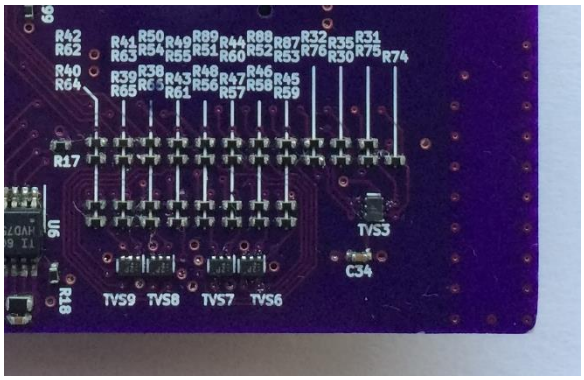


Figure 40: Multi-Interface resistor array

Table 1: Digital I/O Communication Functions

DIO	Possible Communication Functions
1	Wakeup Pin
3	Clock output, USART1 clock
5	General I/O
7	General I/O
2	USART1 transmit
4	USART1 receive
6	USART1 clear to send/USB data minus
8	USART1 request to send/USB data plus
9	I2C1 data, USART1 receive
11	I2C1 clock, USART1 transmit
13	I2C2 data, USART3 receive
15	I2C2 clock, USART3 receive
10	UART5 receive, SPI3 Software NSS
12	SPI3 clock, serial wire trace output
14	SPI 3 master-in-slave-out
16	I2C1 SMBA, SPI3 master-out-slave-in

All pins that are broken out by the multi-interface connector are protected from overcurrent by series 180Ω resistors; all except for one pin are pulled up to 3.3V. The one exception is DIO12, which can be configured as an SPI SCK line, normally low by standard convention. In case of conflicting drive levels, the resistors limit the current to 22.8mA. The series and pull-up/down resistors are all placed in the same region on the bottom of the board for easy soldering (Figure 40). All I/Os also interface with TVS diode arrays, which will clamp voltages at -0.8V and 4.1V on each pin.

3.5 Software Design

The RICNU Plan firmware has an interrupt-driven, state-machine-based architecture that utilizes a variety of free and/or open-sourced software libraries to achieve real time control over a prosthetic device and effective data routing between the RICNU User application and low-level control components. Its main tasks are to:

- Gather sensor data from the FlexSEA system
- Analyze gathered data to form high-level control decisions
- Assert high-level control over the low-level control boards
- Route the sensor and state data to the RICNU User application
- Perform fast and robust data logging onto a local memory card

A core concept in understanding the firmware structure of RICNU Plan is that as a high-level controller, RICNU Plan's main task is to maintain a device control finite state machine. This DCFSM is not hard-coded by the user, however. Instead, RICNU Plan builds it at run-time from a configuration manifest file hosted on a local microSD card.

3.5.1 Firmware Layers

In the spirit of the Open Source Initiative, the software hosted on RICNU Plan utilizes a variety of free, and in some cases, open-source software libraries. The code developed for this project is also publicly accessible via online repositories. These libraries, along with the RICNU application and the DCFSM configuration manifest, form a top-down abstraction layer architecture, depicted Figure 41.

3.5.1.1 Hardware Abstraction Layer

At the lowest level, the firmware interacts with the Cortex Microcontroller Software Interface Standard (CMSIS) library, which interfaces with the microcontroller's CPU directly. The standard has been developed by ARM to streamline the development process for applications utilizing ARM Cortex M processors. STMicroelectronics also developed their own hardware peripheral library in C to maintain firmware compatibility across different STM32 microcontroller families and peripherals. It offers a standard interface to the various peripherals on the STM32 microcontrollers.

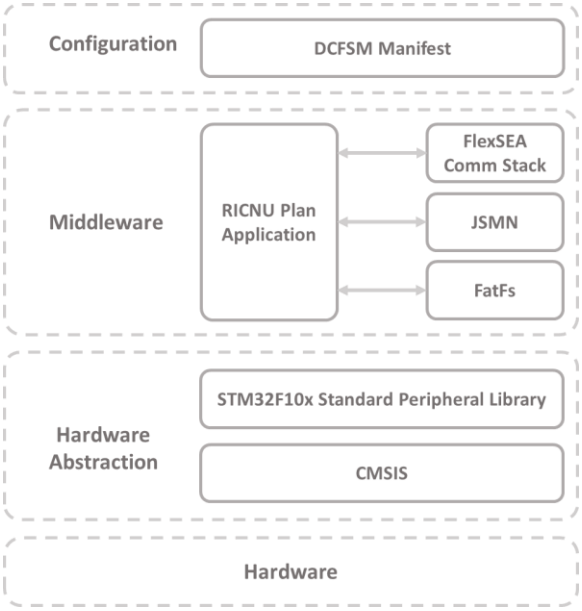


Figure 41: Software Layer Organization on RICNU Plan

3.5.1.2 Middleware Libraries

The FlexSEA system utilizes a specially-developed communication module that is somewhat platform-independent, and is used on both Execute, Manage, and Plan levels. The RICNU system uses these functions to prepare, send, receive, and unpack data on the FlexSEA Manage SPI bus.The firmware uses the time-tested FatFs library to navigate the FAT file system on the SD card. FatFs is a completely platform-independent FAT/exFAT filesystem module for small embedded systems written in ANSI C. To use it, a given system only needs to provide it with low-level interface functions, which is taken care of by the RICNU Plan application. The open-sourced JSMN library is a minimalistic JSON parsing module. The DCFSM manifest file hosted on the SD card is read using the FatFs library, and tokenized using JSMN, after which the RICNU Plan application continues the parsing process using the generated tokens.

3.5.2 Firmware Architecture

Because RICNU Plan must maintain real-time control over a prosthetic device, its firmware has been built around a simple real-time platform. Almost all operations are scheduled to happen within a repeating 4ms frame, which is the maximum refresh rate for the sensor data gathered by the FlexSEA system. Within each frame, the application acquires new data from and sends control commands to the FlexSEA boards, analyzes incoming data, forms high-level control decisions based on it, and relays the sensor data and the device state information upstream to the RICNU User application. Data logging, however, is a relatively slow process and happens asynchronously; thus, all other abovementioned tasks are interrupt-driven and preempt the logging process. The interrupts invoking these tasks are generated based on hardware timers and on completion of direct memory access (DMA) data transfers. Because all of the scheduled actions are performed on each new packet of available data, the system is able to achieve real-time performance.

On top of scheduling tasks within the 4ms time frames, the firmware implements a state machine to handle various system states. For example, the program can assume a calibration state, an idle state, and an active control state, among others. The state diagram in Figure 42 depicts the program state machine.

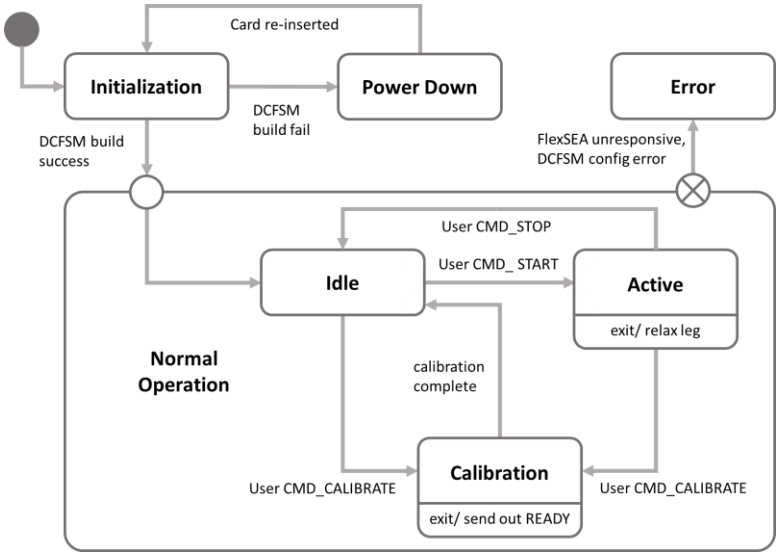


Figure 42: System state machine

3.5.2.1 Initialization State

Without a correct DCFSM structure to follow, RICNU Plan loses its purpose in the system as a high-level controller. Thus, the DCFSM build process outcome determines whether the board enters normal operation or maintains a low power waiting state. Consequently, besides configuring hardware peripherals of the STM32F103RD microcontroller, the initialization sequence also attempts to mount the memory card's file system, read the DCFSM manifest file, and build an internal representation of the DCFSM structure. If this process fails, the board goes into a low power state and indicates to the user that there is an error by turning on a red LED. The prosthetic device is safely left in its original state.

3.5.2.2 Power Down State

If the initialization sequence has failed, the MCU enters a low power state. It exits this state only once the memory card is re-inserted, presumably after a fix to the DCFSM structure, file system integrity, or the memory card itself. Initialization state is assumed once again.

3.5.2.3 Normal Operation Superstate

Once the DCFSM has been successfully built during initialization, the program assumes normal operation, which entails three substates: Idle, Active, and Calibration. No matter the substate, the microcontroller works on the 4ms time frame mentioned above; it actively communicates with the FlexSEA system and routes the received data to RICNU User, so the researcher is able to view sensor data at all times. However, the differences between the three substates lie in assertion or lack of control over the prosthetic device, data logging, and other processes.

3.5.2.4 Idle Substate

After initialization, the Idle state is assumed. In Idle state, the board is simply waiting for a command from the RICNU User app to either calibrate downstream sensors or to start controlling the prosthetic device using the DCFSM structure.

3.5.2.5 Active Substate

Active state is assumed when the user sends a "start" command from the RICNU User app. The DCFSM can be thought of as a submachine within the Active system state. Besides collecting sensor data, the microcontroller analyzes it and forms the high-level control decisions based on its interaction with the DCFSM structure. If logging is turned on through the RICNU User app, the microcontroller also asynchronously logs sensor data, system state, and DCFSM information. Upon exit from Active state, the microcontroller sends a "no control" command to the FlexSEA system, relaxing the prosthetic device. Alternatively, with further development of the firmware, it is possible to send the motor into a low-compliance state, which enables unpowered ambulation.

3.5.2.6 Calibration Substate

The user may also select to perform a calibration procedure at any time during the device operation. Calibration entails finding sensor data offsets and prosthetic device motor limits (the latter is unimplemented at this time). Upon exit from the Calibration state, the MCU assumes the Idle state and sends out an affirmation packet to RICNU User so that it can change its user interface appropriately.

3.5.2.7 Error State

If the FlexSEA system becomes unresponsive or an error is detected in the DCFSM structure (caused by user error when creating the manifest), the device enters an Error state. No matter the cause of the error, the board simply tries to send out a "no control" command to FlexSEA for safety reasons.

At this time, the only way to exit this mode is to reset the device. For one, if the DCFSM structure contains an error, it is still necessary for the user to take out the SD card, fix the DCFSM configuration file, and plug it back in. Then, the difference between an MCU reset and re-building the DCFSM straight out of the Error state is a press of the reset button. In case of loss of communication with FlexSEA, the DCFSM cannot be maintained or tracked anyway, so the best strategy is to just assume that DCFSM control is lost and re-start control from the initial DCFSM state after an MCU reset.

3.5.3 Communication

RICNU Plan handles communication with various devices. From the downstream side, Plan communicates with FlexSEA Manage using full-duplex SPI and user-defined peripheral devices using other serial communication protocols. One the upstream side, RICNU Plan communicates with the RICNU User application via a Bluetooth Low Energy or Classic link, depending on the central device.

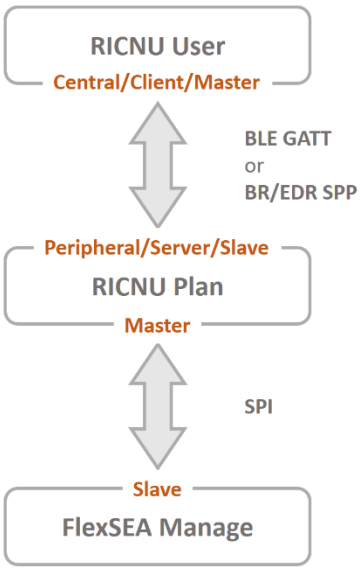


Figure 43: RICNU Plan communication roles

The RICNU system maintains a strict master/slave hierarchy proposed by the FlexSEA system, where higher-control components assert a master role over lower-control components. Figure 43 shows the roles assumed by RICNU Plan in both of those communication protocols. When interacting with other devices through other communication interfaces, RICNU Plan can be configured to be either Master or Slave, although the Master role is preferred here.

3.5.3.1 Interrupt-driven DMA transfers

Communication tasks beyond start and stop actions are handled completely in the microcontroller's hardware using Direct Memory Access (DMA) peripheral, in conjunction with the actual communication peripheral in question. Data packets are prepared beforehand using software buffers, and when it is time to communicate with another device, RICNU Plan simply enables DMA memory-to-peripheral transfers. When a transfer completes, DMA issues an appropriate "transfer complete" (TC) interrupt, which is then handled by disabling the appropriate DMA channel. Receiving channels are also interrupt-driven – when a packet of expected length is received, another TC interrupt request is raised. This way, CPU is notified when the data is ready for analysis. All memory-to-peripheral DMA channels are configured in a circular mode. By utilizing DMA this way, the CPU is only responsible for preparing outgoing and analyzing incoming data; no computation time is wasted on maintaining physical communication layer.

3.5.3.2 SPI with FlexSEA Manage

RICNU Plan controls the SPI clock when communicating with FlexSEA Manage. For compatibility, the SPI peripheral on the MCU is configured to work on a low polarity 1.5MHz clock, first edge capture, first bit MSB, and in 8-bit mode. Higher frequency clocks are rejected by FlexSEA. The chip select (CS) line is not handled in hardware, but rather in software. In hardware control mode, the STM32 SPI peripheral circuit drives CS low from the time the SPI peripheral is enabled to the time it is disabled. FlexSEA Manage, on the other hand, expects the line to be pulled low at the beginning of each FlexSEA Packet, and high right after. Because of this STM32 quirk, driving the chip select line is done in software instead.

3.5.3.3 SPI with microSD Card

As with FlexSEA Manage, RICNU Plan is a master to the local microSD card. Chip select line is also driven in software, and the very same parameters apply as well, besides baud rate. However, SD Card initialization requires some extra consideration. On boot, SD Cards are in SDIO mode. To switch a

memory card to SPI mode, the appropriate initialization sequence has to occur at a lower clock speed. So, during initialization, clock is slowed down to 375kHz. After initialization, SPI speed is set to 6 MHz.

On the application link layer, interface with the FAT file system on the microSD card is done using Elm Chan's FatFs library, which is a platform-independent FAT/exFAT module. The module needs low-level SPI bus implementation, which is handled in the RICNU Plan application.

In this version of the firmware, SPI communication with the SD card is handled by blocking operations. The main reason for this is that the Secure Digital SPI protocol is command-based and defines busy states for the memory card. The master device sends command packets to the memory card and the card responds after a certain time. Continuous read and write operations to the card often require waiting for it to exit busy states. The FatFs module thus features blocking functionality and using DMA to enable interrupt-driven operation does not add to the functionality.

3.5.3.4 UART with BT121

Communication with the RICNU User app is done via the UART peripheral of the STM32F1 microcontroller on the Plan board. Same DMA concepts apply here as to the SPI communication with FlexSEA Manage. Because communication is asynchronous and does not require acknowledgement packets (data upstream is sent blindly), the main MCU is able to continue sending data to BT121 even if it is not listening – when it is reset or when it is in programming mode.

3.5.3.5 RICNU-FlexSEA communication packet structure

The RICNU Plan application follows the FlexSEA system packet structure when communicating with FlexSEA Manage. Each packet consists of a starting token, number of payload bytes the packet contains, the actual data payload, a packet ID, a checksum byte, and an end byte [12]. The RICNU payload consists of an Emitter ID (Plan), Receiver ID (Execute), number of commands in the data packet (by default, 1),

first command (RICNU Read/Write), and a data buffer. For detailed specification of the RICNU/FlexSEA packet structure, please refer to Appendix E.

3.5.3.6 Plan-User communication packet structure

Communication between Plan and User happens wirelessly via a BLE GATT service. To achieve highest data throughput possible, the actual data sent over to the User app is packaged in a very simple way. For an outgoing packet, the first byte is a packet type identifier, indicating to the User app whether the packet contains IMU data, strain gauge data, whether it is another command. The rest of the packet contains the actual data. An incoming packet consists of two command bytes. Both are described in Appendix F.

3.5.4 Data Logging

Data logging is accomplished using the FatFs module. Using a FAT file system on the memory card introduces strict time constraints on the system, but it allows easy interface with the user's desktop computer for data analysis and DCFSM manifest file editing.

The timing of read and write operations to SD cards are non-deterministic. For example, after filling up an allocation unit on the card, the card takes some time to take care of internal processes. During this "busy" state, any commands sent to the card do not take effect.

Because of this non-deterministic behavior, the data logging process uses a software First-In-First-Out (FIFO) implementation (Figure 44). As the data log strings are generated every 4ms time frame, they are written to a buffer consisting of multiple 512-byte sectors. When a sector is filled up, it is enqueued to the FIFO structure. At a later time, the program writes this sector to the memory card, and the sector is dequeued. This way of pushing data to the memory card allows for complete disregard of the size of log strings generated at each time frame.

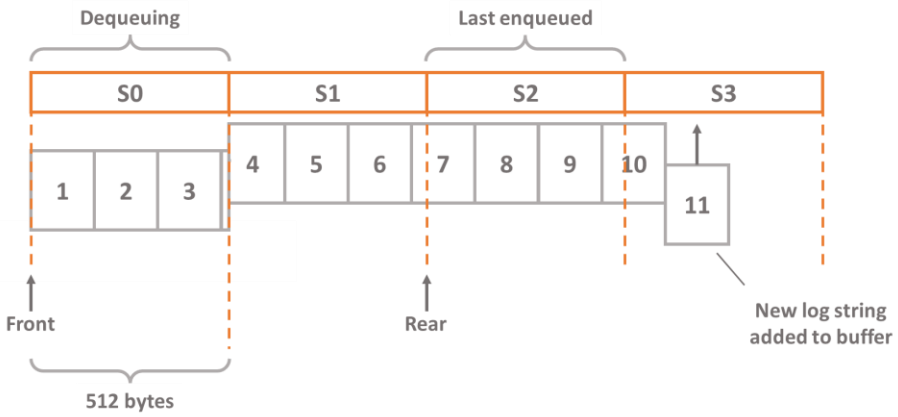


Figure 44: FIFO Implementation

The reason data is written in 512-byte-long packets is because that is the standard sector size on block-addressing SD cards (SDHC and SDXC); in fact, byte-addressing cards are configured to use a sector size of 512 bytes during the initialization procedure for cross-card compatibility.

3.5.5 Data Processing

Data from FlexSEA Manage is unpacked using the FlexSEA communication stack. If the data has been correctly packaged by Manage, the packet's payload is packed into a UART packet sent to BT121 en route to the RICNU User application. Data from each sensor channel is isolated and put into a C struct. Certain data channels (load cell data, for example) are also filtered by averaging ten previous values together. The code then checks if any DCFSM transition thresholds have been hit (Section 3.5.9.7), and if so, the command packet destined for FlexSEA Manage is modified.

3.5.6 System Timing

The Real Time Clock (RTC) peripheral is used to keep accurate time for data logging and other purposes. It uses an external 32.768 kHz crystal to count off each second of operation. Unfortunately, the chosen microcontroller's RTC peripheral does not have a millisecond capability, so millisecond timing is done by

another peripheral timer that is reset at each RTC second overflow using an RTC interrupt handler. The two are not well-synchronized, and a better alternative should be explored in next iterations of the board.

The timing for the 4ms frames is done using the internal CPU system tick (SysTick) timer, available on all ARM Cortex devices. It is configured to fire an interrupt at every 50µs, and it clocks a software timer for real-time software operation. With each SysTick interrupt, a different task can be scheduled. At the bare minimum, the SysTick interrupt starts the SPI communication frame for FlexSEA Manage.

Four LEDs are used for system state indication. All four LEDs are controlled using another peripheral timer. Upon a 0.25 second overflow, the timer issues an interrupt request; the corresponding interrupt handler changes the on/off states of each of the four LEDs based on a status bitfield variable. For this application, each LED can assume three states – on, off, or periodically toggling (flashing on and off twice each second – toggling at 4Hz).

3.5.7 Task Priority

All tasks besides data logging are handled and handled using interrupts. Their task priority and preemption are handled using the powerful Nested Vector Interrupt Controller (NVIC), featured on all ARM Cortex-M devices. The controller can be programmed to assign specific preemption priority and subpriority to each interrupt handler.

By convention across the embedded system world, a process that has a “high priority” is the “more urgent”. Somewhat confusingly, ARM Cortex-M convention reverses that – assigning a “higher priority field value” to a process means giving it a “lower urgency” status, and vice versa [27]. To eliminate the confusion, the following discussion will refer to priority in its most conventional sense – as urgency.

One of the biggest challenges of the firmware in this application is performing fast data logging while communicating with FlexSEA Manage, the RICNU User application, and, potentially, other devices. If

done sequentially and taken care of by the CPU alone, these tasks would take too much time for effective data acquisition and timely state machine decisions. Instead, as mentioned, most of the communication tasks are done using the DMA controller – meaning, without byte-to-byte CPU involvement. However, data-logging-related functions are blocking due to the intricacies of the SD card read/write protocol and have to be carried out by the CPU directly. The solution to this is to enable other processes to preempt the data logging process, first come first serve. By virtue of the FIFO implementation described in Section 3.5.4, data logging can be performed asynchronously, while data has to be acquired and processed in a timely manner.

Consequently, every process is assigned a higher urgency than data logging, with timing-critical tasks being the most urgent. Shorter processes are also assigned a higher urgency than longer tasks, as short interrupts do affect the overall timing of the system significantly. Table 2 shows the preemption priority of all major tasks.

For instance, the RTC Second interrupt is assigned the highest urgency, as its operation is critical to accurate data time stamping, but the actual handler only performs a small set of instructions. Thus, it does not affect the large-scale system timing when it preempts other processes.

SPI and UART Transmit TC handlers are assigned the next highest urgency, as these handlers are very critical for communication packet integrity. If not disabled in time, the respective DMA channels will continue circling through the respective memory buffers and sending data off to the respective communication peripherals.

The SysTick interrupt takes the next highest urgency. It is generated every 50µs, which is a long time in the embedded system world. The RTC Second and SPI/UART TX TC handlers are so brief that even when they preempt the SysTick interrupts, they do not pose any significant effect on the system timing.

Table 2: Task Priority

Task	Entry Point	Summary of Action	STM32 Preemption Priority	Urgency
RTC Second Handler	RTC second counter increments	Reset RTC if reached 23:59:59, reset millisecond timer	0	Highest
SPI Transmit Transfer Complete Handler	DMA Channel responsible for transfering outgoing data on the SPI bus completes a packet transfer	Disable SPI Transmit DMA channel	1	
UART Transmit Transfer Complete Handler	DMA Channel responsible for transfering outgoing data on the UART bus completes a packet transfer	Disable UART Transmit DMA channel	1	
SysTick Handler	SysTick overflows	Start DMA transfer of outgoing FlexSEA Manage command packet	2	
TIM2 Overflow Handler	Timer 2 overflows	Update LED states	3	
SPI Transmit Transfer Complete Handler	DMA Channel responsible for transfering incoming data on the SPI bus to memory completes a packet transfer	Unpack new data from Manage, determine control if in Active state or calibrate sensors if in Calibrate state, generate a log string if logging is turned on, and send out data to RICNU User (start DMA transfer)	4	
UART Transmit Transfer Complete Handler	DMA Channel responsible for transfering incoming data on the UART bus to memory completes a packet transfer	Unpack new command from RICNU User	4	
Main (infinite while loop)	No other tasks are pending	Write a sector to the SD card if any enqueued and system is in Active state	N/A	Lowest

The LED state timer interrupt takes on the next highest urgency. Although technically not important to system timing at all, this handler is very brief and is only requested in 0.25 second intervals. Handling it at a higher urgency does not affect other processes in any significant way.

The least urgent tasks are the DMA Receive TC handlers. The Manage SPI Receive handler is used for data unpacking and analysis, while the BT121 UART Receive handler receives commands generated through the RICNU User app by the researcher. Both handlers perform critical computational and data

routing tasks, but their *timing* is not critical. As long as both handlers are taken care of within each 4ms time frame, it does not matter when they happen *within* that frame.

Finally, at the bottom of the priority spectrum is the data logging process. It is the lengthiest of all RICNU Plan processes, and it is highly dependent on the memory card state. Thus, it is left alone to operate asynchronously, and can be preempted by all other tasks without a problem.

3.5.8 State Indication

The four main states of the system – ACTIVE, IDLE, CALIBRATION, and ERROR are indicated using four LEDs of different colors. Table 3 explains the color scheme for state indication at this point in the project development.

Table 3: LED Indication States

LED Color	LED State	System State
Green	Constantly on	IDLE
	Toggling at 4Hz	ACTIVE
Yellow	Constantly on	ACTIVE, logging is on
	Toggling at 4Hz	CALIBRATION
Red	Constantly on	Initialization error or SD card full
	Toggling at 4Hz	Run-time error
Blue	Constantly on	(user defined)
	Toggling at 4Hz	(user defined)

The green LED, as in many embedded systems, indicates normal operation. A constant ON state usually indicates readiness or a waiting period, and it does so in this case as well. However, activity is usually indicated using various flashing patterns (in various wireless internet routers, etc.).

The yellow LED has a two-fold purpose as well – to indicate that the data logging process is executing, and to indicate whether the board is in a calibration state. These actions are mutually exclusive. Again, the waiting vs. activity indication concept is applied here as well – the CALIBRATION state is a state of normal activity just like the ACTIVE state, but the board just performs slightly different operations.

The red color is usually used to indicate system errors. In this application, it made the most sense to have the red LED constantly on if the board fails to build the DCFSM structure (right after the reset), while flashing it if a run-time error occurred, following the same waiting vs. activity convention.

The above considerations cover every system state developed so far, except for actual initialization. However, initialization happens very quickly; to indicate success, the board performs a “rainbow” sequence with the four LEDs, after which the board enters an IDLE state.

3.5.9 Run-time DCFSM Implementation

On RICNU Plan, the DCFSM is built from a user-defined XML manifest (see Section 3.5.10). Because each member structure of the DCFSM (mode, state, transition, control type, etc.) can be structurally defined, the DCFSM can be nicely represented using a complex C-style struct, as shown in Figure 45.

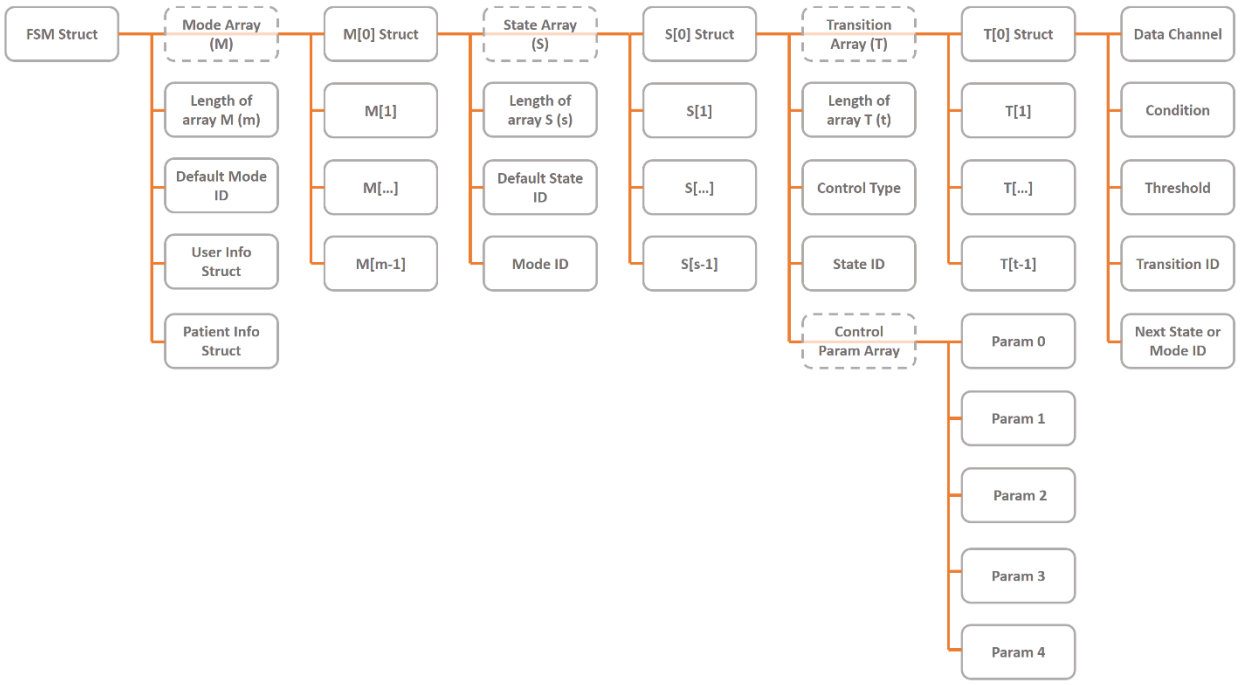


Figure 45: DCFSM Structure

3.5.9.1 Addressing

Various components of the structure need to point to others in order for transitions to happen correctly. All components in the system feature integer ID numbers that have to be unique to each component. If any two IDs in any two structures are the found to be same, the structure will not be built and RICNU Plan will not allow active control of the prosthetic device.

3.5.9.2 Transition Structure

Each transition structure features an ID number and its terminal state's ID. The transition also lists information regarding its triggering event(s). For now, only one event is allowed (i.e. a certain accelerometer axis experiences acceleration larger than some threshold acceleration), but with further development, combinations of events (i.e. event1 AND event2), time-based events, and other types of triggers can be specified as well.

3.5.9.3 State Structure

Each state structure features an ID number, an array of transitions to other states, the amount of transitions listed in the array, and state's control information. For example, the user may select between position, current, impedance, and no control to be applied to the prosthetic device when the state is assumed. Each control type features its own gains and constants.

For now, only invariable control is supported by the manifest standard. Function-based control loops (i.e. increment a gain from *x* to *y* in *z* amount of time while in the current state) are not supported yet – this level of sophistication is outside of the scope of this thesis.

3.5.9.4 Mode Structure

Each mode structure features an ID number, an array of member state structures, the amount of states listed within the array, and the default state ID. When a transition calls for a move to a particular mode, the mode's default state is assumed.

3.5.9.5 FSM Structure

The overarching FSM structure is very similar to the mode structure, except one level up – listing a default mode (used as a starting point for device control), the various mode structures, and the amount of mode structures present.

The FSM Structure also features information about the current user and patient (or test), including first and last names, integer IDs, and potentially other information if needed (as long as the Schemas and RICNU firmware agree).

3.5.9.6 Memory Allocation

Although the *structure* of the DCFSM can largely be defined, the *amount* of sub-structures like modes, states, and transitions listed under their respective parents cannot be predicted. In this version of the firmware, the DCFSM structure is statically allocated, allowing a certain maximum number of transitions per state, states per mode, and modes per FSM.

Figure 45 shows the mode, state, and transition arrays as having configurable lengths (m, s, and t), but in reality, all three arrays are actually pre-defined at compile time with constant maximum sizes. In example, the DCFSM structure can be defined to contain up to five modes. If only two are populated, the code will never reference the unpopulated members of the mode array, but they are still allocated in memory.

The microcontroller used for the task, STM32F103RD, has plenty of RAM, and even more flash. If that is not enough, perhaps future versions of FSM-building code should have the capability of parsing through the file without building the structure first, counting up exactly how much memory would be needed to build the actual structure, and dynamically allocate that memory on the heap. This way, memory will be more efficiently used.

3.5.9.7 Transitioning Mechanism

The firmware implements a special C struct to track current mode and state. Upon new data reception, the code goes through all possible transitions stemming from the current state and analyzes data to see if any respective transition rules are met. If any of the rules of a certain transition are satisfied, the code moves on to the next state specified by that particular transition. In case two transitions have exactly the same rules (which technically breaks the concept of a deterministic state machine, but is possible to specify in the manifest), the transition first listed in the manifest is followed.

3.5.10 DCFSM Building Process

A finite state machine is an easy concept to represent graphically, and Figure 42 in Section 3.5.2 is one example. Representing an FSM in C language is just as easy, provided that all parameters, thresholds, and the overall structure elements are known at compilation time. However, inferring a complex FSM structure at run-time from human-readable input is quite a large undertaking. This section attempts to walk the reader through the process to enable the reader to extend this process beyond its current state of development. At its core, the process resembles a complex parser that takes human-readable text input and isolates different elements and structures from it. The output is the abovementioned C-style struct built using a recursive-descent top-down DOM-like parsing algorithm.

3.5.10.1 User Input

The input to the system is an XML-style manifest. The user specifies the state machine using mode, state, transition, and other tags, with multiple attributes and values for each (i.e. Figure 46).

XML is relatively easy to parse because of its tag and attribute structure, but most XML parsers are too memory-heavy for small embedded systems like RICNU Plan. For this first version of RICNU Plan, it was decided to channel the process through a JavaScript Object Notation (JSON) conversion. The XML manifest can be converted by the user into JSON format using any web parser – there are a variety of tools that accomplish the same conversion with no loss of structural integrity.

The two formats are intrinsically similar in the data they can represent; the only difference is that XML is a markup language, while JSON is an object-representation language. Because DCFSM is essentially just a large object, it can be equivalently represented using XML or JSON.

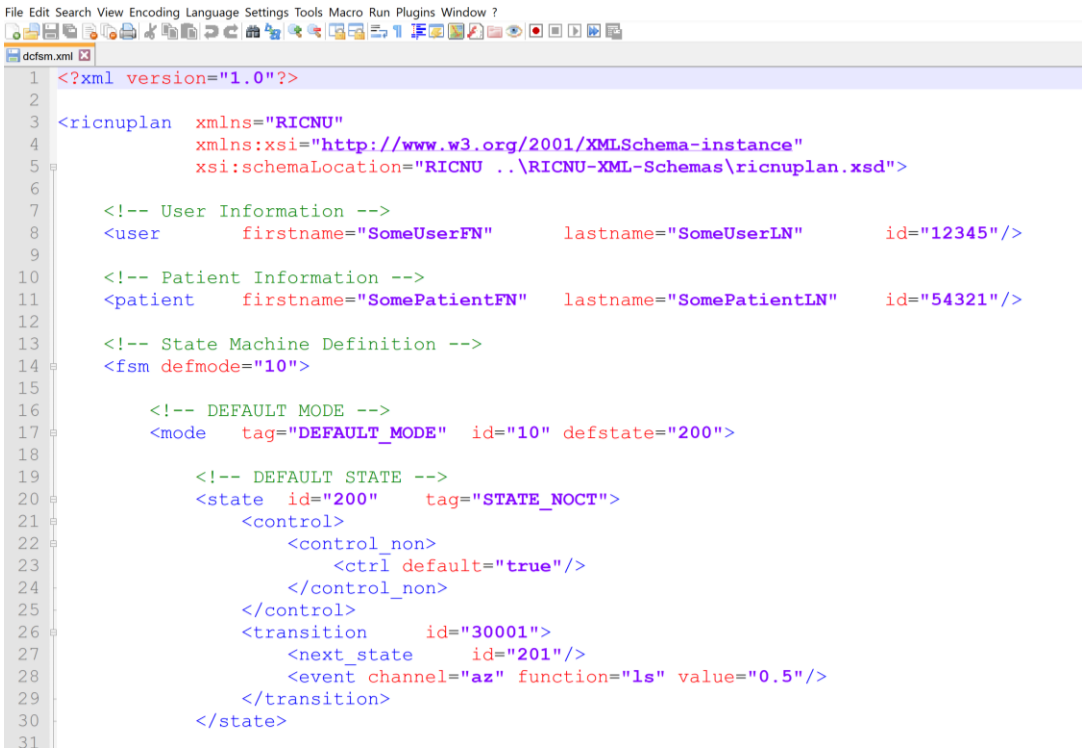


Figure 46: Snippet of an example XML manifest file

3.5.10.2 Validation

The new JSON manifest is then put onto a microSD card and the card is plugged into RICNU Plan, where the parsing process completes. But, in fact, the parsing process *starts* even before the XML to JSON conversion. The eXtensible Markup Language format was chosen to represent user-generated DCFSM because XML has a very convenient validation mechanism, which includes checking if an XML document is:

- Well-formed – follows the general XML syntax
- Valid – follows a specific document structure specified in XML Schema file(s)

Validation can happen automatically when creating the DCFSM manifest file using a text editor like Notepad++. With auto-validation enabled, the text editor checks whether the current XML document follows the general XML syntax and a specific document structure defined in the provided XML Schema files. This automatically relieves RICNU Plan from validating the file on its own. After the XML-to-JSON conversion, the JSON file is guaranteed to have appropriate structure and tag/attribute elements, as long as the Schema files used for validation are in agreement to the DCFSM structure RICNU Plan expects to see.

3.5.10.3 Transfer to RICNU Plan

Once created, the JSON manifest is loaded onto the microSD card, which is plugged into RICNU Plan at run-time. When the SD card is inserted, the FatFs module attempts to mount the file system and read the manifest file. If the file exists, FatFs attempts to read it in its entirety. If the file size is too large or there is a problem with the file system or card integrity, the process puts the board into the Power Down state described in Section 3.5.2. In future development, this process should be reprogrammed such that reading, parsing, and interpreting the manifest file is done in a streaming fashion.

3.5.10.4 Parsing Process: JSMN Tokens

Once the file is in the MCU's 64 kB RAM, the JSMN library is used to parse it. The output of JSMN parsing process are tokens – pointers to various items within a JSON file like strings, objects, arrays and numerical values. At this point, RICNU Plan has access to all of the various elements contained in the JSON manifest, and the step that is left to perform is to identify what each of these elements means and incorporate it into the DCFSM structure.

3.5.10.5 Parsing Process: RICNU Plan FSM Builder

The tokens are then used by the custom-developed RICNU Plan software to move through the file and isolate and interpret the XML-style tags, attributes, and values. The FSM building functions form a stack-based top-down parser implemented with a set movement and data-retrieving functions. The parser implements a parent-child-sibling topology, which uses three functions to move within the manifest file using the *jsmn* tokens. Using these functions, the fsm-building code implements a table of movement instructions to travel through the manifest. These movement functions are:

- `fsm_move_in()` – move to current element's child
- `fsm_move_up()` – move to current element's parent
- `fsm_move_to()` – move to current element's sibling
- `fsm_get_att()` – attempt to get an attribute from current element
- `fsm_get_tag()` – attempt to get a tag from current element
- `fsm_get_elm()` – attempt to get a specified element of an array
- `fsm_get_i()` − attempt to get an integer value
- `fsm_get_f()` − attempt to get a floating-point value
- `fsm_get_str()` − attempt to get a string value

On an even lower level, boolean functions evaluate whether an element at a current token is an XML tag, an XML attribute, a string, a JSON object, or an array. A simple stack is implemented to track where the parser is at the moment; the movement functions modify the stack using `put()`- and `pop()`-like custom functions. In effect, the above functions form a fully-functional rudimentary API. When the DCFSM XML specification is further developed, the RICNU Plan application code can be easily expanded appropriately using the `fsm_` API.

3.5.10.6 Recursive-Descent Parsing

How is it structurally possible to predict what the user will define in the file? The XML validation process comes into play, and it is essential that the user validates the manifest files against the provided XML Schemas. These files have been created in agreement with RICNU Plan's firmware; the firmware expects to see structures that the XSD files define.

For example, the XSD files provided with the software package dictate that the root tag `<ricnuplan>` must contain one and only one `<user>` tag, a choice of either a `<patient>` or a `<test>` tag, and one and only one `<fsm>` tag. Further, the `<fsm>` tag must contain at least one `<mode>` tag and an attribute `defmode` that specifies which of the modes listed (walking, standing, stairs, test, etc.) is the default mode. Each `<mode>` tag must contain at least one `<state>` tag, an attribute called `defstate`, which specifies which of the many states in the mode is the default, an attribute `id`, which is the mode's specific integer ID, and a `tag` attribute, which is basically a string name for the mode. The list of rules goes on.

The parser is written to expect certain features to be present in the file based on where the parser currently is. For complete XML specification, please refer to the Quickstart Manual included with the online documentation of the project.

3.5.11 STM32 Firmware Tools

Throughout most of the project, the primary IDE for the MCU firmware has been IAR Embedded Workbench for STM32, Kickstart edition. While this is a great IDE in a professional version, the Kickstart version limits the code size to 32 KB. With all of the added libraries, the code size increased above that limit, and the IDE of choice became the recently released Atollic TrueSTUDIO for STM32. This is an Eclipse based tool; code builds are made against C99 with GNU extensions using GCC.

3.5.12 Bluetooth – BT121 Firmware

Figure 47 shows the full Bluetooth Smart Ready stack developed by Bluegiga for their BT121 SoC module. User applications are programmed using Bluegiga's BGScript language; alternatively, the external host can directly control the module using the BGAPI command protocol via UART or SPI. BGScript is a BASIC-style language that allows easy interface with the module's variety of functions.

In this application, the custom BGScript application interacts with a custom Generic Attribute Profile (GATT), which is then used to communicate with BLE-capable mobile devices. With some adjustments to the code package developed so far for RICNU Plan, it is also possible to communicate with purely Bluetooth Classic devices using the Serial Port Profile (SPP).

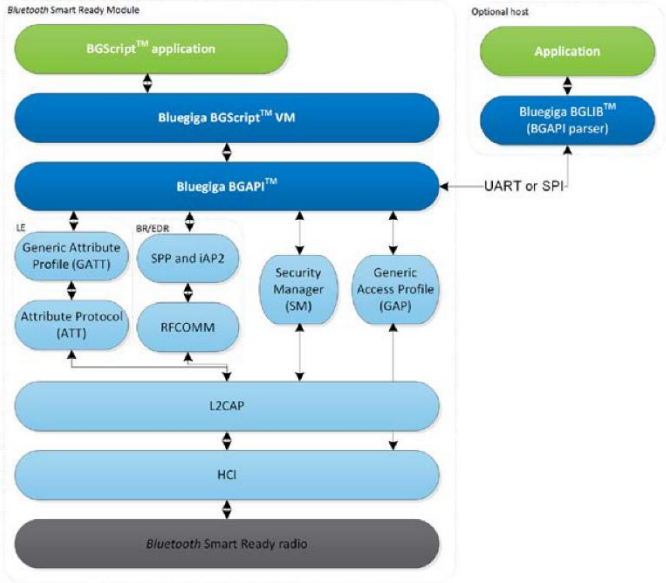


Figure 47: BT121 Bluetooth stack

3.5.12.1 Interaction with main MCU

As opposed to the stack shown in Figure 47, the host (in this case the STM32F103RD microcontroller) does not control BT121 using BGAPI; the module itself is used to take care of all communication-related tasks. Instead, BT121 merely acts as a UART pass-through. The module's custom BGScript application automatically handles tasks like device initialization, LE advertising, connection opening and closing control and other tasks. This means that the module is operating independently of the board's microcontroller, and each can be programmed and/or powered down without disturbing the other.

Bluetooth SIG specifies two important relationships between communicating devices, along with various roles these devices can assume. The two relationships are "client-server" at the GATT layer and "central-peripheral" at the GAP layer. RICNU Plan acts as the GATT server - the board supplies data to the mobile device. At the GAP layer, RICNU Plan acts as the peripheral device – it advertises its availability, but it is the RICNU User app that is used to actually request the connection to be made. At the link layer, RICNU Plan assumes the Advertiser role prior to established connection, and then acts as a Slave to RICNU User app after the connection is made.

3.5.12.2 BT121 Software Package

The RICNU Plan BT121 project consists of only four files:

- project.xml – specifies source and output file names used in the build process
- hardware.xml – configures hardware peripherals and power modes of the module
- gatt.xml – specifies the custom GATT profile used by the RICNU system
- ricnuplan.bgs – a BGScript application file dealing with various Bluetooth events

Among the three XML configuration files, project.xml merely tells the BGBuild (or BGTool) compiler where to find certain types of configuration information, as in Figure 48. The hardware.xml file merely configures the module's UART peripheral (baud rate of 230400 bps and no flow control) and disables the external BGAPI control functionality, since UART is used only for data transmission (Figure 49).

```
<?xml version="1.0" encoding="UTF-8" ?>

<!-- Project configuration including BT121 device type -->
<project device="bt121">

    <!-- XML file containing GATT service and characteristic definitions -->
    <gatt in="gatt.xml" />

    <!-- Local hardware interfaces configuration file -->
    <hardware in="hardware.xml" />

    <!-- BGScript source code file -->
    <scripting>
        <script in="ricnuplan.bgs" />
    </scripting>

    <!-- Firmware output file -->
    <image out="ricnuplan.bin" />

</project>
```

Figure 48: BT121 configuration: project.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<hardware>

    <!-- UART enabled @230400bps, no RTS/CTS, BGAPI not used-->
    <uart baud="230400" flowcontrol="false" bgapi="false"/>

</hardware>
```

Figure 49: BT121 configuration: hardware.xml

Finally, the gatt.xml file defines the services used by the RICNU Plan protocol. The two services required by the Bluetooth SIG specification are the Generic Access Service (UUID 1800) and Device Information Service (UUID 180A). They provide standard information to the inquiring device during device discovery procedure.

The third service, named "RICNU Plan Data Service" (custom UUID a0d6b998-67ef-11e7-907b-a6006ad3dba0), is what actually communicates serial data between the app and the board's microcontroller. It only has one characteristic, named "RICNU Plan Data" (custom UUID a0d6b998-67ef-11e7-907b-a6006ad3dba0), through which data is communicated back and forth.

Data can be read and written to a BLE device in a variety of ways. For instance, if data is “indicated” by the server (BT121), the host has to acknowledge that the data has been received. For small data packets, this takes up close to half the bandwidth. Instead, “RICNU Plan Data” characteristic uses notifications – data is presented to the client without the need for the client to respond. While this is not as robust as having the data indicated, RICNU User does not perform any real-time control decisions, and minor losses of data on the app side is not a problem. What really matters is the update rate at which the sensor data is communicated.

In a similar fashion, this characteristic can be written to only using the “write without response” procedure, meaning that the client cannot expect a response when it writes to the server. Again, this is done for optimized data rates.

Finally, the ricnuplan.bgs file is the embedded application code running on the BT121 module and taking care of various BLE procedures, such as advertising, connecting, loss of connection, and others.

3.5.12.3 Programming BT121

The firmware of the module is updated using Bluegiga’s BGTool (or, alternatively, BGBuild and BGUpdate tools together). This software is a graphical interface that can be used by anyone to program the device via the USB port on RICNU Plan. Figure 50 shows the BGTool during the uploading process. Performing a DFU is as easy as finding the project.xml file in the user’s file directory, clicking “Build” to initiate the build process, choosing the serial port used by RICNU Plan, and clicking “Upload” to upload the firmware to the module. Note that in order for this to work, RICNU Plan must be put into “BT121 programming mode” by switching the slide switch into “USB” position and resetting BT121 using its reset push-button.

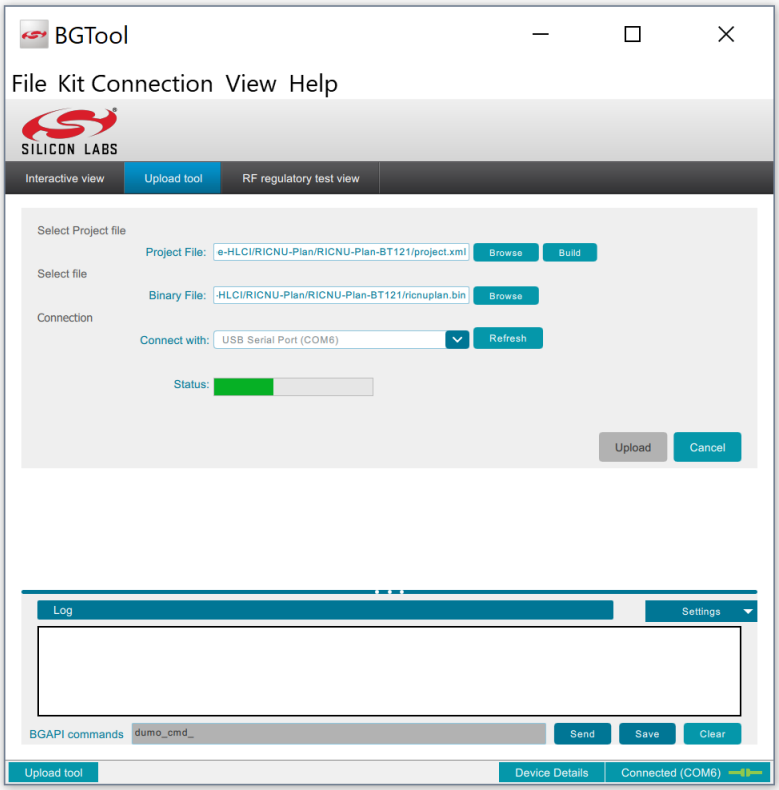


Figure 50: Programming BT121 using BGTool

4 RICNU User

The RICNU User mobile application serves as a data display and control panel for the RICNU Plan high-level controller. Following the format of Chapter 3, this chapter will first discuss the design considerations and rationale that guided the creation of the RICNU User application. Then, the software design is discussed.

4.1 Design Considerations

Important features in any mobile application are ease of use, intuitive user interface, and intuitive workflow. The simpler the UI, the better. To accomplish its main tasks, the app needs to perform two main activities – scanning and connecting to RICNU Plan and interacting with RICNU Plan's state system. It makes sense to break the two tasks up into two sequential screen displays, as opposed to showing everything together on one screen. Connectivity can be taken care of on startup, and once the connection is made, the application should transition to a data monitoring and state control display.

Another important desired feature is status communication. Whether prompting the user to scan for nearby devices, indicating that the scan is complete, or that a connection has been made and data is now streaming, the user needs a status notification feature that is common throughout the app. But, besides textual indication, this feature also needs to implement a common graphical indicator of system status. Good candidates for this type of graphical indication is a checkmark for "normal status", a cross or an exclamation-mark alert icon for "something needs your attention", and a circling progress bar to indicate a running process. Common examples of how these are used in other systems are shown in Figure 51.

Finally, information needs to be presented to the user in a concise, yet intuitive manner. Appropriate spacing between various components on the screen display needs to be observed.

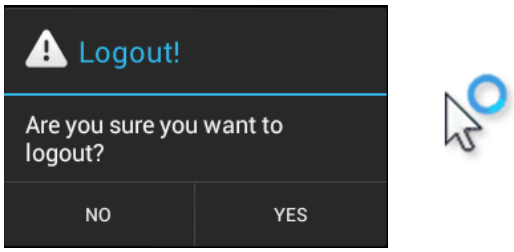


Figure 51: Common alert and progress icons Android logout alert (left), Windows 8 waiting cursor (right)

4.2 Software Design

The RICNU User application links to RICNU Plan wireless using a Bluetooth Low Energy (BLE) link and is hosted on an Android platform. At this early stage in the design, the application separates user tasks into Bluetooth connectivity tasks and data monitoring. Each group of tasks features a dedicated Android activity that has a unique layout and interface features. In the background, the application runs a BLE service, which takes care of all of the low-level BLE tasks. Figure 52 shows the RICNU User task design.

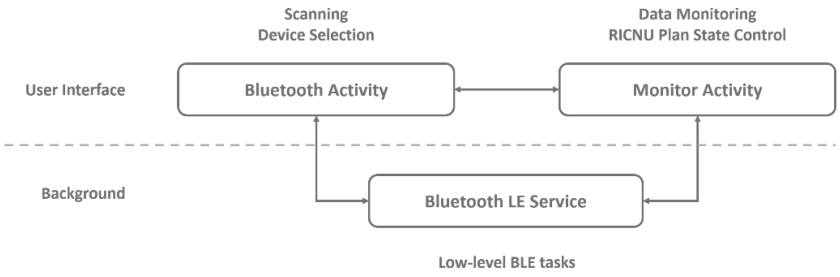


Figure 52: RICNU User task allocation

The UI has to be interactive and responsive to user actions. The app utilizes multiple state machines to navigate through the various states each activity and its UI components can assume. All Bluetooth connectivity tasks are maintained as a background process using a custom Bluetooth LE service. A common theme throughout the application is the presence of an app title bar at the top of the screen, and a status bar below it (Figure 53). The status bar displays current application status using brief text and also using an icon, which can assume three states – alert, process in progress, and normal status. This status bar is used as both a status indicator and an action prompt.

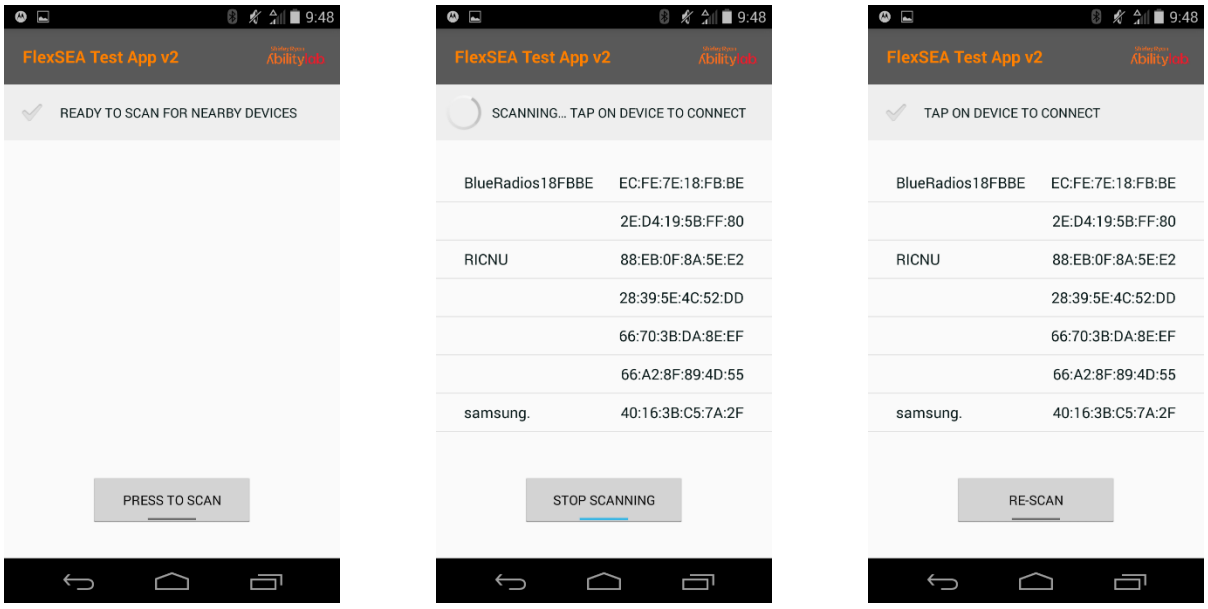


Figure 53: Example screens - scanning procedure
Starting from the left is the “ready to scan” screen, followed by “scan progress”, “end of scan” screens

4.2.1 General Application Workflow

In its current state, the application entails a very simple workflow (Figure 54). Upon a connection event accomplished via the Bluetooth Activity, the Monitor Activity is started; if Bluetooth radio is turned off, the app prompts the user to turn it back on and, if denied, exits the application.

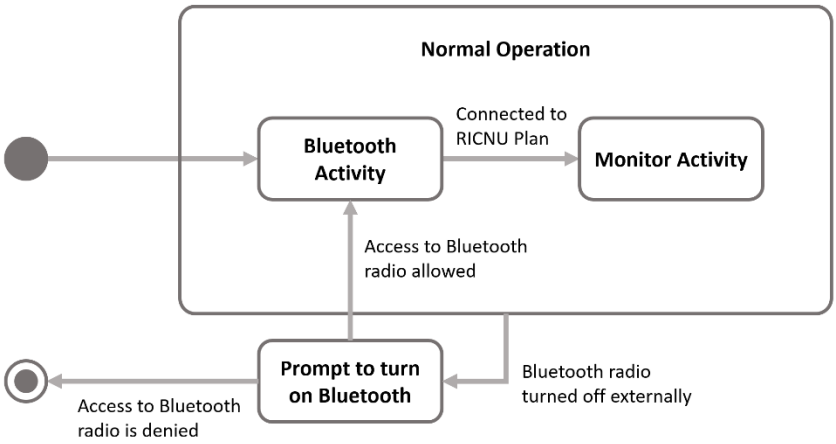


Figure 54: General application workflow

4.2.2 Bluetooth Activity

The Bluetooth Activity is dedicated to connecting to a RICNU Plan controller via a Bluetooth Low Energy radio. Further development can include a Bluetooth Classic capability, although it is found less and less frequently in modern mobile devices. The Bluetooth Activity allows the user to scan for nearby devices, select an appropriate server device, and seamlessly connect to it. Figure 55 depicts the general state machine representation of the Bluetooth Activity.

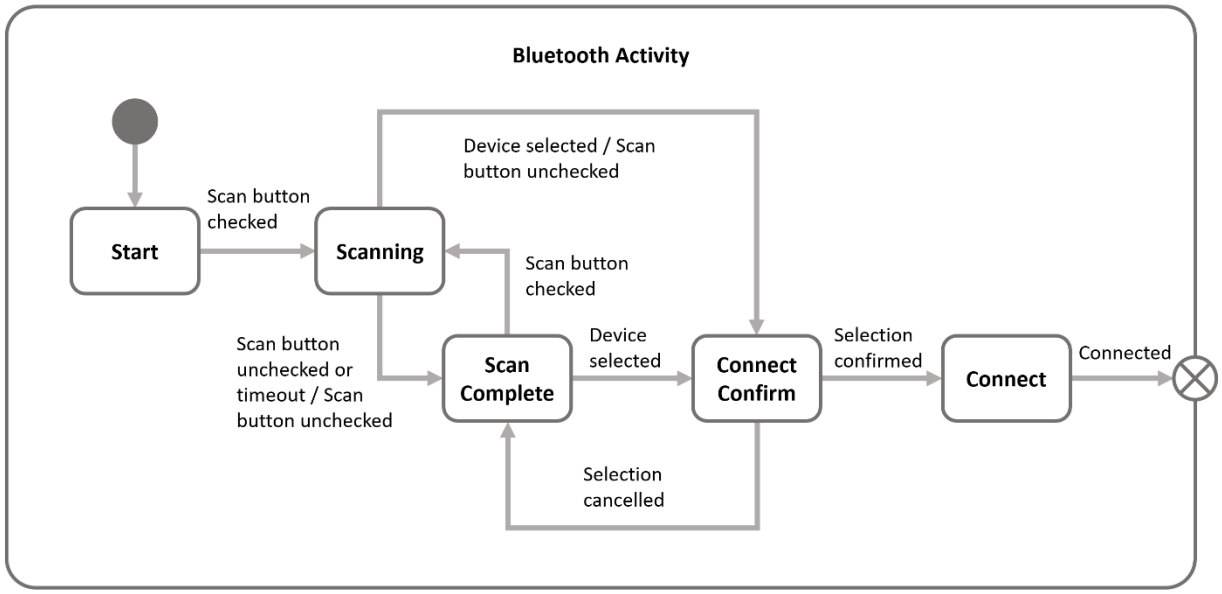


Figure 55: Bluetooth Activity state machine diagram

Figure 53 shows the graphical layout of the activity. At the top, consistent across the app, are the title and status bars. The status bar changes based on the state of the application. Below the status bar appears a list of nearby devices when the user chooses to scan for nearby BLE devices. Each entry in the list displays the “friendly” name of the device and its MAC address, to keep the UI simple. A RICNU-compatible board will identify itself as “RICNU” by default.

At the bottom, in an easily-accessible place on the screen is the SCAN toggle button. When checked, the activity performs a scan for nearby devices using the Bluetooth LE Service (Section 4.2.4), and the button changes its text and appearance to indicate to the user that the scan process is ongoing. The user may choose to cancel the scan either by unchecking the SCAN button or tapping on an appropriate device in the scan list.

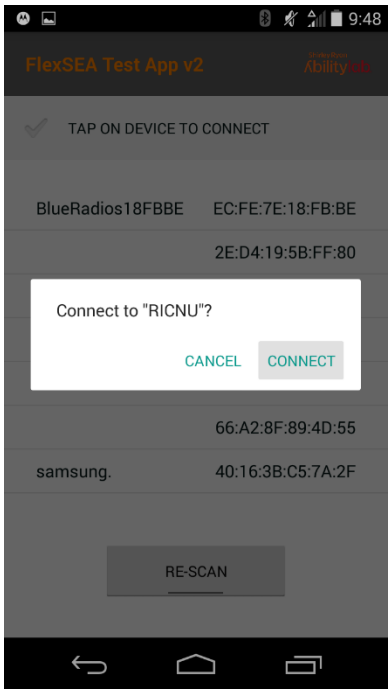


Figure 56: Connection confirmation fragment

Upon selection of a device from the list, the activity asks the user for choice confirmation using an Android fragment, as shown in Figure 56. If the user confirms the selection, the Bluetooth Activity checks compatibility of the selected device's GATT profile and, if it matches the expectations, connects to the device. If the selected device's GATT profile does not match the RICNU format, the app will indicate so to the user and return to the pre-scan screen. If the device is compatible and connection is successful, the Bluetooth Activity starts up the Monitor Activity, and the user is presented with another screen.

Navigating the different Bluetooth connectivity and user actions, especially when they are unintended, can be unmanageable without a robust state machine implementation. Under the hood, the activity implements five different state machines: activity state, Bluetooth state, scan button state, status icon state, and general user interface state. There are five state machine classes defined for the Bluetooth Activity: Bluetooth state, Scan Button state. Status Icon state, General Activity state, and UI state. Each of these state classes implements the same structure, consisting of Android *onChange* and *onChangeListener* methods, among others. The only structural differences between the five is that each class defines its own relevant states that an instance of that class can assume.

4.2.3 Monitor Activity

The main task of the monitor activity is to display to the user all of the sensor data sent to it by RICNU Plan and to give the user the ability to control the RICNU Plan board using a simple user interface. Graphically, the activity is laid out as shown in Figure 57.

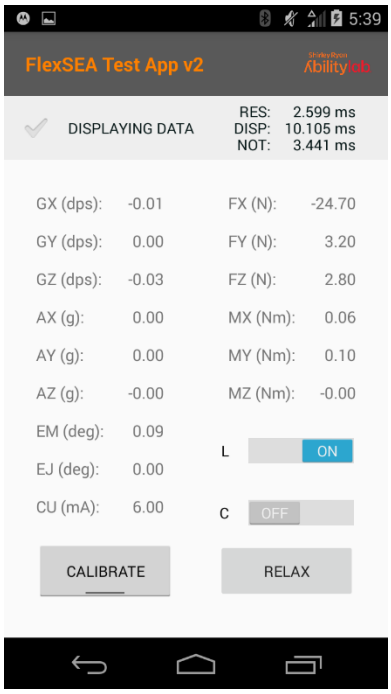


Figure 57: Monitor Activity graphical layout

As in the Bluetooth Activity, the topmost component is the title bar. Below it is, again, the status bar. Here, in addition to a simple icon and status test, there are three time displays: The RES time is the time difference between when a command has been sent to RICNU Plan and the next notification received from Plan. DISP is the time it takes for new data to be displayed after arrival. Finally, the NOT field is the time difference between two subsequent notifications. The DISP and NOT fields have been verified to work correctly, but the RES field has not been verified.

Below the status bar is a data monitor field, which shows all 15 implemented data channels: x/y/z accelerometer channels, x/y/z gyroscope channels, motor and joint encoder values, motor current, and x/y/z forces and moments sensed by the strain gauge on the prosthetic device.

Finally, there are two toggle switches and two buttons for simple control over RICNU Plan's state. The top toggle switch is used to tell RICNU Plan if it should log data or not. By itself this command has no effect; if enabled, logging only starts once Plan is put into the Active state. This is done using the second toggle switch – when on, it puts Plan into the Active state by issuing a START command. When off, it puts Plan into the Idle state by issuing a STOP command (for command implementation, see Section 4.2.7). The push-button on the right simply issues a RELAX command to Plan. On RICNU Plan, a

RELAX command causes the board to cease DCFSM control of the device and issue a “no control” FlexSEA command to FlexSEA Manage. The button on the left starts the calibration process when pressed. On the Plan side, a CALIRATE command has the RELAX command’s functionality; additionally, it causes Plan to perform a sensor calibration sequence. While the calibration goes on, all UI buttons are frozen, so the user cannot issue any unwanted commands (Figure 58). Once calibration is complete, RICNU Plan send out a confirmation packet, and the User app returns the UI buttons and switches to their normal state.

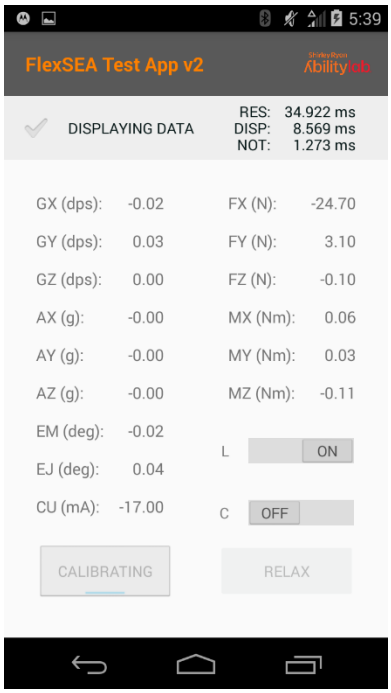


Figure 58: Calibration procedure UI change

4.2.4 Bluetooth LE Service

The Bluetooth LE Service is a background process that manages all low-level BLE events. Without going into detail, it has the capability to start and stop scanning for nearby devices, connect to and disconnect from a GATT profile, check the connected GATT for RICNU compatibility, and broadcast Bluetooth events to the Bluetooth and Monitor Activities. Communication with RICNU Plan is performed on notification basis – Plan notifies User with new data, and does not expect an acknowledgement from the User app. The same is true regarding the reverse direction – User writes commands to Plan without expecting an acknowledgement. This greatly enhances data throughput.

4.2.5 Broadcasting

Event communication between the three major app components is achieved using Android broadcasting methods. Each entity implements a broadcast filter function that listens to specific events. When an event is generated, it is caught by the active entity’s broadcast filter, and an appropriate action is taken.

4.2.6 FlexSEA Data Class

To maintain highest data throughput possible, RICNU Plan does not send processed data to RICNU User. Instead, it sends the raw data that it receives from FlexSEA Manage. To display it to the researcher, the User app has to perform computations similar to RICNU Plan. These computations are done in methods of the FlexSEA Data Class. An instance of the class is defined in Monitor Activity; this FlexSEA Data Class instance stores the data from all channels in separate values and prepares these data entries for display to the user.

4.2.7 RICNU Command Class

RICNU User commands are implemented using a two-byte bitfield. Only half of the first byte of the bitfield is used so far, but the rest of the bits there for later development. Table 4 shows the bit allocation within the command bitfield.

Table 4: User Command Bitfield

MSB bits	7	6	5	4	3-0
Meaning	0: stop active control 1: assert active control	0: do not log 1: log when active	0x00: no action 0x01: stiffen the leg (not used) 0x02: relax the leg 0x03: perform calibration		Unimplemented
LSB bits	7-0				
Meaning	Unimplemented				

The most significant bit of the MSB is the START/STOP bit, and the next bit is the LOGON/LOGOFF control. These bits maintain their state from command to command; other three commands (RELAX, CALIBRATE, STIFF) are only sent once, and their corresponding two-bit field is cleared right after. The two-bit field can assume four values, each of which is assigned to one of the above commands. The unimplemented STIFF command could be used to prosthetic device into a low-compliance state, enabling patient ambulation without any powered aid. The NULL command is simply a placeholder and does not actually trigger any event on RICNU Plan.

5 System Testing

When the RICNU Plan board was manufactured, it was hand-populated and reflown. The hardware design was tested and validated, after which the firmware development continued on from development on a test setup. Software testing included a demonstration of DCFSM build and maintenance, verification of task preemption design and real-time computational performance, and data logging capabilities.

5.1 Hardware Design Validation

Upon reception of the RICNU Plan PCB boards, two boards were populated, and one served as the development board for the rest of the project. This board's hardware was tested for protective capabilities and proper operation. ESD and surge protection tests, along with EMI evaluation are out of the scope of this project.

The inclusion of multiple 0Ω resistors throughout the design allowed testing to be done on isolated circuits. At first, all 0Ω resistors were left unpopulated. Power input stages were tested for overvoltage and reverse polarity protection; next, the power multiplexer was tested for logical behavior, and the 3.3V buck SMPS converter – for appropriate behavior. After validating the power stage, each sub-circuit like the Bluetooth Module and the USB-to-Serial converter were connected to power by populating the respective 0Ω resistors.

5.1.1 Power Input Stage

Both power inputs were tested for overvoltage and reverse polarity supply protection using a setup shown in Figure 59, made using the power input stage circuit described in Section 3.4.2. A 108.7 Ω resistance was used to simulate the expected load – drawing about 46mA at 5V supply (power budget is 250mW). A configurable power supply was used to apply a -7V to 20V potential across the input terminal (both USB and non-USB). A series ammeter was used to measure the current consumed by the system; a voltmeter

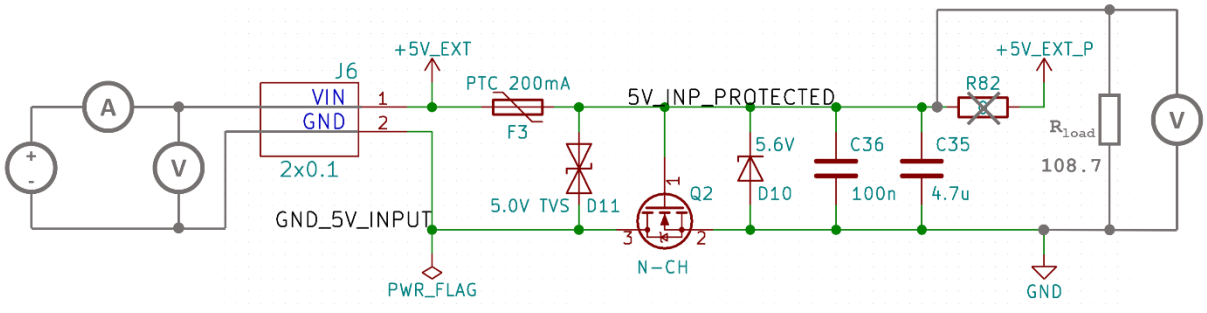


Figure 59: Power input test setup

readout was used to gauge the supply potential apparent at the input. Another voltmeter was used to measure the voltage potential seen by the next stage of the circuit. 0Ohm series resistors (in this case, R82) was taken out of the circuit to isolate the power multiplexer from the input circuit. The results of a detailed characterization test are depicted in Figure 60, and almost identical results were obtained for the USB input. For the non-USB input, which was tested first, the test entailed incrementally increasing the applied voltage in increments of 0.1V and recording data on all three instruments at each step. The measured supply potential is the potential seen by the leftmost voltmeter in Figure 59.

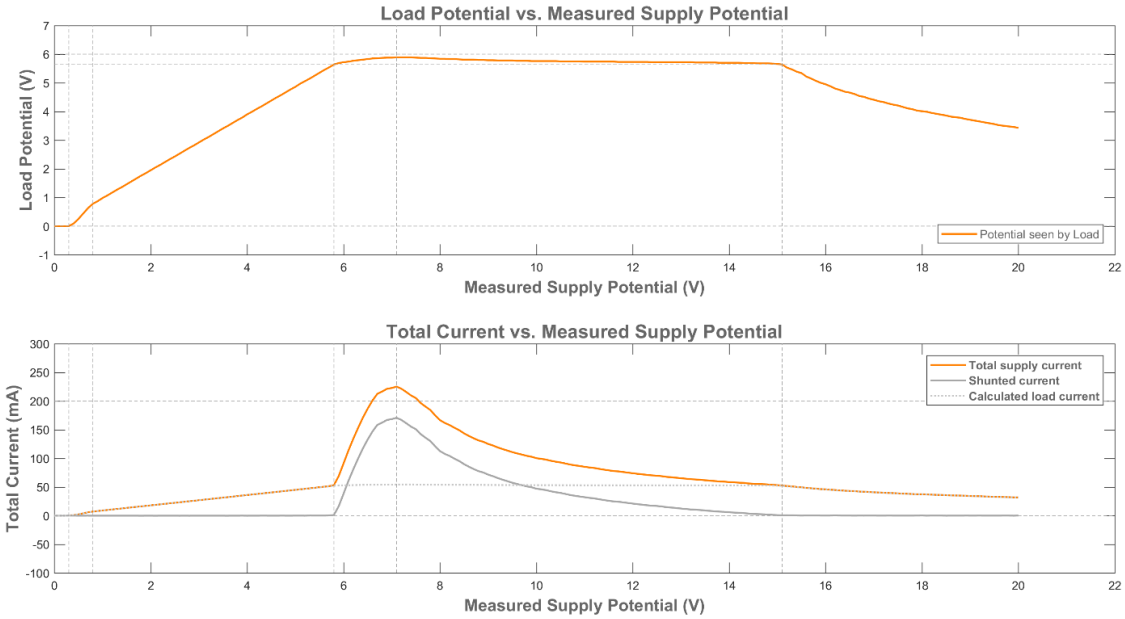


Figure 60: Power input stage overvoltage characterization- non-USB input

As apparent from the data, the input stage protects the board from overvoltage conditions of up to 20V, clamping the output voltage at no higher than 5.89V.

At sub-1V potentials, the N-channel MOSFET situated on the ground return path does not fully turn on. From about 1V to 5.7V, the input stage has a linear behavior, and this is the normal input range for the board. Beyond about 5.7V, the Zener diode starts clamping the voltage by passing through increasing current. With increasing current, the thermal impedance of the PPTC fuse increases as well. At about 7.1V seen at the input, the current shunted by the Zener diode is great enough that the current-limiting effect of the PPTC fuse overpowers the current shunting effect of the diode. From this phase, the current is further limited, returning into the normal range at around 15.1V input. Beyond this point, the diode no longer shunts current, as the voltage potential seen by the diode is limited to less than 5.7V by the PPTC fuse impedance. Thereafter, the voltage and current experienced by the output stage decreases until 20V input and beyond. Inputs of higher than 20V were not tested.

Reverse voltage protection was tested the same way, except the power supply leads were reversed. No discernible current was drawn by the circuit until about -6.3V, at which point the TVS diode D11 was reaching its bidirectional breakdown point and started shunting current. The effect is very similar to that of the voltage clamping done by the Zener. However, the Zener diode is a 3W device, whereas the TVS diode is relatively small and has not been designed to withstand long term overvoltage. Input voltages lower than -6.5V were not tested.

The USB power input circuit is extremely similar to the non-USB input circuit, the filtering stage being the only exception. Thus, the USB input was tested in larger potential increments just to validate that the components are not behaving inappropriately or differently than the ones in the non-USB input circuit.

5.1.2 Power Multiplexer

After the two inputs were tested, they were connected to the power multiplexer. The test setup was the same as in the power input stage testing, except that both series 0Ω resistors connecting inputs to the multiplexer were populated. The test load and the output voltmeter were connected across ground and the output of the multiplexer. Each input was individually excited, after which both were plugged in. Current was measured in series with each of the inputs to determine which one was used by the MUX in this condition. The supply selection behavior was as expected – the device chose the USB input over the non-USB input and isolated the two from each other; when both were present, no discernible current was drawn from the non-USB supply. In case of a single supply present, the multiplexer did not conduct between its input and output until the input reached at least 2.5V. Full current conduction was observed starting at 2.7V input. The multiplexer circuit was also tested for supply disturbances due to switching. For this, the non-USB supply was connected at all times, and the USB input was plugged in and out. The output of the PMUX was observed using an oscilloscope during switching. No discernible disturbances were observed.

5.1.3 3.3V Buck SMPS

A similar incremental test was performed on the 3.3V Buck SMPS converter. The 0Ω resistor between PMUX and Buck was populated, and the simulated load and output voltmeter were connected to the output of the Buck. Full current conduction was observed at about 3.1V input potential. After that, the converter held the voltage at 3.33V throughout the incremental test. Current loading was not performed on the converter due to unavailability of a variable load rated for high enough current. Because RICNU Plan was not designed to provide power outputs, the highest loads experienced by the supply would be those applied on it by the rest of the board. Normal behavior during regular board operation with data logging proved that the supply is more than capable of supplying the needed current.

5.1.4 Bluetooth Module and Related Circuits

For the Bluetooth Module to work, the USB-to-Serial converter and the UART bus driver circuits needed to be tested. Upon powering the FTDI converter and connecting the board to a PC USB port, the board was registered as a COM port device, as expected. The bus driver and the Bluetooth module were then powered, and a UART-based DFU of BT121 was attempted, with success on the first try. Proper functionality of the module was further validated by scanning for it using the RICNU App and easily finding it within a 20-meter range, which is more than plenty for patient testing applications.

5.1.5 Main Microcontroller

Next, the main microcontroller was powered. The SWD-based debugging session was opened without problems, and the chip was flashed with the RICNU Plan firmware. Proper operation was verified by observing the data streaming from RICNU Plan to RICNU User.

5.1.6 LED Balance

When testing the board and shining all four LED's, it became apparent that the green LED was shining too brightly when limited by a 330Ohm resistor. The resistor was changed to a 470Ohm component. The LED intensities are still not equalized and need to be further investigated.

5.1.7 microSD Card

However, the microSD card circuit as it was originally designed cause the card to remain in an idle state. The problem stemmed from a relatively high series resistance imposed on the card supply line by the protective PPTC fuse F2 (Figure 38). Once it was shorted, the microcontroller was able to perform proper reads and writes from the card, validating the rest of the SD card circuit.

5.1.8 Current Consumption

The total current consumed by the board at normal operation (active control, no data logging) was about 43.5 mA at 5V supply, which is under the specified power budget of 250mW. In error state, the board consumes about 24.8 mA at 5V supply, which is mostly due to the Bluetooth Module still functioning in normal state. When data logging is on, current changes at non-deterministic times, making it a bit complicated to measure the average current consumption without an advanced multimeter.

5.1.9 Untested Circuits

The Multi-interface, CAN, and RS-485 circuits were not tested due to time constraints of the project. Neither of the three is essential in this application and these circuits have been included mainly for board compatibility to other prosthetic control boards and sensors. As mentioned, EMI and ESD/surge event testing was out of the scope of this project.

5.2 DCFSM Build and Maintenance

As part of the general validation of the system design, RICNU Plan was tested on building and maintaining a custom DCFSM based on an XML manifest. This test was conducted to test the functionality of the entire system in one procedure. The physical test setup is shown in Figure 61. Although the dynamics of this system are different than those of a prosthetic device, the purpose of the test was not to develop a patient-ready DCFSM, but to demonstrate the RICNU Plan is able to maintain any DCFSM configured by the user using the XML manifest. FlexSEA Execute was powered by a Mastech HY1803D 18V power supply; RICNU Plan, FlexSEA Manage, and the FlexSEA load cell amplifier were all powered by the 5V power supply circuit on FlexSEA Execute. The load cell was connected to the FlexSEA load cell amplifier, which, in turn, was connected to the I2C data bus on FlexSEA Execute.

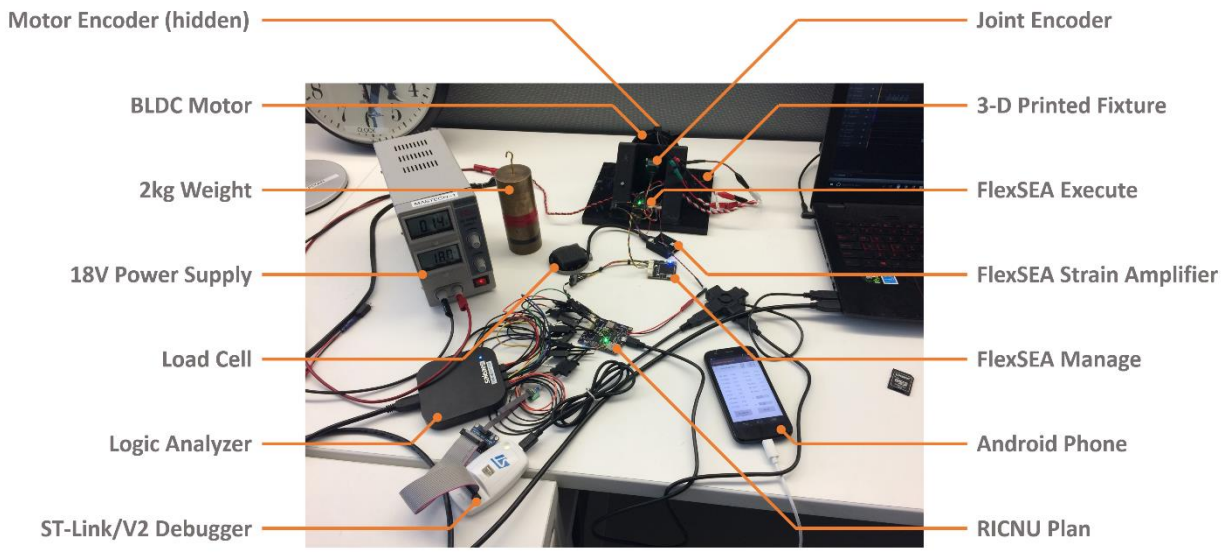


Figure 61: DCFSM Test Setup

The example DCFSM used in this test is included in Appendices A and B, which contains the XML and the JSON versions of the manifest, respectively. The DCFSM included three states – stance, early swing, and late swing (respectively named STANCE, SWING1, and SWING2). Figure 62 shows the simulated behavior configured by this DCFSM.

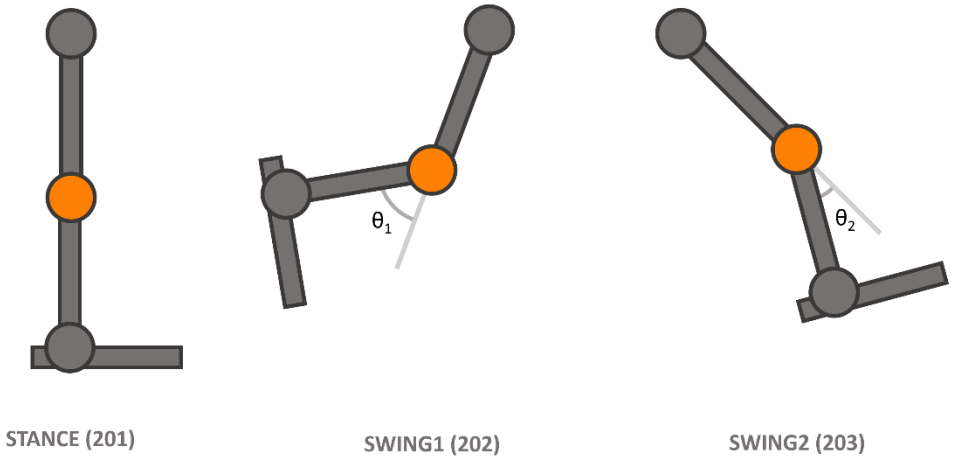


Figure 62: Simulated gait states

At stance, the leg maintains a low-compliance state. At toe-off, the knee flexes as the patient rotates the hip joint forward in early swing. At entry into late swing, simulated here by a reaching a certain angular position, the knee extends almost fully. Finally, at heel strike (load increases), the knee extends fully into the stance equilibrium position. The motor was controlled using impedance control during all three states. During stance, higher spring and damping coefficients were used; flexion and extension during early and late stance phases featured higher damping coefficients relative to their spring coefficients.

Seven gait cycles were performed, all successfully. The results can be seen in Figure 63.

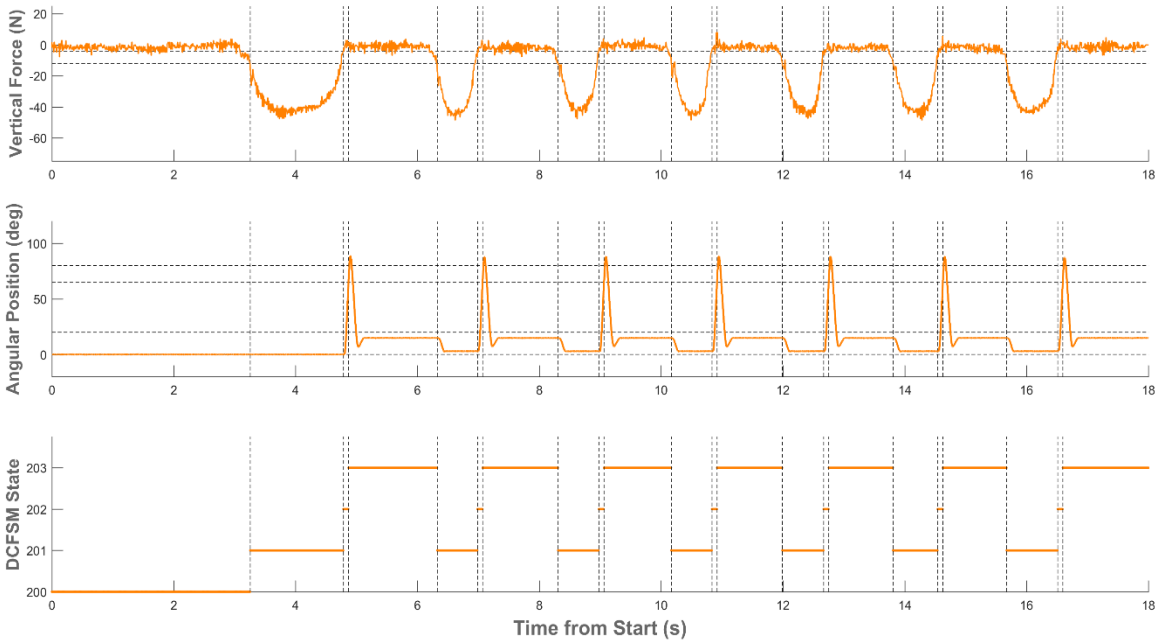


Figure 63: DCFSM maintenance in action

The top plot shows the test load applied to the load cell connected to the FlexSEA system. The convention here is that applying a force downwards results in a negative force read out. In this state machine, applying a weight to the load cell results in a transition from the SWING2 (203) or INITIAL (200) to STANCE (201) state. Releasing that load transitions the device into SWING1 (202), which forces the knee to approach an angular position of 80 degrees (middle plot). When the position exceeds 65 degrees, a transition to SWING2 (203) is triggered, and the knee quickly moves back to a 20-degree equilibrium position. Applying the load again triggers a transition to STANCE (201), and so on. The data indicates that the motor experienced slight overshoots and undershoots in angular position; these were in large part due to friction between the 3-D printed motor shaft and the 3-D printed test fixture.

The actual procedure involved a simple set of steps. First, the whole system was power cycled. RICNU Plan's calibration procedure was then invoked before asserting active control on the motor. Data logging functionality was also switched on prior to asserting active control. The RICNU User app was used wirelessly for all manipulations of the system state of RICNU Plan, and data was obtained for analysis using Plan's data logging capability. The data was logged onto a 16GB SDHC Class 10 microSD card from Kingston Technology.

The load applied to the load cell was not standardized. Although a 2kg weight was used for minimum loading, some of the downward vertical force was also exerted by the author's arm. The purpose of the test was to show state machine transitions, not the particular characteristics of the test state machine. As long as the applied load was stably below the transition threshold to move from into the STANCE state, the transition was triggered.

5.3 Data Logging

Besides the above test, data logging was also tested for both long-term stability and packet loss. The same data that was collected for the DCFSM test was used to gauge short-term performance, and another much longer data period was gathered for gauging long-term packet loss performance, using the same test setup.

5.3.1 Short-Term Performance

Figure 64 and Figure 65 show the short-term data logging performance of RICNU Plan. No packets are lost, but the time stamps show a periodic deviation from and a transient oscillation around the standard time stamp advancement.

Ideally, each log entry would have a time stamp that is 4ms away from the previous log entry. Looking at the time deviation plot (Figure 65), there is a periodic disruption of time stamp flow. This is due to the fact that the RTC peripheral on the chosen microcontroller does not have a millisecond capability.

Instead, another timer is used as the millisecond counter. The two timers are slightly out of sync with each other, and the millisecond timer reaches the value 1000 before the RTC second counter increments. In software, Plan performs a fix for this by subtracting a 1000 from the overflown millisecond counter and incrementing a variable holding the value of seconds. Clearly, this approach is not sufficient, and a hardware solution is necessary in future development.

Occasionally, the time stamps are created in a 3ms/5ms/3ms pattern, shown in Figure 65 by a transient behavior around packet 2225. The cause of this is yet unknown, but a 1ms difference between log entries is not a sufficient problem for real-time performance, as new data is only available from the FlexSEA system at 4ms intervals.

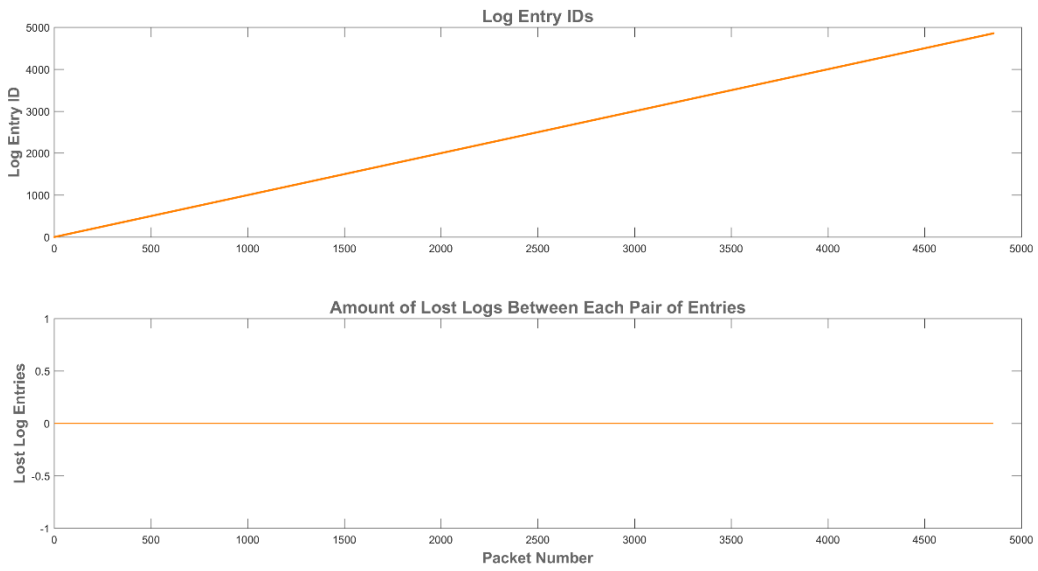


Figure 64: Short term packet loss

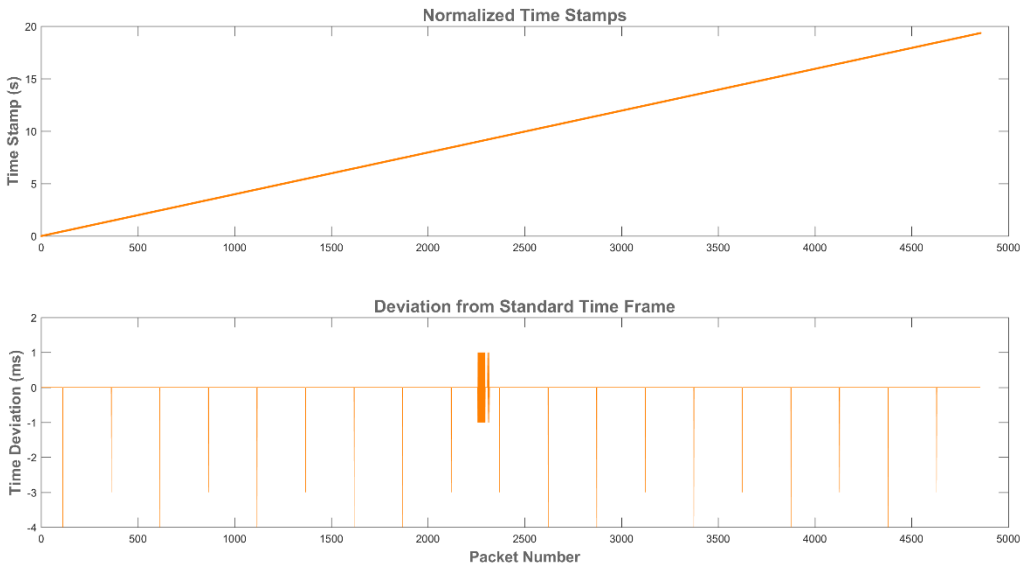


Figure 65: Short term time stamp deviation

5.3.2 Long-Term Performance

Another log file was created over the course of about 7 minutes and 49 seconds. This time period was chosen arbitrarily, as any erroneous patterns that would emerge in the long term would emerge during this time. Testing for this amount of time yielded two erroneous patterns that should be fixed with further software development of RICNU Plan.

Figure 66 and Figure 67 show the results. First, it is apparent that some data logs become lost in a periodic cycle. This is most likely due to a prolonged SD card busy state unforeseen during software development and short-term testing. A solution to this would be to increase the size of the log FIFO even further. On the time deviation plot, one may notice two patterns – small burst of deviations, and a couple second-long deviations. The former is due to the packet loss evident in the packet loss figure – those bursts happen around exactly the same times. The latter are most likely due to a software bug related to the millisecond overflow software fix.

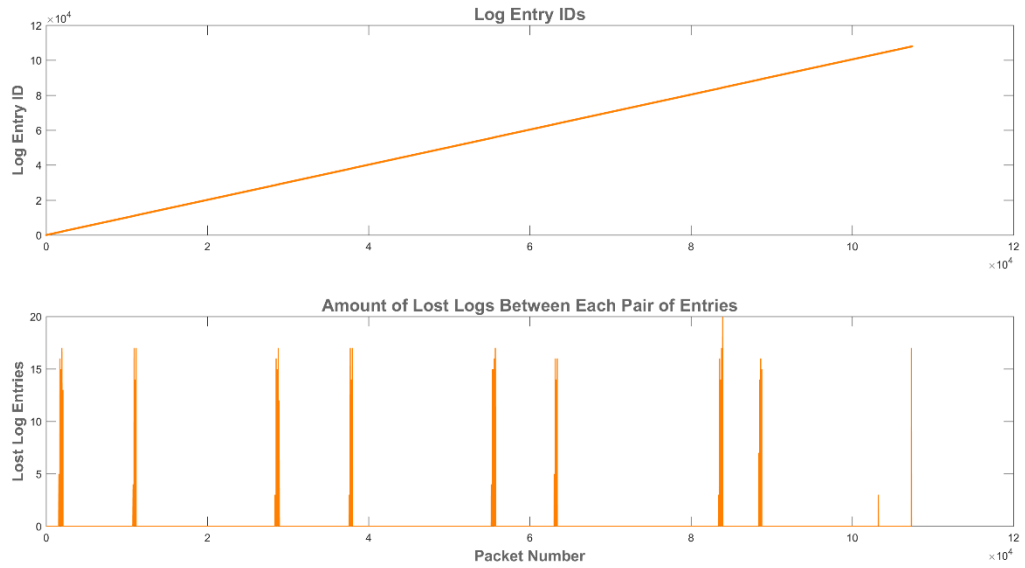


Figure 66: Long-term packet loss

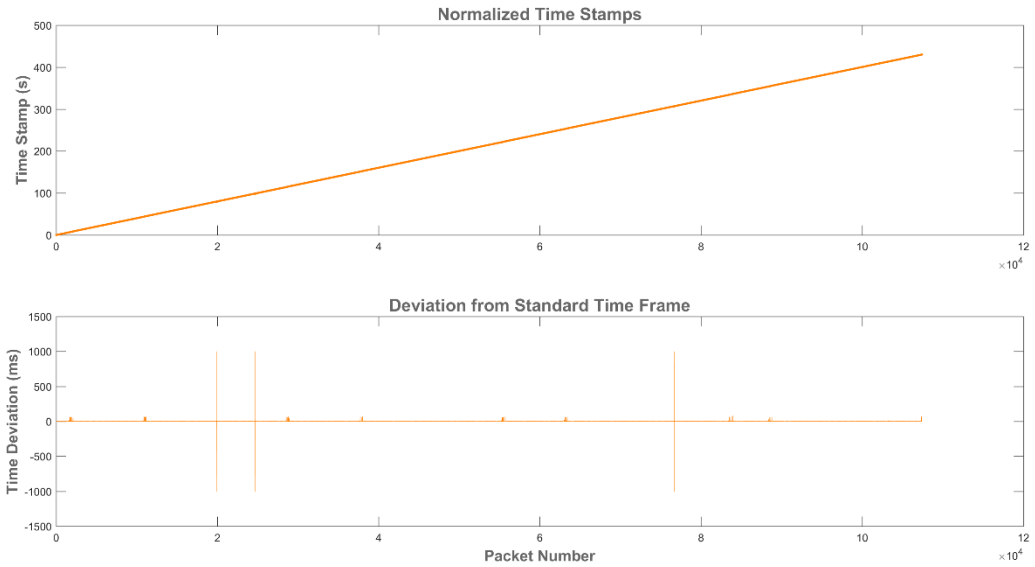


Figure 67: Long-term time deviations

5.4 Real-time computational performance

The logic analyzer in Figure 61 was used to evaluate real-time computational performance of the system. Figure 68 shows a snapshot of signals occurring on the FlexSEA Manage SPI bus, BT121 UART bus, and SD Card SPI bus over the course of about 80 milliseconds. All tasks, as expected, preempt the data logging process (top SPI bus). Communication with BT121 (Channel 4, yellow) happens after a short processing period that is triggered by completing a DMA transfer of incoming SPI data from FlexSEA Manage (bottom SPI bus).

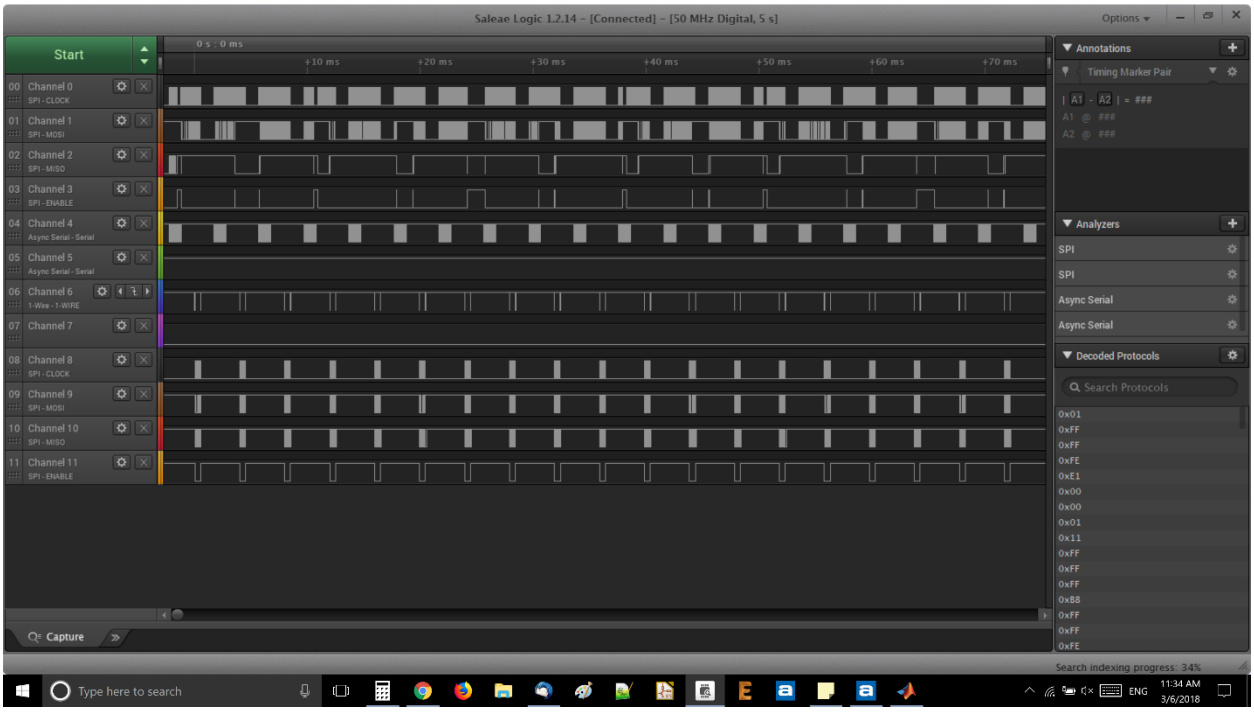


Figure 68: Communication signals snapshot

Zooming out into the second scale (Figure 69), there is an apparent SD Card busy state that happens about every 2.5 seconds. This could be due to filling of multiple allocation unit sizes in the SD card flash memory, but the exact cause is not known. However, this effect does not seem to be a likely cause of the multi-packet losses seen in the long-term logging test, as those losses happen on a scale of minutes.

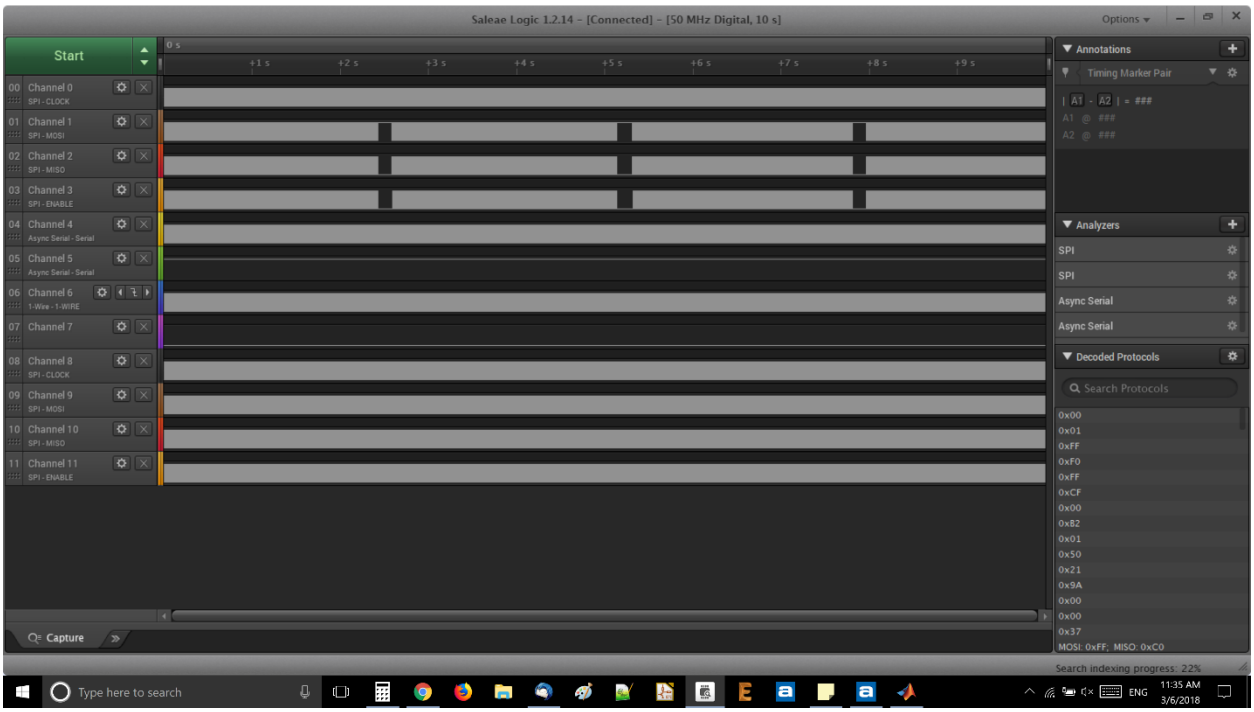


Figure 69: Real-time performance on a second scale

5.5 SD Card Compatibility

RICNU Plan's firmware was tested for compatibility with various SD card standards and manufacturers. Six cards were tested, ranging from a 1GB SDSC card to a 16GB SDHC Class 10 card. All cards were preformatted with an exFAT filesystem and all cards were readable and writeable to start with; card integrity was verified by writing a verified DCFSM file to each card using a desktop computer. RICNU Plan was able to read the DCFSM file from and log data to all cards tested, as seen in Table 5.

Table 5: SD Card Testing

Size	Family	Speed Class	UHS Class	Manufacturer	Read DCFSM	Log Data
16GB	SDHC	10	I	Kingston	Success	Success
8GB	SDHC	6	--	SAMSUNG	Success	Success
4GB	SDHC	2	--	N/A (China)	Success	Success
2GB	SDSC	--	--	SanDisk	Success	Success
2GB	SDSC	--	--	N/A (Taiwan)	Success	Success
1GB	SDSC	--	--	N/A (Japan)	Success	Success

6 Open Source Access

Various RICNU software packages, including design files for RICNU Plan, are located in separate version control repositories for the sake of independent maintenance and version control. However, instead of having to clone each repository separately, all repositories can be obtained via a single Git-Repo operation. Git-Repo is Google's git-wrapping software used to manage multi-repository projects. Git-Repo not only provides an easy means of maintaining each repository separately, but it also takes care of the file structure upon cloning all of the repositories at once; all components are placed in a single project tree and can still be individually maintained. When cloning, all source code, design files and documentation from all of the remote repositories are placed in this single project tree for easy access. For description of exact project tree and steps involved in obtaining the software, please refer to the Quickstart document included in the online documentation. Online documentation (including this thesis and Quickstart Manual) can be found at https://github.com/AlexeyRevinski/RICNU_HLCI_Doc

Once the everything is cloned and built properly, the user only has to perform the following four steps:

- Install third-party software used for building source code using links provided with the package
- Flash the board's main microcontroller using an ST-LINK/V2 debugger (or similar)
- Flash the board's Bluetooth module using a USB connection and BGTool installed on PC
- Install the RICNU User app on their Android device

Beyond this, testing can already take place – the user would need to create or modify an existing XML manifest file and use it to have RICNU Plan perform active high-level control over the prosthetic device of choice. Documentation on how to get started with the project, including this document and the Quickstart document.

7 Future Work

The next iteration of the RICNU control system could see the following improvements in the design:

- **Extending the RICNU User capabilities:** at this stage in the design, RICNU User acts only as a monitoring and Plan state control system. It is highly desirable to be able to either configure the entire DCFSM or at least be able to adjust the DCFSM parameters using RICNU User.
- **Implementing multi-DOF capability:** while most wearable robotic systems only consist of one axis of actuation, others, like the RICNU Open-Source Prosthetic Leg, entail multiple joints that allow for complicated control.
- **Implementing a single-point DFU capability on RICNU Plan:** the easiest way for the user to perform firmware upgrades for both processors on board could be either through a single USB port or wirelessly through the Bluetooth link.
- **Developing profiling schemes and ways to track patient data over time:** the current RICNU XML specification provides a very elementary means of tracking patient data through patient and user tags. Next iterations of the system could use a more sophisticated cloud-based approach to profiling.
- **Improving the data logging capability:** through testing and validation, it became apparent that some log entries are lost over time due to excessive and non-deterministic SD card latencies. The project's time constraints did not allow for a detailed analysis of the problem.
- **Reducing physical size of RICNU Plan:** for compatibility reasons, the RICNU Plan board has been designed to feature multiple inter-board communication interfaces. Upon further evaluation of system requirements, these may be found unnecessary.
- **Extending the DCFSM specification:** device control state machines can reach unmanageable complexities. The current DCFSM specification has been designed to pave the way for extending the specification.

8 Conclusion

In the course of about nine months, the RICNU control system has developed into a fully functional, although a bit restricted in functionality, high-level control system. The main goal of the project – to design, build, and develop a first generation of an energy-efficient open-source high-level controller board and a wireless graphical user interface to a lower-limb prosthetic device has been thoroughly accomplished, and, in some sense, exceeded. Besides addressing all of the minimum requirements of the project, this first version of the RICNU control system also paves flexible ways for further development and sophistication of the system. But, even at this stage in the design, the system can already be used for preliminary untethered testing on compatible prosthetic devices. The RICNU control system has been brought to a sufficient level of completion to serve as both a development framework for next generations of the system and as a research device.

9 References

[1] G. P. Kontoudis, M. V. Liarokapis and A. G. Zisimatos, "Open-Source, Anthropomorphic, Underactuated Robot Hands with a Selectively Lockable DIfferentiatial Mechanism: Towards Affordable Prostheses," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg (Germany),* 2015.

[2] "Open Prosthetics," 2017. [Online]. Available: https://openprosthetics.org/.

[3] EXII, "HACKberry Open Source Community," [Online]. Available: http://exiii-hackberry.com/.

[4] J. Gibbard, "Open Hand Project," [Online]. Available: http://www.openhandproject.org/index.php.

[5] D. S. Falero, "Drakkar - Printable Robotic Prosthetic Leg," [Online]. Available: https://hackaday.io/project/10904-drakkar-printable-robotic-prosthetic-leg.

[6] J. Zuniga, D. Katsavelis and J. Peck, "Cyborg beast: a low-cost 3d-printed prosthetic hand for children with upper-limb differences," *BMC Research Notes,* 2015.

[7] P. Slade, A. Akhtar and M. Nguyen, "Tact: Design and Performance of an Open-Source, Affordable, Myoelectric Prosthetic Hand," *Robotics and Automation (ICRA), 2015 IEEE International Conference,* 2015.

[8] D. Hill and H. M. Herr, "Effects of a powered ankle-foot prosthesis on kinetic loading of the contralateral limb: A case series," *Rehabilitation Robotics (ICORR), 2013 IEEE International Conference,* 2013.

[9] "An active step: Ottobock acquires BionX," Ottobock HealthCare, 8 March 2017. [Online]. Available: https://www.ottobockus.com/ottobock-acquires-bionx.html.

[10] S. Au, M. Berniker and H. Herr, "Powered ankle-foot prosthesis to assist level-ground and stair-descent gaits," *Neural Networks,* no. 21, 2008.

[11] I. P. Pappas, M. R. Popovic, T. Keller, V. Dietz and M. Morari, "A Reliable Gait Phase Detection System," *IEEE Transactions on Neural Systems and Rehabilitation Engineering,* vol. 9, no. 2, 2001.

[12] J. F. Duval and H. M. Hurr, "FlexSEA: Flexible, Scalable Electronics Architecture for wearable robotic applications," *Biomedical Robotics and Biomechatronics (BioRob), 2016 6th IEEE International Conference,* 2016.

[13] B. Radios, "Bluetooth 4.0 Single Mode Low Energy Programmable Universal Sensor Fob," [Online]. Available: http://www.blueradios.com/nBlue%20BR-SEN_FOB-LE4.0-S3A%20Summary%20Datasheet.pdf.

[14] J. Geerling, "Raspberry Pi Zero - Power Consumption Comparison," 27 November 2015. [Online]. Available: https://www.jeffgeerling.com/blogs/jeff-geerling/raspberry-pi-zero-power.

[15] Z. Albus, A. Valenzuela and M. Buccini, "Ultra-Low Power Comparison: MSP430 vs. Microchip XLP Tech Brief: A Case for Ultra-Low Power Microcontroller Performance," 2009. [Online]. Available: http://www.ti.com/lit/wp/slay015/slay015.pdf.

[16] S. Song and B. Isaac, "Analysis of WiMAX and WiFi and Wireless Network Coexistence," *International Journal of Computer Networks and Communications,* 2014.

[17] D. Electronics, "Comparing Low-Power Wireless Technologies (Part 1)," 26 October 2017. [Online]. Available: https://www.digikey.com/en/articles/techzone/2017/oct/comparing-low-power-wireless-technologies.

[18] K. Torvmark, "Three Flavors of Bluetooth: Which one to choose?," [Online]. Available: http://www.ti.com/lit/wp/swry007/swry007.pdf.

[19] I. Bluetooth SIG, "Our History," Bluetooth SIG, Inc., 2017. [Online]. Available: https://www.bluetooth.com/about-us/our-history.

[20] J. S. Meena, S. M. Sze, U. Chand and T.-Y. Tseng, "Overview of Emerging Non-volatile Memory Technologies," *Nanoscale Research Letters,* 2014.

[21] IEEE Spectrum, "Chip Hall of Fame: Toshiba NAND Flash Memory," IEEE Spectrum, 30 June 2017. [Online]. Available: https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-toshiba-nand-flash-memory.

[22] Toshiba America Electronic Components, Inc., "NAND vs. NOR Flash Memory: Technology Overview," [Online]. Available: http://aturing.umcs.maine.edu/~meadow/courses/cos335/Toshiba%20NAND_vs_NOR_Flash_Memory_Technology_Overviewt.pdf.

[23] Silicon Labs, "Programming the BT121," Silicon Labs, 09 07 2016. [Online]. Available: https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2016/07/09/programming_the_bt12-KAeM.

[24] USB 2.0 Promoter Group, "Universal Serial Bus Revision 2.0 Specification," [Online]. Available: http://www.usb.org/developers/docs/usb20_docs/#usb20spec.

[25] FTDI Chip, "Application Note AN_146: USB Hardware Design Guidelines for FTDI ICs," 01 11 2013. [Online]. Available: http://www.ftdichip.com/Documents/AppNotes/AN_146_USB_Hardware_Design_Guidelines_for_FTDI_ICs.pdf.

[26] Cypress Semiconductor, " Guide to a Successful EZ-USB FX2LP Hardware Design," [Online]. Available: http://www.cypress.com/file/135006/download.

[27] ARM, "Cortex-M3 Devices Generic User Guide," [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/index.html.

Appendix A: Example XML Manifest

This XML manifest file was used to configure RICNU Plan during the DCFSM testing. For XML specification guiding the creation of this manifest, refer to the Quickstart Manual included with the online documentation package for the project.

```
<?xml version="1.0"?>

<ricnuplan     xmlns="RICNU"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                       xsi:schemaLocation="RICNU ..\RICNU-XML-Schemas\ricnuplan.xsd">

        <!-- User Information -->
        <user          firstname="SomeUserFN"        lastname="SomeUserLN"         id="12345"/>

        <!-- Patient Information -->
        <patient       firstname="SomePatientFN"     lastname="SomePatientLN"      id="54321"/>

        <!-- State Machine Definition -->
        <fsm defmode="10">

                <!-- DEFAULT MODE -->
                <mode   tag="DEFAULT_MODE"     id="10" defstate="200">

                        <!-- DEFAULT STATE -->
                        <state  id="200"       tag="STATE_NO_CONTROL">
                                <control>
                                        <control_non>
                                                <ctrl default="true"/>
                                        </control_non>
                                </control>
                                <transition            id="30001">
                                        <next_state            id="201"/>
                                        <event channel="fz"    function="ls" value="-12"/>
                                </transition>
                        </state>

                        <!-- STATE 1 -->
                        <state  id="201"       tag="STANCE">
                                <control>
                                        <control_imp>
                                                <k>
                                                        <function_constant>
                                                                <constant_value value="20"/>
                                                        </function_constant>
                                                </k>
                                                <b>
                                                        <function_constant>
                                                                <constant_value  value="12"/>
                                                        </function_constant>
                                                </b>
                                                <e>
                                                        <function_constant>
                                                                <constant_value  value="0"/>
                                                        </function_constant>
                                                </e>
                                                <kp>
                                                        <function_constant>
                                                                <constant_value  value="10"/>
                                                        </function_constant>
```

```
                        </kp>
                        <ki>
                                <function_constant>
                                        <constant_value  value="1"/>
                                </function_constant>
                        </ki>
                </control_imp>
        </control>
        <transition             id="30102">
                <next_state             id="202"/>
                <event channel="fz"     function="mr" value="-9"/>
        </transition>
</state>

<!-- STATE 2 -->
<state  id="202"        tag="SWING1">
        <control>
                <control_imp>
                        <k>
                                <function_constant>
                                        <constant_value value="20"/>
                                </function_constant>
                        </k>
                        <b>
                                <function_constant>
                                        <constant_value  value="25"/>
                                </function_constant>
                        </b>
                        <e>
                                <function_constant>
                                        <constant_value  value="80"/>
                                </function_constant>
                        </e>
                        <kp>
                                <function_constant>
                                        <constant_value  value="10"/>
                                </function_constant>
                        </kp>
                        <ki>
                                <function_constant>
                                        <constant_value  value="1"/>
                                </function_constant>
                        </ki>
                </control_imp>
        </control>
        <transition             id="30203">
                <next_state             id="203"/>
                <event channel="em"     function="mr" value="65"/>
        </transition>
</state>

<!-- STATE 3 -->
<state  id="203"        tag="SWING2">
        <control>
                <control_imp>
                        <k>
                                <function_constant>
                                        <constant_value value="20"/>
                                </function_constant>
                        </k>
                        <b>
                                <function_constant>
                                        <constant_value  value="25"/>
                                </function_constant>
                        </b>
                        <e>
                                <function_constant>
                                        <constant_value  value="20"/>
```

```
                                                </function_constant>
                                        </e>
                                        <kp>
                                                <function_constant>
                                                        <constant_value  value="10"/>
                                                </function_constant>
                                        </kp>
                                        <ki>
                                                <function_constant>
                                                        <constant_value  value="1"/>
                                                </function_constant>
                                        </ki>
                                </control_imp>
                        </control>
                        <transition             id="30301">
                                <next_state             id="201"/>
                                <event channel="fz"     function="ls" value="-12"/>
                        </transition>
                </state>
        </mode>
</fsm>
</ricnuplan>
```

Appendix B: Example Auto-Generated JSON Manifest

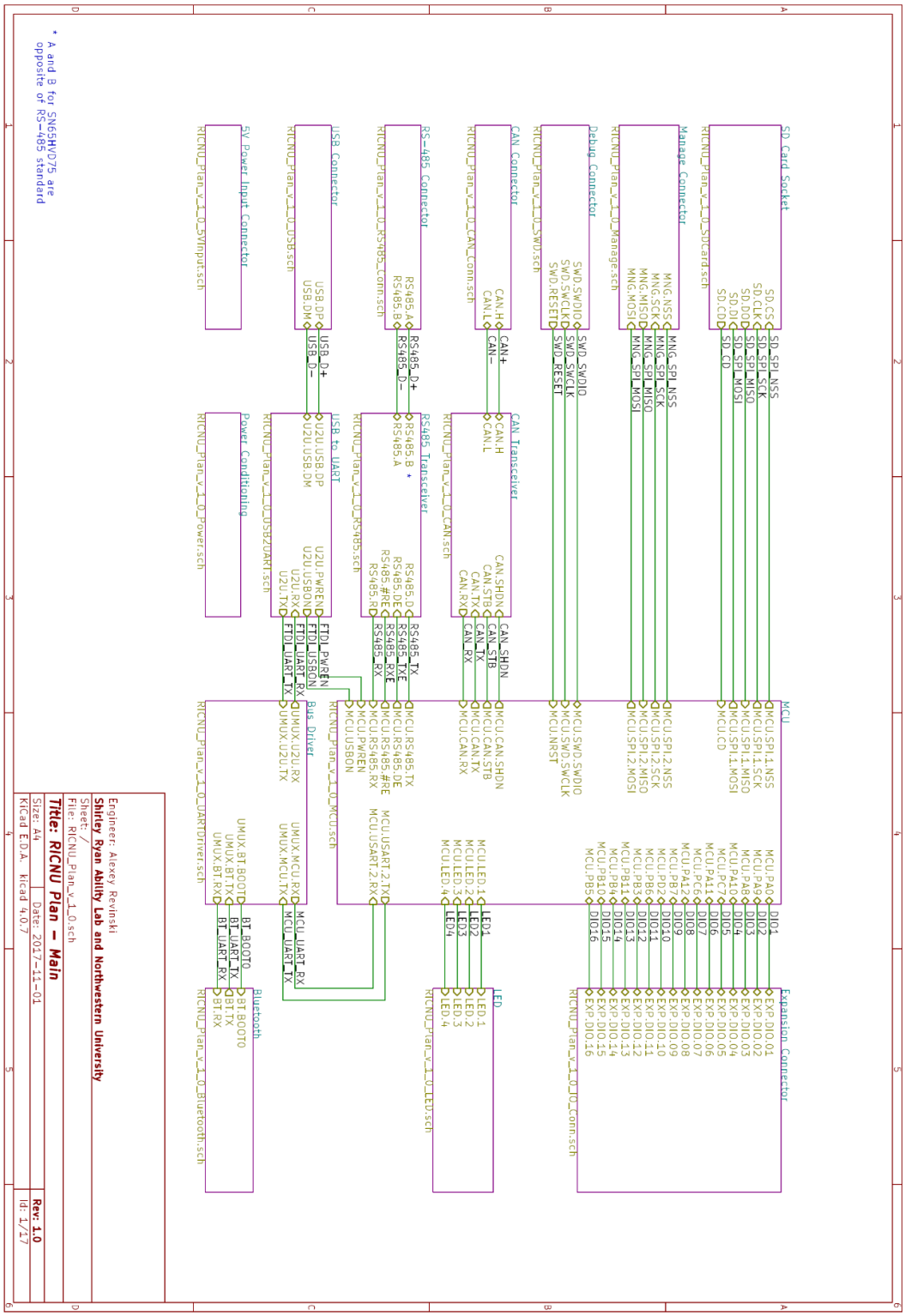
This JSON file was obtained by converting the example XML manifest using an online tool. This file was the manifest that RICNU Plan directly interfaced with.

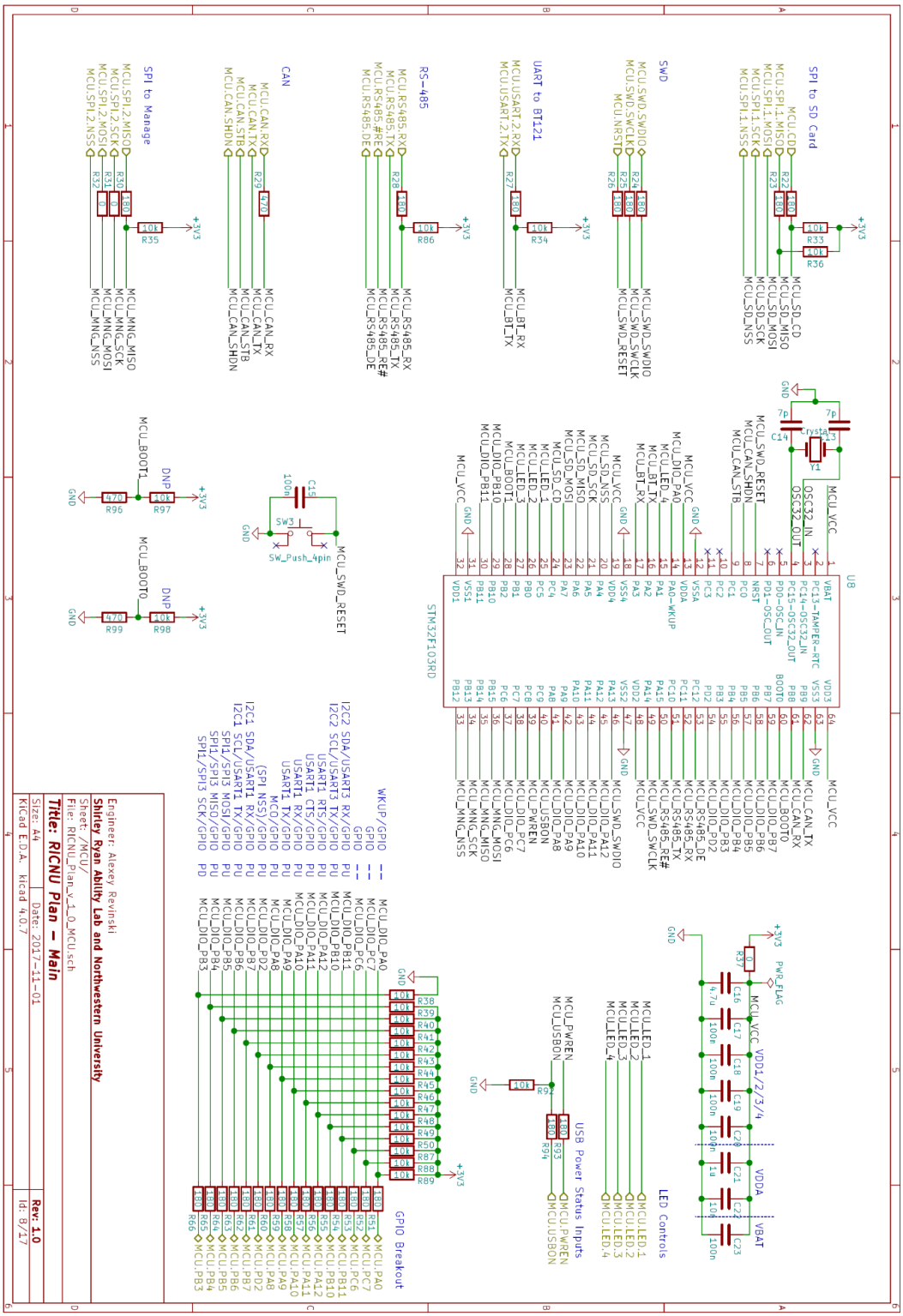

```
{
  "ricnuplan": {
    "-xmlns": "RICNU",
    "-xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",
    "-xsi:schemaLocation": "RICNU ..\\RICNU-XML-Schemas\\ricnuplan.xsd",
    "user": {
      "-firstname": "SomeUserFN",
      "-lastname": "SomeUserLN",
      "-id": "12345"
    },
    "patient": {
      "-firstname": "SomePatientFN",
      "-lastname": "SomePatientLN",
      "-id": "54321"
    },
    "fsm": {
      "-defmode": "10",
      "mode": {
        "-tag": "DEFAULT_MODE",
        "-id": "10",
        "-defstate": "200",
        "state": [
          {
            "-id": "200",
            "-tag": "STATE_NO_CONTROL",
            "control": {
              "control_non": {
                "ctrl": { "-default": "true" }
              }
            },
            "transition": {
              "-id": "30001",
              "next_state": { "-id": "201" },
              "event": {
                "-channel": "fz",
                "-function": "ls",
                "-value": "-12"
              }
            }
          },
          {
            "-id": "201",
            "-tag": "STANCE",
            "control": {
              "control_imp": {
                "k": {
                  "function_constant": {
                    "constant_value": { "-value": "20" }
                  }
                },
                "b": {
                  "function_constant": {
                    "constant_value": { "-value": "12" }
                  }
                },
                "e": {
                  "function_constant": {
```

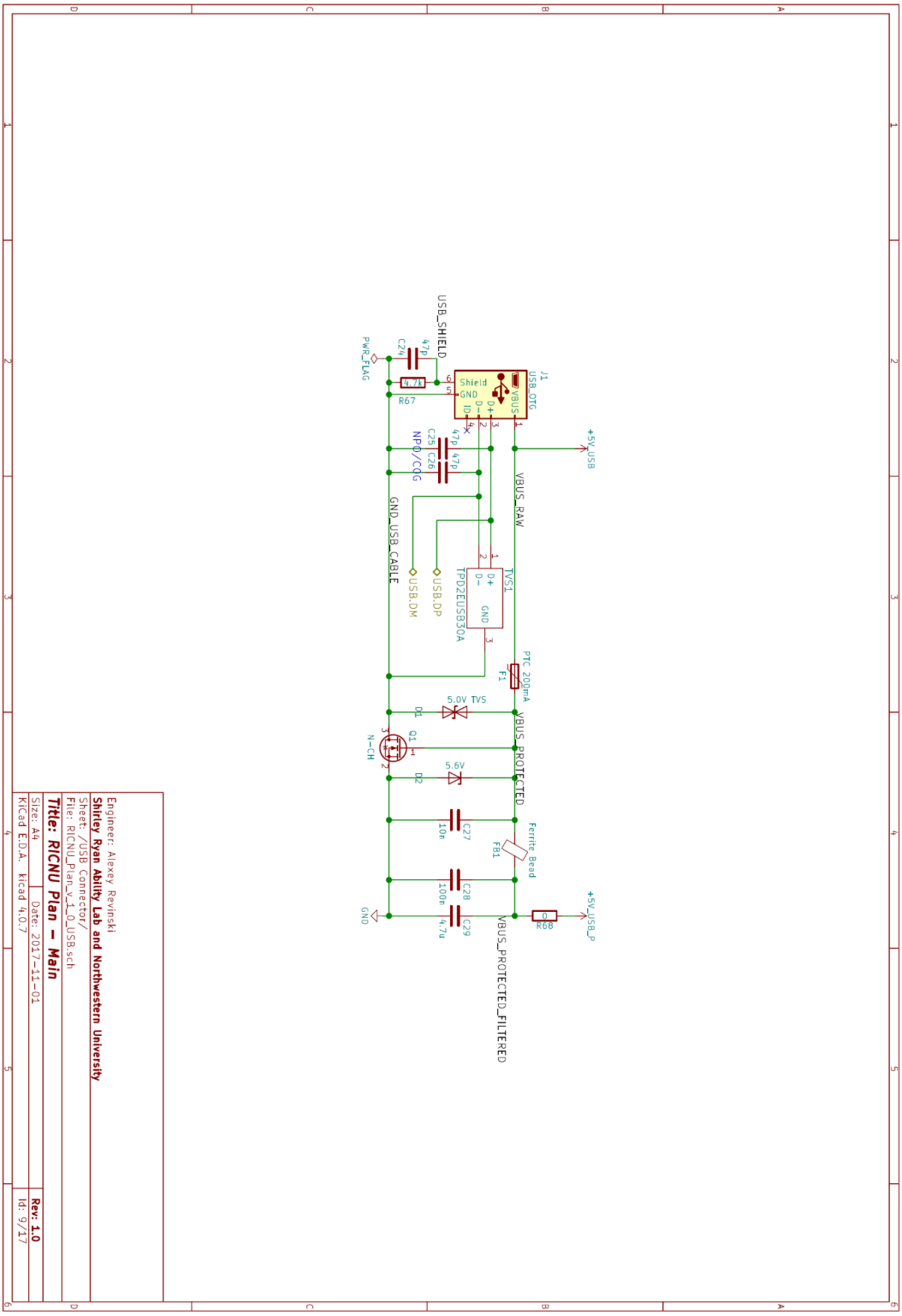
```
          "constant_value": { "-value": "0" }
        }
      },
      "kp": {
        "function_constant": {
          "constant_value": { "-value": "10" }
        }
      },
      "ki": {
        "function_constant": {
          "constant_value": { "-value": "1" }
        }
      }
    }
  },
  "transition": {
    "-id": "30102",
    "next_state": { "-id": "202" },
    "event": {
      "-channel": "fz",
      "-function": "mr",
      "-value": "-4"
    }
  }
},
{
  "-id": "202",
  "-tag": "SWING1",
  "control": {
    "control_imp": {
      "k": {
        "function_constant": {
          "constant_value": { "-value": "20" }
        }
      },
      "b": {
        "function_constant": {
          "constant_value": { "-value": "25" }
        }
      },
      "e": {
        "function_constant": {
          "constant_value": { "-value": "80" }
        }
      },
      "kp": {
        "function_constant": {
          "constant_value": { "-value": "10" }
        }
      },
      "ki": {
        "function_constant": {
          "constant_value": { "-value": "1" }
        }
      }
    }
  },
  "transition": {
    "-id": "30203",
    "next_state": { "-id": "203" },
    "event": {
      "-channel": "em",
      "-function": "mr",
      "-value": "65"
    }
  }
},
{
```

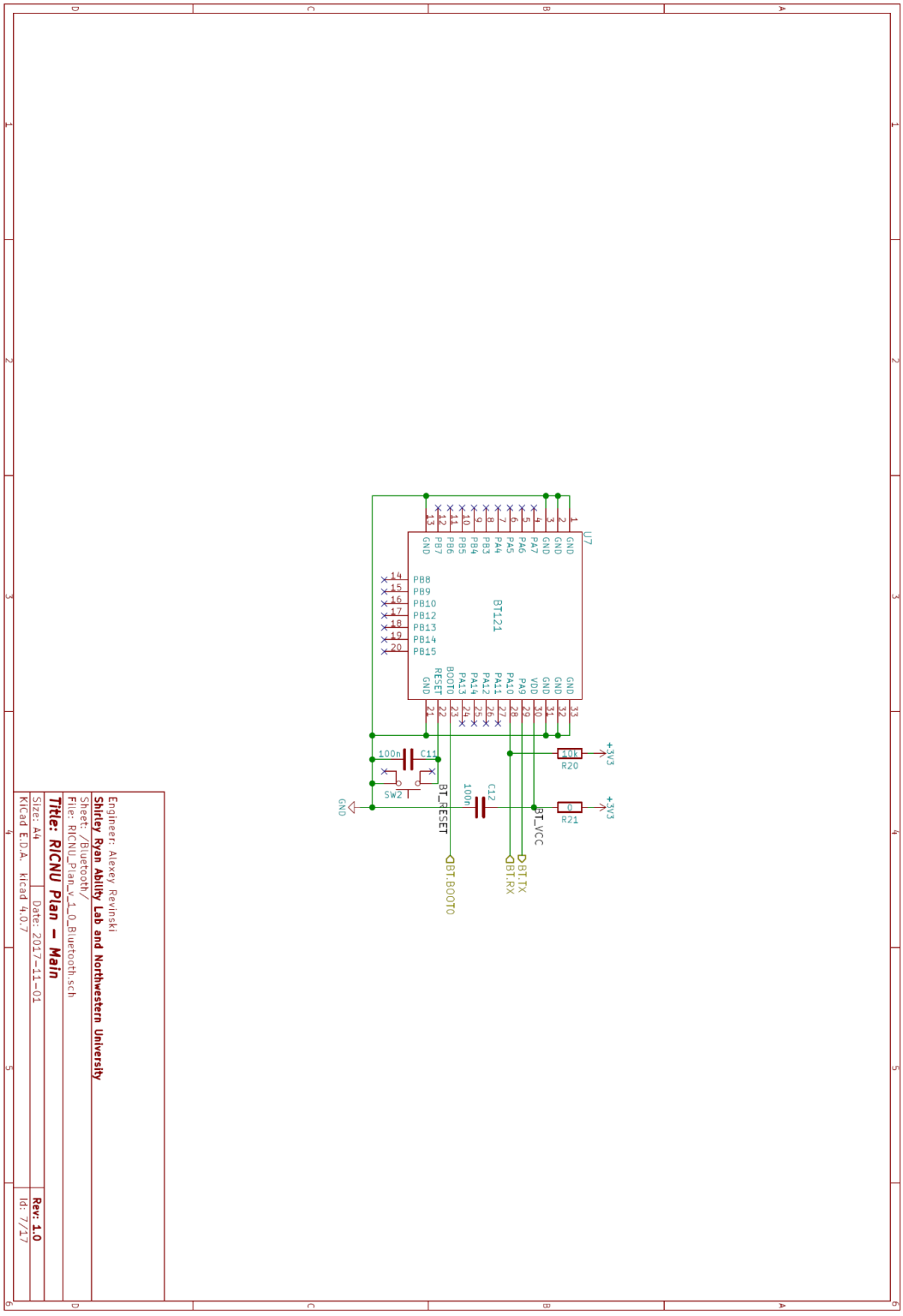
```
            "-id": "203",
            "-tag": "SWING2",
            "control": {
              "control_imp": {
                "k": {
                  "function_constant": {
                    "constant_value": { "-value": "20" }
                  }
                },
                "b": {
                  "function_constant": {
                    "constant_value": { "-value": "25" }
                  }
                },
                "e": {
                  "function_constant": {
                    "constant_value": { "-value": "20" }
                  }
                },
                "kp": {
                  "function_constant": {
                    "constant_value": { "-value": "10" }
                  }
                },
                "ki": {
                  "function_constant": {
                    "constant_value": { "-value": "1" }
                  }
                }
              }
            },
            "transition": {
              "-id": "30301",
              "next_state": { "-id": "201" },
              "event": {
                "-channel": "fz",
                "-function": "ls",
                "-value": "-12"
              }
            }
          }
        ]
      }
    }
  }
}
```

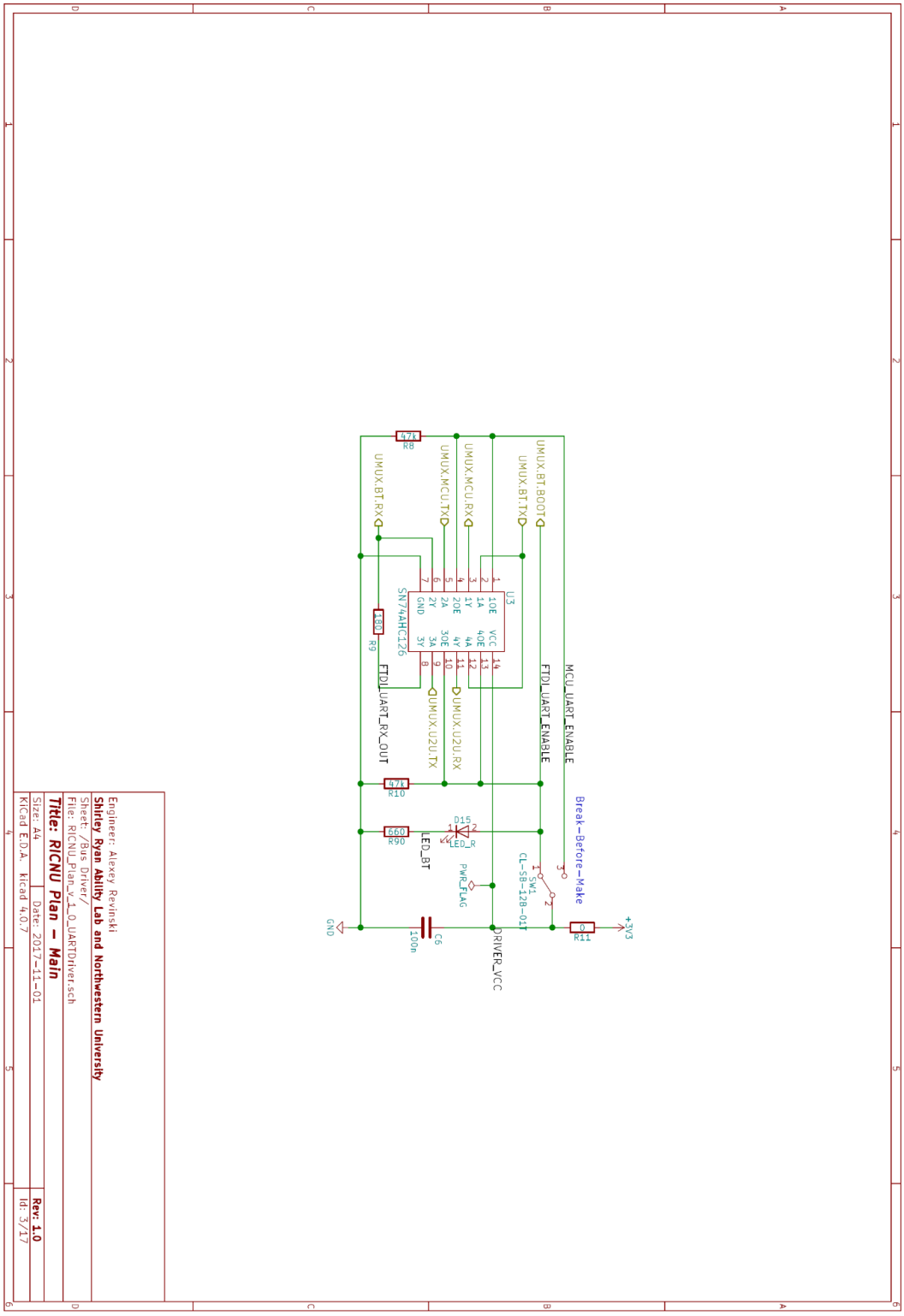
Appendix C: Hardware Schematics

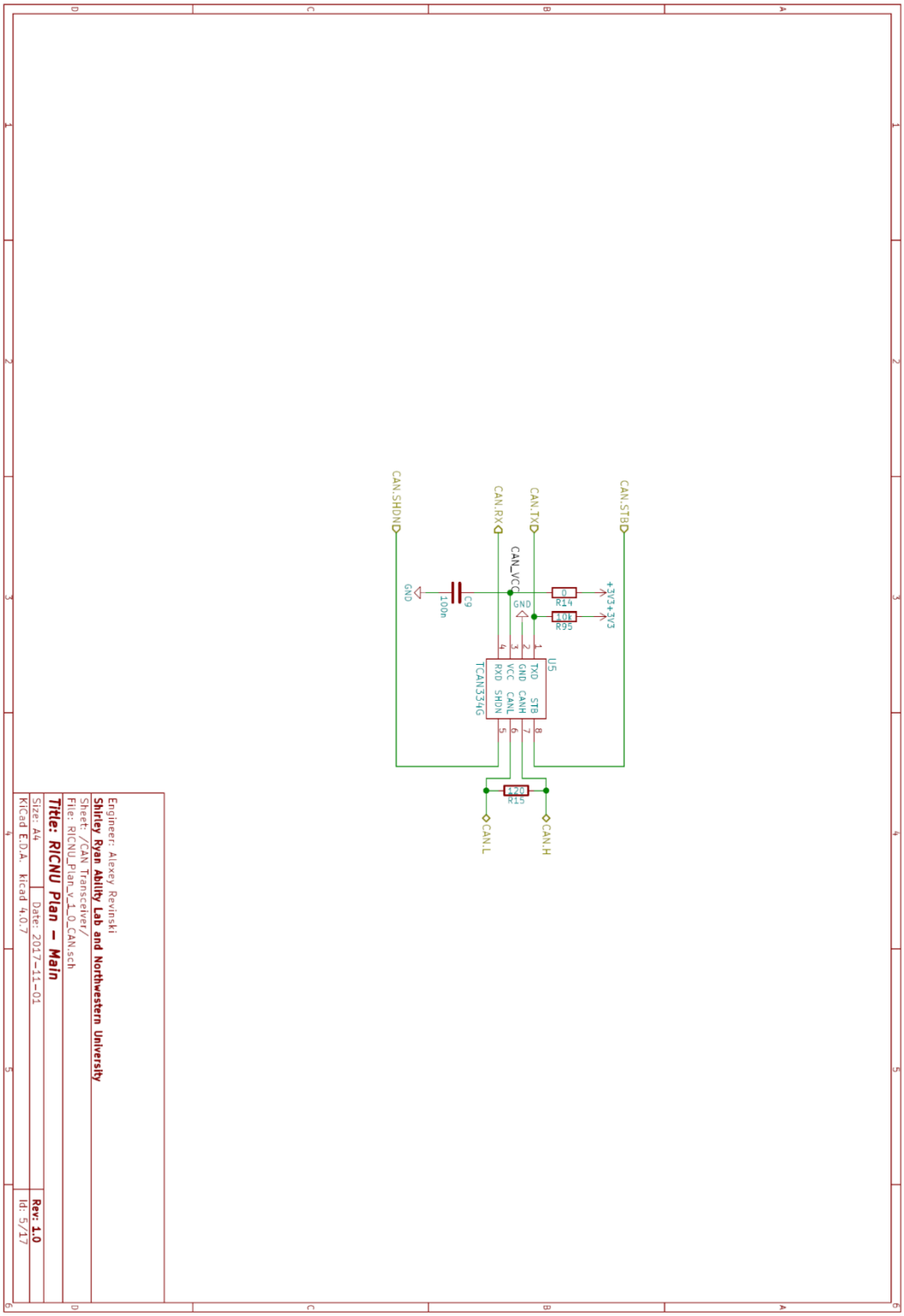


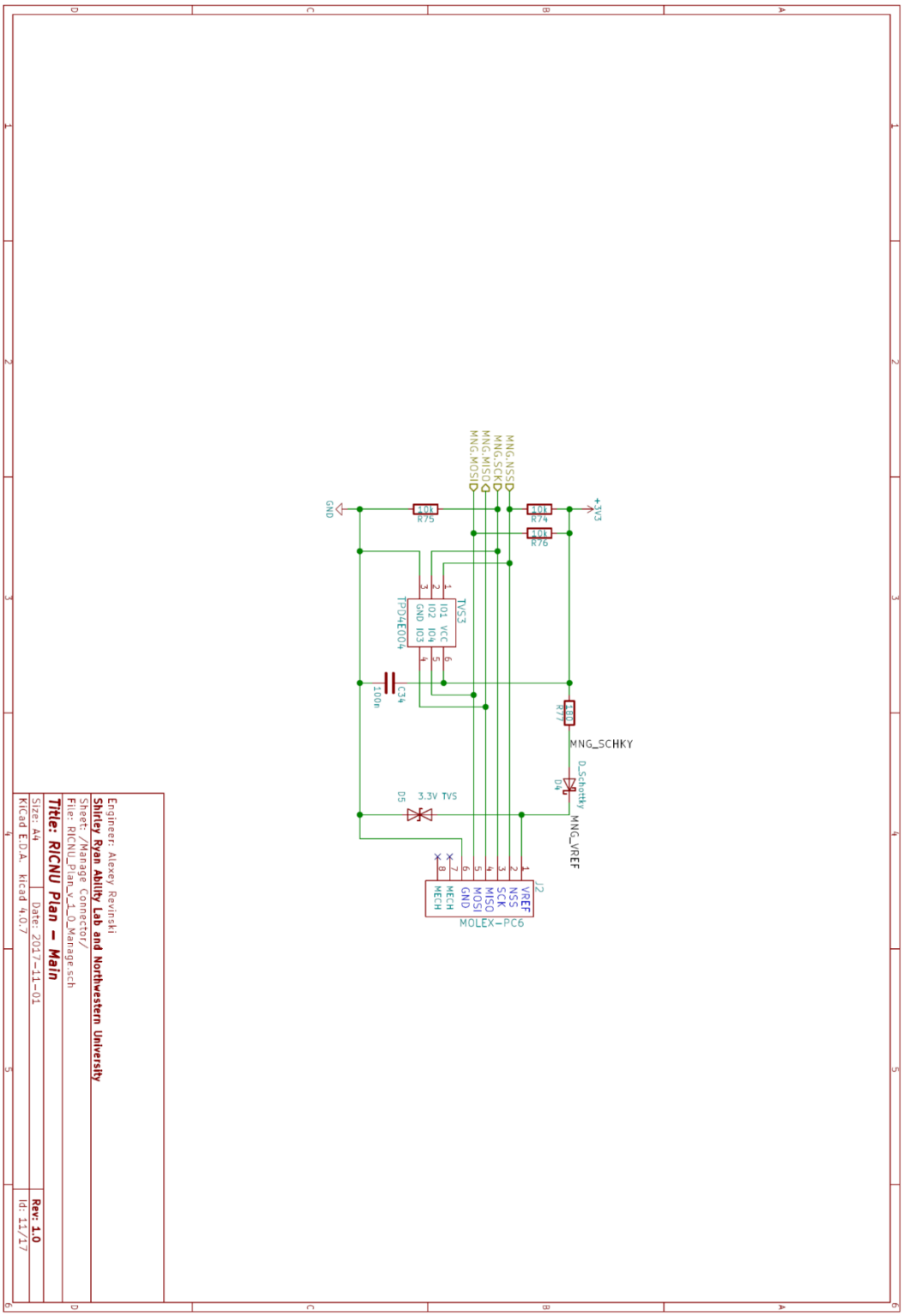

SPI to SD Card
SWD
UART to BT121
RS-485
CAN
SPI to Manage
STM32F103RD
LED Controls
USB Power Status Inputs
GPIO Breakout
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Title: RICNU Plan – Main
Rev: 1.0

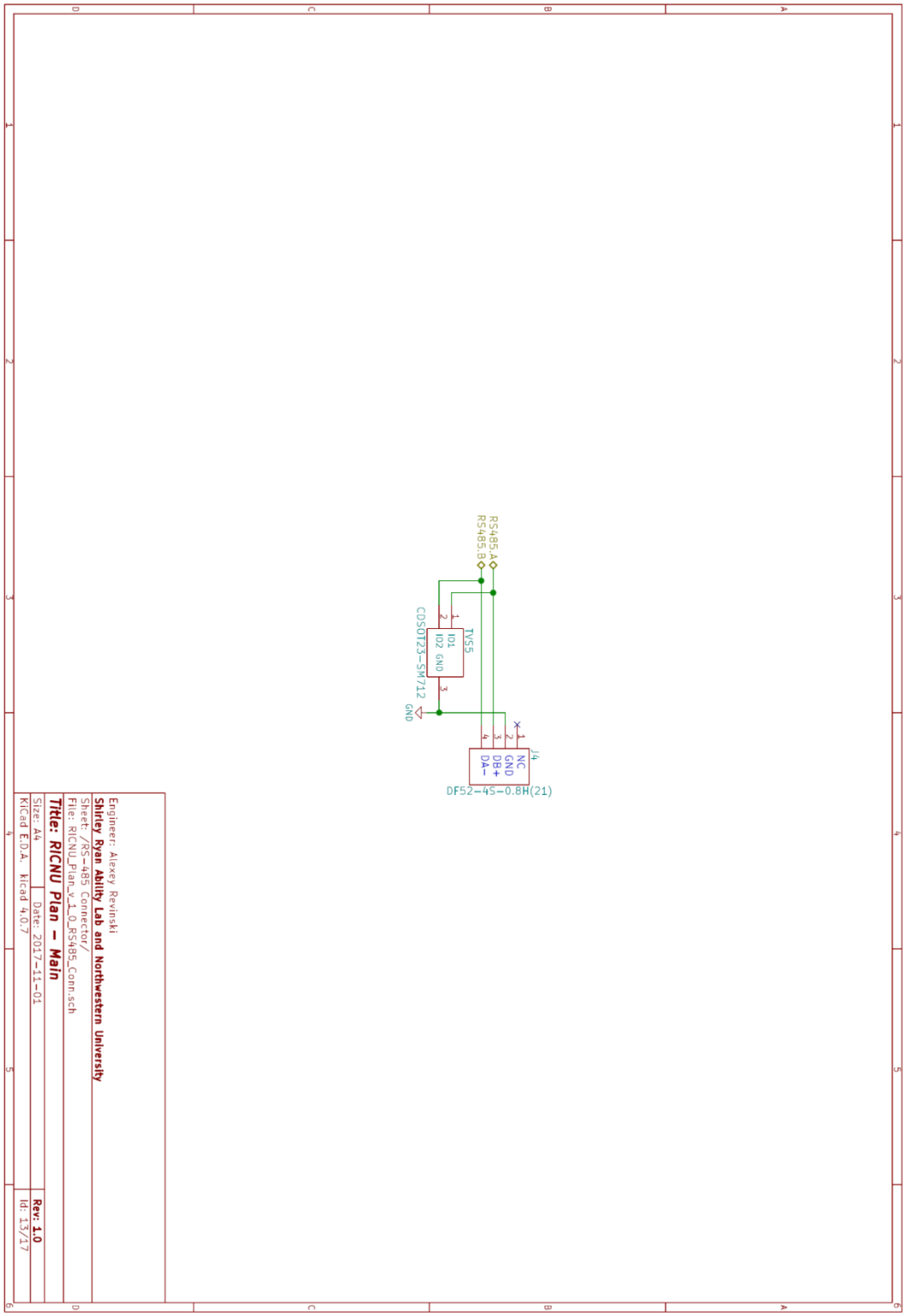

USB_SHIELD
PWR_FLAG
C24
47p
R67
4.7k
J1
USB_OTG
Shield
GND
VBUS
D+
D-
ID
+5V_USB
C25
47p
C26
47p
NP0/C0G
VBUS_RAW
GND_USB_CABLE
USB_DM
USB_DP
TVS1
TPD2EUSB30A
D+
D-
GND
F1
PTC_200mA
D1
5.0V TVS
Q1
N-CH
VBUS_PROTECTED
D2
5.6V
C27
10n
FB1
Ferrite Bead
C28
100n
C29
4.7u
GND
R68
0
+5V_USB_P
VBUS_PROTECTED_FILTERED
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /USB Connector/
File: RICNU_Plan_v_1_0_USB.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
KiCad E.D.A. kicad 4.0.7
Rev: 1.0
Id: 9/17

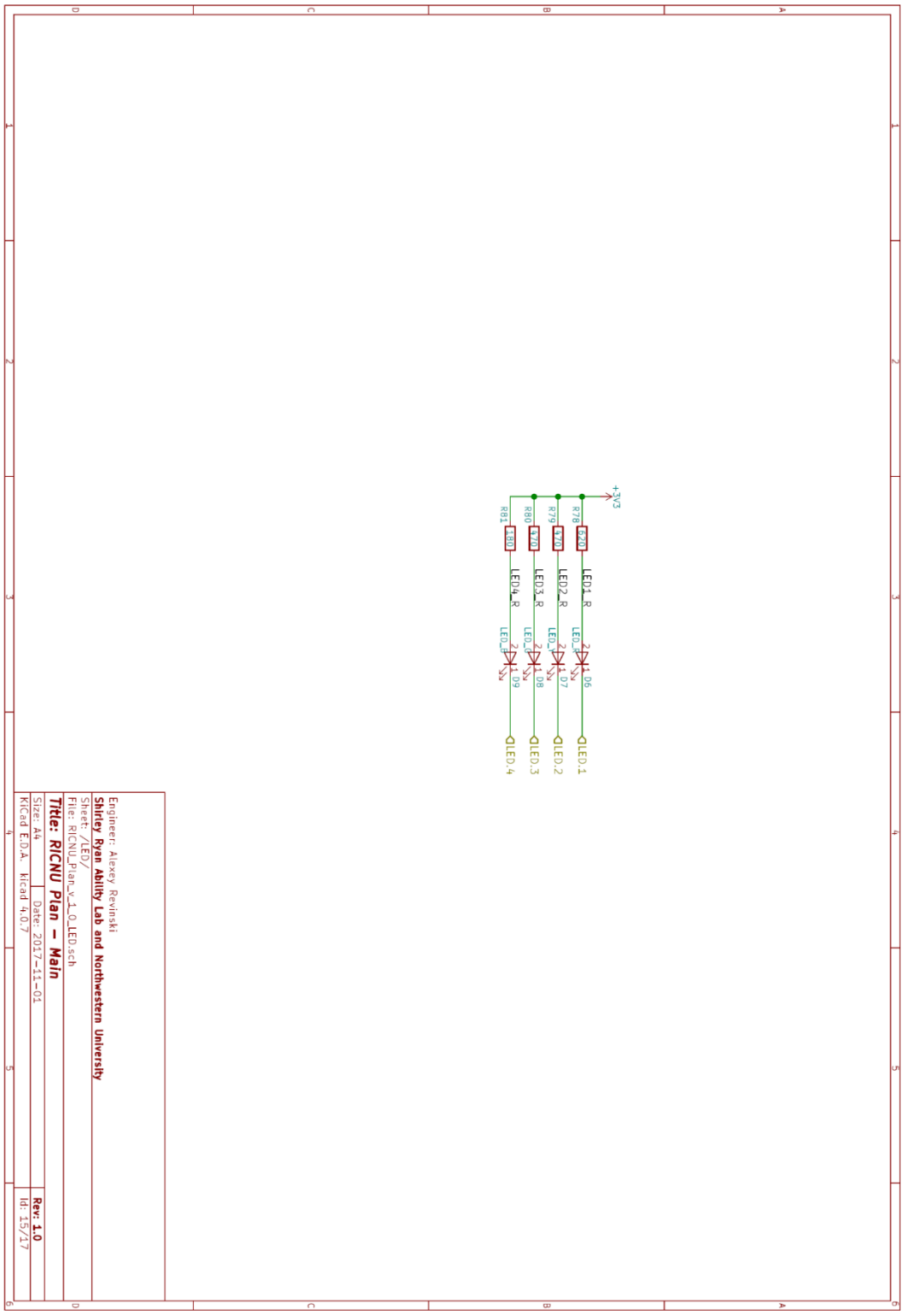

U7
BT121
GND
PB8
PB9
PB10
PB12
PB13
PB14
PB15
PB7
PB6
PB5
PB4
PB3
PA4
PA5
PA6
PA7
RESET
BOOT0
PA13
PA14
PA12
PA11
PA10
PA9
VDD
C11
100n
SW2
10k
R20
0
R21
C12
100n
+3V3
BT_RESET
BT_VCC
BTTX
BTRX
BTBOOT0
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /Bluetooth/
File: RICNU_Plan_v_1_0_Bluetooth.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
KiCad E.D.A. kicad 4.0.7
Rev: 1.0
Id: 7/17

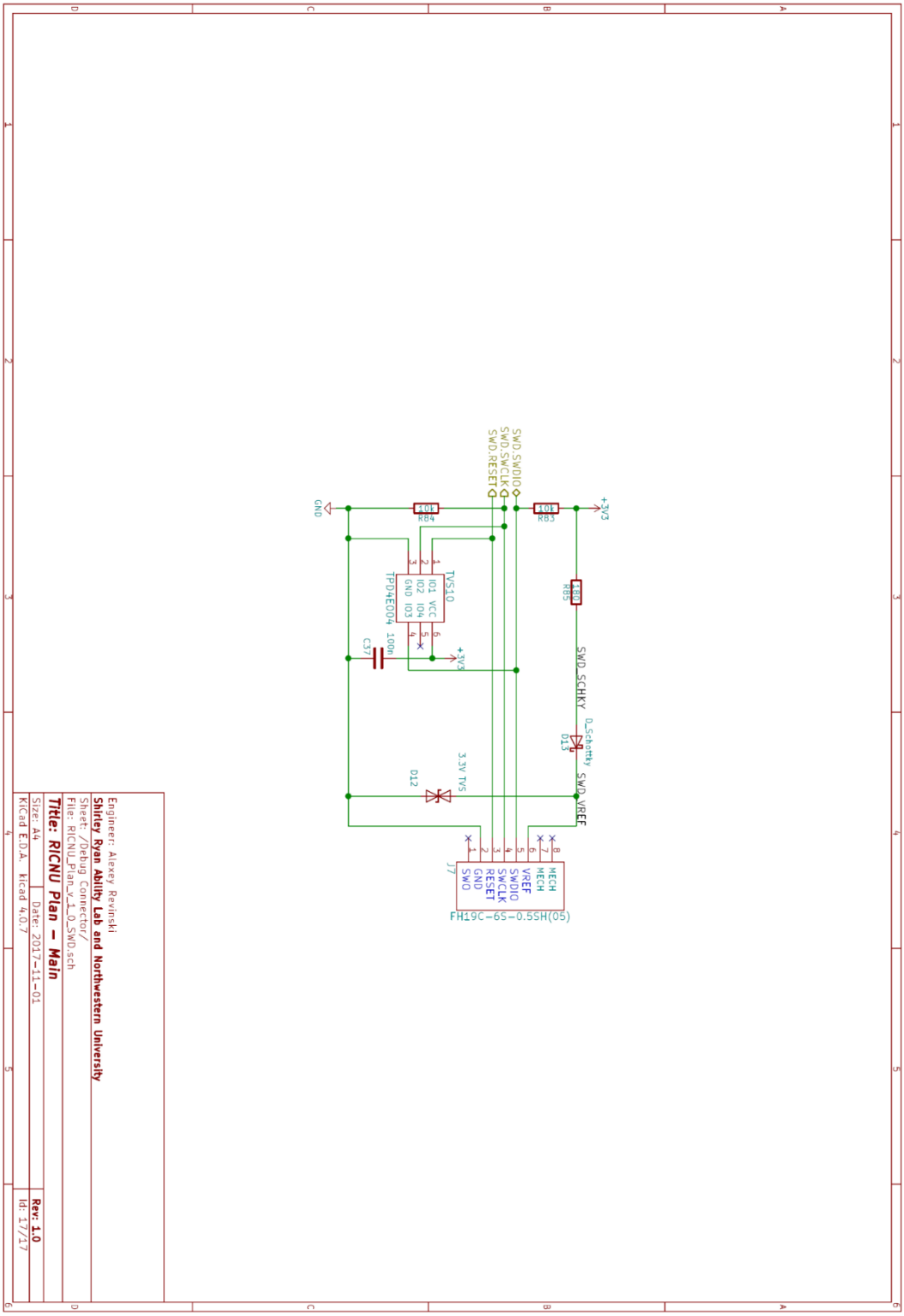

Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /Bus Driver/
File: RICNU_Plan_v_1_0_UARTDriver.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
KiCad E.D.A. kicad 4.0.7
Rev: 1.0
Id: 3/17
U3
SN74AHC126
MCU_UART_ENABLE
FTDI_UART_ENABLE
Break-Before-Make
SW1
CL-SB-12B-01T
LED_BT
LED_R
PWR_FLAG
DRIVER_VCC
GND
+3V3
FTDI_UART_RX_OUT
UMUX.BT.BOOT
UMUX.BT.TXD
UMUX.BT.RX
UMUX.MCU.RX
UMUX.MCU.TX
UMUX.U2U.RX
UMUX.U2U.TX
47k
R8
47k
R10
330
R9
660
R90
0
R11
100n
C6


CAN.SHDN
CAN.RX
CAN.TX
CAN.STB
CAN_VCC
GND
100n
C9
0
R14
10k
R95
+3V3
U5
TCAN334G
TXD
GND
VCC
RXD
STB
CANH
CANL
SHDN
120
R15
CAN.L
CAN.H
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /CAN Transceiver/
File: RICNU_Plan_v_1_0_CAN.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
KiCad E.D.A. kicad 4.0.7
Rev: 1.0
Id: 5/17


MNG_NSS
MNG_SCK
MNG_MISO
MNG_MOSI
GND
+3V3
10K
R75
10K
R74
10K
R76
TVS3
TPD4E004
IO1
IO2
GND
IO3
VCC
IO4
100n
C34
470
R77
MNG_SCHKY
D_Schottky
D4
3.3V TVS
D5
MNG_VREF
J2
VREF
NSS
SCK
MISO
MOSI
GND
MECH
MECH
MOLEX-PC6
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /Manage Connector/
File: RICNU_Plan_v_1_0_Manage.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
Rev: 1.0
KiCad E.D.A. kicad 4.0.7
Id: 11/17


RS485.A
RS485.B
TVS5
IO1
IO2
GND
CDSOT23-SM712
GND
J4
NC
GND
DB+
DA-
DF52-4S-0.8H(21)
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /RS-485 Connector/
File: RICNU_Plan_v1_0_RS485_Conn.sch
Title: RICNU Plan -- Main
Size: A4
Date: 2017-11-01
KiCad E.D.A. kicad 4.0.7
Rev: 1.0
Id: 13/17


+3V3
R81
330
R80
330
R79
330
R78
330
LED4_R
LED3_R
LED2_R
LED1_R
LED_B
LED_G
LED_Y
LED_R
D9
D8
D7
D6
LED:4
LED:3
LED:2
LED:1
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /LED/
File: RICNU_Plan_v_1_0_LED.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
Rev: 1.0
KiCad E.D.A. kicad 4.0.7
Id: 15/17


SWD:RESET
SWD:SWCLK
SWD:SWDIO
GND
+3V3
R84
10k
R85
10k
R86
4k99
TPD4E004
TVS10
GND
IO1
IO2
IO3
IO4
VCC
C37
100n
+3V3
SWD_SCHKY
D_Schottky
D13
SWD_VREF
3.3V TVS
D12
J7
FH19C-6S-0.5SH(05)
SWO
GND
RESET
SWCLK
SWDIO
VREF
MECH
MECH
Engineer: Alexey Revinski
Shirley Ryan Ability Lab and Northwestern University
Sheet: /Debug Connector/
File: RICNU_Plan_v_1_0_SWD.sch
Title: RICNU Plan – Main
Size: A4
Date: 2017-11-01
KiCad E.D.A. kicad 4.0.7
Rev: 1.0
Id: 17/17

Appendix D: Bill of Materials

	Device	Description	Package	#	Model	Manufacturer
PCB Components	C	CAP CER 0.1UF 16V X7R 0402	0402	21	EMK105B7104KVHF	Taiyo Yuden
	C	CAP CER 10000PF 16V X7R 0402	0402	2	EMK105B7103KV-F	Taiyo Yuden
	C	CAP CER 10UF 10V X7R 0805	0805	1	LMK212B7106KG-TD	Taiyo Yuden
	C	CAP CER 1UF 16V X7R 0805	0805	1	EMK212B7105KG-T	Taiyo Yuden
	C	CAP CER 4.7UF 16V X7R 0805	0805	5	EMK212B7475KG-T	Taiyo Yuden
	C	CAP CER 47PF 50V NP0 0402	0402	4	UMK105CG470JVHF	Taiyo Yuden
	C	CAP CER 7PF 50V NP0 0402	0402	3	UMK105CG070DVHF	Taiyo Yuden
	R	RES SMD 24K OHM 1% 1/10W 0402	0402	1	ERJ-2RKF2402X	Panasonic
	R	RES SMD 1K OHM 1% 1/5W 0402	0402	1	ERJ-PA2F1001X	Panasonic
	R	RES SMD 100K OHM 1% 1/5W 0402	0402	2	ERJ-PA2F1003X	Panasonic
	R	RES SMD 0 OHM 0402	0402	13	ERJ-2GE0R00X	Panasonic
	R	RES SMD 453K OHM 1% 1/10W 0402	0402	1	ERJ-2RKF4533X	Panasonic
	R	RES SMD 47K OHM 1% 1/10W 0402	0402	2	ERJ-2RKF4702X	Panasonic
	R	RES SMD 180 OHM 1% 1/10W 0402	0402	30	ERJ-2RKF1800X	Panasonic
	R	RES SMD 10K OHM 5% 1/10W 0402	0402	38	ERJ-2GEJ103X	Panasonic
	R	RES SMD 120 OHM 1% 1/4W 0603	0402	2	ERJ-PA3F1200V	Panasonic
	R	RES SMD 470 OHM 1% 1/10W 0402	0402	4	ERJ-2RKF4700X	Panasonic
	R	RES SMD 4.7K OHM 1% 1/5W 0402	0402	2	ERJ-PA2F4701X	Panasonic
	R	RES SMD 620 OHM 1% 1/10W 0402	0402	2	ERJ-2RKF6200X	Panasonic
	R	RES SMD 330 OHM 1% 1/10W 0402	0402	1	ERJ-2RKF3300X	Panasonic
	TVS	TVS Diodes 4Ch ESD Protection Array	6-SON (1.45x1)	7	TPD4E004DRYR	Texas Instruments
	TVS	TVS Diodes SOT-23 7V 600W Low Capacitance	SOT-23	1	CDSOT23-SM712	Bourns
	TVS	TVS Diodes 24V CAN bus Protector AEC-Q101	SOT-23	1	CDSOT23-T24CAN-Q	Bourns
	TVS	TVS Diodes 2Ch ESD Solution DRT-3	DRT-3	1	TPD2EUSB30DRTR	Texas Instruments
	TVS	5V TVS	SOD-523	2	PESD5V0S1BB,115	Nexperia
	TVS	3.3V TVS	SOD-523	4	PESD3V3S1UB,115	Nexperia
	Schottky	Schottky Diode 500mA	SOD-323	2	CUS05S40,H3F	Toshiba
	Zener 5.6V	Zener Diodes 3W 5.6V	SMA	2	3SMAJ5919B-TP	MCC
	LED	Red	0402	2	APHHS1005LSECK/J3-PF	Kingbright
	LED	Yellow	0402	1	APHHS1005LSYCK/J3-PF	Kingbright
	LED	Green	0402	1	APHHS1005LZGCK-V	Kingbright
	LED	Blue	0402	1	APHHS1005LQBC/D-V	Kingbright
	PTC Fuse	PTC RESET FUSE 9V 200MA 0603	0603	2	FEMTOSMDC020F-2	Littelfuse Inc.
	PTC Fuse	PTC RESET FUSE 9V 160MA 0603	0603	1	FEMTOSMDC016F-2	Littelfuse Inc.
	Ferrite Bead	Ferrite Bead 0402 30mOhm DC	0402	1	BKP1005HS100-T	Taiyo Yuden
	Connector	Single Row 6 Pin Receptacle	Custom	1	501568-0607	Molex
	Connector	Single Row 6 Pin Receptacle	Custom	2	DF52-6S-0.8H(21)	Hirose
	Connector	microUSB B Receptacle SM	Custom	1	105017-1001	Molex
	Connector	Dual Row 20 Pin Receptacle	Custom	1	501571-2007	Molex
	Connector	Single Row 6 Pin Receptacle	Custom	1	FH19C-6S-0.5SH(05)	Hirose
	L	2.2uH 28.6mOhms +/-30%Tol 3.5A	Custom	1	NR5040T2R2N	Taiyo Yuden
	MOSFET	MOSFET 20V 6A N-CH	SOT-23-3	2	SI2312CDS-T1-GE3	Vishay Siliconix
	Power MUX	5V Power MUX	TSSOP-8	1	TPS2113PWR	Texas Instruments
	SMPS Buck	1-A High Efficiency Buck Converter	SOT-23-5	1	TLV62568DBVR	Texas Instruments
	3-State Buffer	Quad Tri-State Buffer	TSSOP-14	1	SN74AHC126PWR	Texas Instruments
	USB2UART	Full Speed USB to UART interface	SSOP-28	1	FT232RL	FTDI Chip
	CAN TX/RX	3.3-V CAN Transceiver with CAN FD	SOT-23-8	1	TCAN334GDCNR	Texas Instruments
	RS485 TX/RX	3.3 V Half-Duplex RS-485, with IEC ESD, 20 Mbps	VSSOP-8	1	SN65HVD75DGKR	Texas Instruments
	Bluetooth SoC	Bluetooth Smart Ready Module SoC	Custom	1	BT121-A-V2	Bluegiga
	32-bit MCU	MCU 32BIT Cortex M3 72MHz 64pins	LQFP-64	1	STM32F103RET6	ST Microelectronics
	kHz Crystal	32.768 kHz 6pf 2-Lead Crystal	3.2x1.5	1	ECS-.327-6-34RR-TR	ECS Inc
	SD Card Socket	microSD Card Socket Push-Pull	Custom	1	DM3AT-SF-PEJM5	Hirose
	Slide Switch	Break Before Make Slide Switch	Custom	1	CL-SB-12B-01T	Nidec Copal
	Push button	Microminiature SMT Push Button (Top) IP67	Custom	2	KMR731NG LFS	C&K Components
Extras	Plug	Plug for 6-pin Molex	Custom	1	501330-0600	Molex
	Plug	Plug for 20-pin Moles	Custom	1	501189-2010	Molex
	Plug	Plug for 6-pin DF-52	Custom	2	DF52-6P-0.8C	Hirose
	Cable	FPC Cable 6 conductions	Custom	1	687606050002	Wurth Electronics
	PCB	JTAG to SWD adapter PCB	Custom	1	2094	Adafruit
	Cable	SWD Cable 10 2x5 1.27mm 150mm	Custom	1	1675	Adafruit
	Debugger	STM32 /STM8 Debugging Interface	Custom	1	ST-LINK/V2	ST Microelectronics

Appendix E: RICNU/FlexSEA Communication Packets

TX:

Byte	0	1	2 through N+1	N+2	N+3	N+4
Value	0xED	N	payload[0]-payload[N-1]	0-127		0xEE
Meaning	Header	Payload length	Payload bytes	Packet ID	Checksum	Footer

Byte	2	3	4	5	6 through 13
Value	0x0A (Plan1)	0x28 (Exec1)	1	R/W:1/2	
Meaning	TXID	RXID	Num CMDs	CMD	Data Buffer

Byte	6	7	8	9	10	11	12	13
Value	1/2/3	0/1/2/3/4		K/C:0/1				
Meaning	Offset	Control	Setpoint	Set gains	G0	G1	G2	G3

RX:

Byte	0	1	2 through N+1	N+2	N+3	N+4
Value	0xED	N	payload[0]-payload[N-1]	0-127		0xEE
Meaning	Header	Payload length	Payload bytes	Packet ID	Checksum	Footer

Byte	2	3	4	5	6 through X
Value	0x28 (Exec1)	0x0A (Plan1)	1	R/W:1/2	
Meaning	TXID	RXID	Num CMDs	CMD	Data Buffer

Byte	6	7,8	9,10	10,11	11,12	13,14	15,16	17-20	21-24	25,26	27-30
Value	0										
Meaning	Offset	GX	GY	GZ	AX	AY	AZ	EM	EJ	CU	----

or

Byte	6	7 – 15 (9 bytes)
Value	1	
Meaning	Offset	FX, FY, FZ, MX, MY, MZ (12-bit values)

Controller	ID	Parameter meaning	
None	0	Setpoint	---
		G0	---
		G1	---
		G2	---
		G3	---
Open	1	Setpoint	PWM: -1024 to 1024
		G0	---
		G1	---
		G2	---
		G3	---
Position	2	Setpoint	Desired position in encoder clicks (degrees*45)
		G0	Kp: proportion term = Kp*(position error in clicks)/100
		G1	Ki: integral term = Ki*(sum of position error in clicks at 1kHz)/100
		G2	---
		G3	---
Current	3	Setpoint	Desired current in mA
		G0	Kp: proportion term = Kp*(position error in clicks)/100
		G1	Ki: integral term = Ki*(sum of position error in clicks at 1kHz)/100
		G2	---
		G3	---
Impedance	4	Setpoint	Equilibrium position in encoder clicks (degrees*45)
		G0	Spring constant K: spring current = (deflection in clicks)*K/16
		G1	Damping constant B: damping current = (encoder velocity in clicks per ms) *B
		G2	Current controller Kp
		G3	Current controller Ki

Appendix F: RICNU Plan/User Communication Packets

TX:

Byte	0	1,2	3,4	5,6	7,8	9,10	11,12	13-16	17-20	21,22	23,24
Value	0	raw	raw	raw	raw	raw	raw	raw	raw	raw	raw
Meaning	Offset	GX	GY	GZ	AX	AY	AZ	EM	EJ	CU	----

Byte	0	1-4	5-8	9-12	13-16	17-20	21-24
Value	1	decompressed	decompressed	decompressed	decompressed	decompressed	decompressed
Meaning	Offset	GX	GY	GZ	MX	MY	MZ

RX:

Byte	0	1
Value		0x00
Meaning	command	----

Bit	7	6	5,4	3-0
Value	STRT/STP:1/0	ON/OFF:1/0	CALIB/RELAX:0x03/0x02	0x0
Meaning	CTRL	LOG	CMD	----