rcHex: A Radio-controlled Hexapod

A Senior Project Report

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Dominique Sayo

June 2020

Advisor: Dr. Paul Hummel

Abstract

rcHex: A Radio-controlled Hexapod

Dominique Sayo

rcHex is a radio-controlled hexapod with 18 degrees of freedom, capable of positional and rotational body adjustment as well as omnidirectional travel at variable speeds using three common gaits. Its general-purpose design accessible to hobbyists makes rcHex an platform for further development, whether it be experimentation in advanced robotic movement or retrofitting sensors to utilize technologies such as computer vision and artificial intelligence. This report explores some of the design intricacies of hexapod movement, including gait sequencing and the application of inverse kinematics to multi-jointed limbs.

Contents

List of Tables

List of Figures

1 Introduction

Designing and building a hexapod allows for the exploration of topics regarding radio-controlled robotics, including inverse kinematics and overall system design that is is capable of expansive development. The electronic design allows for modular use of common hobbyist radios and batteries, and the embedded software provides a framework that makes it simple to add or modify hexapod functionality.

1.1 Stakeholders

Hexapod research and development is of interest to those in the field of robotics, whether they be hobbyists, educators, or researchers exploring biologically-inspired locomotion. Additionally, engineers of various backgrounds can expand upon or adapt features towards more specific applications.

1.2 Goals and Objectives

The goal of this project is to design and build the electrical and software components of a mechatronic system from the ground up while using professional embedded development and engineering practices. These practices include searching for and incorporating compatible components, integrating several subsystems to create a functional product, and maximizing the maintainability of a code base over the course of a project. Regarding project management, a timeline with milestones must be established and followed while regularly communicating progress with an advisor.

1.3 Project Deliverables

Deliverables during the project are demonstrations of functional subsystems that accompany the milestones shown in Table 1.1. After completion of the last milestone, the radio-controlled hexapod will be able to adjust body position and orientation in place as well as travel omnidirectionally using three types of gaits.

Table 1.1: Project milestones

Milestone	Description	Demonstration
MS-01	TX/RX protocol	Live control data on terminal
MS-02	Servo controller library	Simultaneous control of servos in one leg
MS-03	Stationary inverse kinematics	Hexapod moving with singular IK commands
MS-04	Gait sequencing	Crawling with LED timing indicators
MS-05	End product	All implemented functionality

1.4 Project Outcomes

The completion of this project allows for further development of legged robotic movement on complex terrain. Although the project only goes as far to develop a solely human-controlled hexapod without the use of any environmental or operational feedback, addition of technologies such as computer vision, artificial intelligence, and the use of sensors can offer insight to the intricacies of legged biological movement.

2 Background

Accompanied with sensors and other technologies, robotic limbs have several useful applications in which they can sometimes outperform humans. One common application is automation of tasks such as assembly line work, moving objects, or spot welding. In the biomedical field, prosthetic limbs are being developed to function as if organic muscles were present [2]. In the space industry, robotic arms are being used for inspection of spacecraft damage, maintenance, and moving payloads. The Canadarm used on previous space shuttles and the Canadarm2 currently on the International Space Station are a few valuable and well-known examples [1]. Furthermore, groups of robotic limbs can be used to coordinate terrestrial movement. Boston Dynamics, a robotics company, is well-known for designing robots with life-like movement. Their product videos show advanced feats such as humanoid robots performing handstands and parkour rolls. Their versatile remote-controlled quadruped robot, Spot, was made commercially available in 2019 to pursue the vision of hardened robots working in the field [3]. Although far from the forefront of the field of robotics, the design of a hexapod will explore a basic level of robotic movement and serve as an entry-point to those interested in such an inspiring field.

3 Formal Project Definition

The following sections outline the project requirements. At each milestone in the design process, these requirements are reviewed and verified through testing, inspection, and analysis.

3.1 Customer Requirements

Customers and users require that the hexapod shall:

- be bindable from any FrSky-compatible radio system
- be powered by battery
- be capable of adjusting body orientation and position while stationary
- be capable of omnidirectional travel with controllable crawl speed
- have fluid motion
- have easily modifiable and expandible behavior though its code base

3.2 Engineering Requirements

Table 3.1 outlines the engineering requirements with tolerance values and risk levels. These requirements are targets and metrics suitable for fulfilling the above customer requirements. Some are chosen for the accessibility of and compatibility with commonly used RC hobby equipment.

Table 3.1: Engineering requirements

Spec	Parameter Description	Target	Tolerance	Risk	Compliance*
1	Number of RX channels	8	Min	High	I
2	Battery voltage	6 V	±1	Med.	S
3	Battery life	5 min	Min	Med.	I, T
4	Degrees of freedom per leg	3	±0	Low	I, S
5	Control loop refresh rate	30 Hz	Min	High	A, T
6	Maximum crawl velocity	0.1 m/s	Min	Low	I, T

*(A)nalysis, (T)est, (S)imilarity to existing designs, and/or (I)nspection.

3.3 End-User Personas

Two personas were developed to assist with design approach:

1. A RC hobbyist has radio equipment and batteries used for their other RC models and want an addition to their fleet. They like to fine-tune controls and adjust parameters whether it be for functional improvements or just for fun.
2. An engineering team is tasked with adapting hexapods for specific applications involving moving a payload over stretches of complex terrain for transportation, sensing, or terrain mapping purposes. They need to be able to modify and handoff hexapods to other teams who deploy them in the field for different scenarios.

3.4 Use Cases

Figure 3.1 shows the basic use cases of the hexapod as a product line in a mobile robotics service. Customers can lease or purchase the hardware and operate the hexapod. The business can modify the hexapod according to customer needs while offering continual support service of the product.

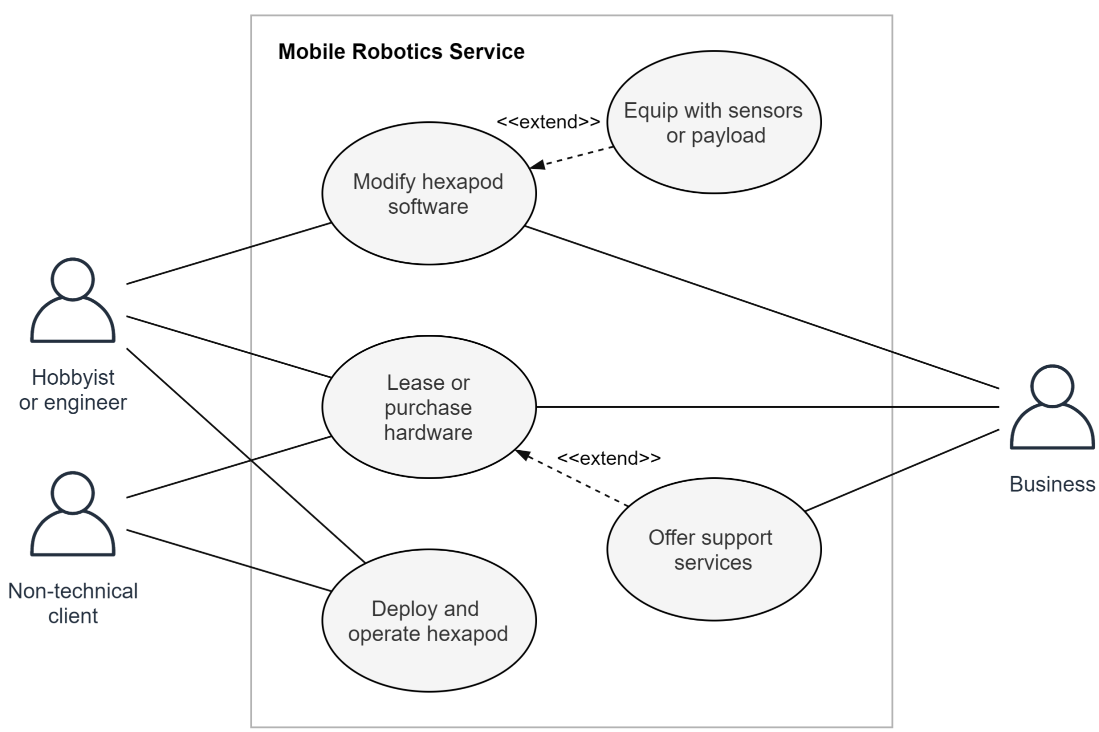


Figure 3.1: Use cases in a mobile robotics service

4 Design

The following sections cover the hexapod's mechanical, electrical, and software subsystems as well as the main technological concepts dedicated to each milestone.

4.1 Mechanical Design

With six legs each having three degrees of freedom (DoF), the hexapod uses 18 servos total. HS-645MG servos were selected due to their specifications suitable for mobile robotics, such as torque and operating volage. An appropriate frame was chosen, with proper-fitting servo brackets and enough capacity to manage cables and mount electronics and a battery. Figure 4.1 shows a single leg labelled with arthropod leg nomenclature. Additionally, servos were installed into the brackets such that the 90° position is aligned with the neutral stance of the robot shown in the figure. Measurements of the hexapod geometry were taken post-assembly for constants used in inverse kinematics calculations, such as body dimensions and the lengths of the coxa, femur, and tibia. A top plate to mount electronics on top of the hexapod frame was designed and 3D printed.

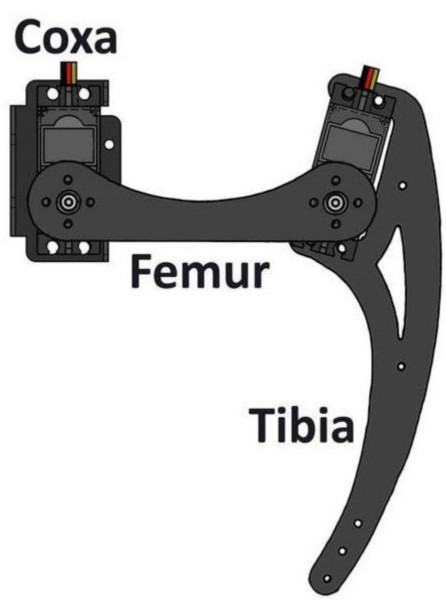


Figure 4.1: Hexapod leg structure

4.2 Electrical Design

The STM32F303K8 was chosen as the MCU because of its extensive range of peripherals and small development board footprint. A FrSky XM+ receiver was chosen due to its UART-compatible communication protocol as well as its compatibility with popular RC transmitters. Lastly, the SSC-32U servo controller board was selected to manage the PWM signals to the 18 servos. Figure 4.2 shows the system schematic connecting these three main components. Additional components are an external LED showing the armed status, and a UBEC (switching voltage regulator) to maintain a stable voltage from the battery pack to the MCU and receiver. The servos are directly powered from the battery through the VS1+ and VS1- rails on the SSC-32U.

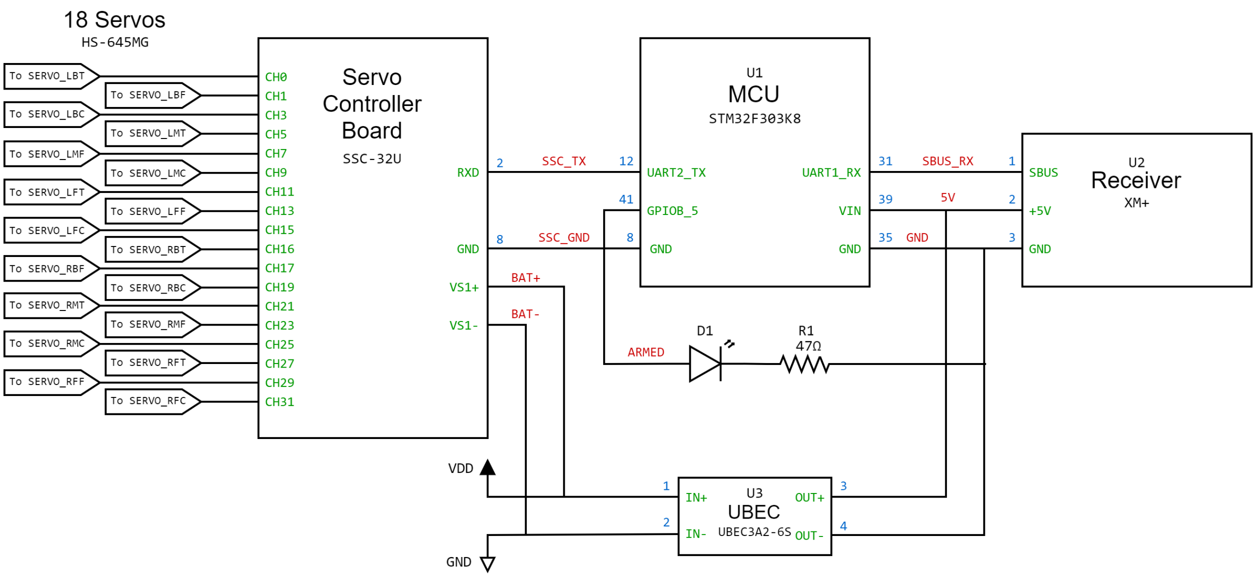


Figure 4.2: Electronics system schematic

4.3 Software Design

In addition to writing libraries to communicate with the receiver and servo controller, a large software focus was to create a robust framework that allows for development towards additional features such as using more receiver channels, gaits, and other hexapod movements or actions.

Interrupts were used for two important timing decisions in the main control loop. First, a DMA interrupt signals a complete SBUS packet reception. The data processing flow is shown in Figure 4.3. Data is parsed according to the channels assigned to the armed and mode switches, and movement is executed according to the status of those switches.

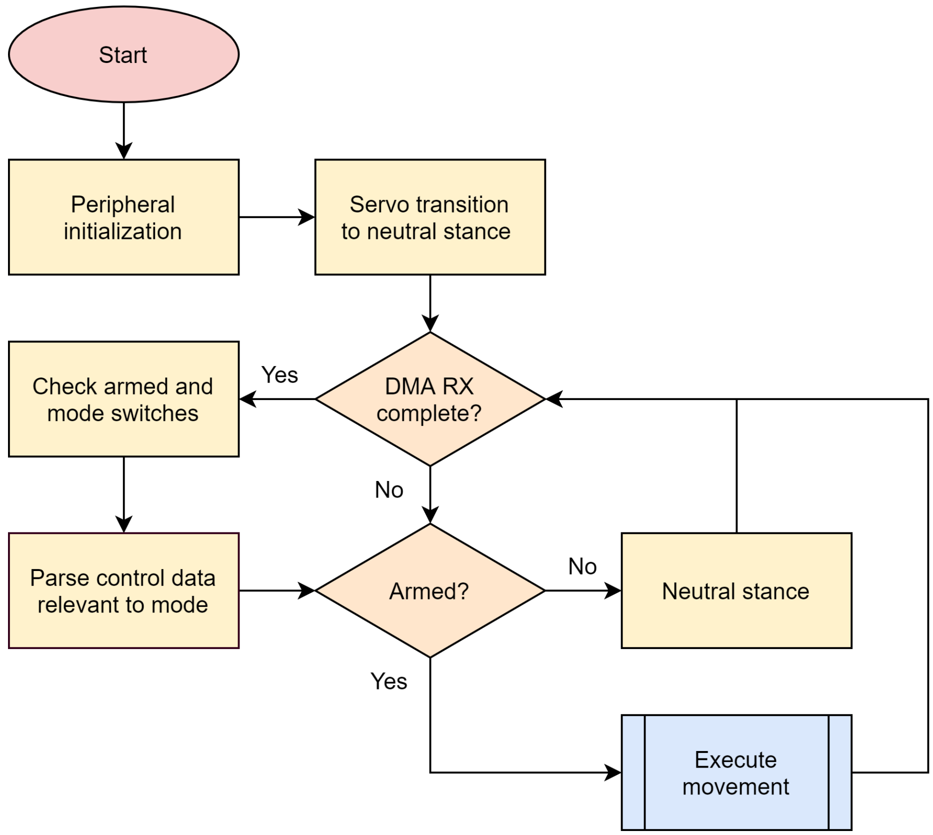


Figure 4.3: Flowchart for main and control data processing

The second interrupt occurs from a timer and is used for the gait sequencer. These interrupts occur at a variable frequency according to the movement speed of the hexapod. Each timer interrupt signals a change in the leg position in the gait sequence. The hexapod can also operate in a stationary mode which more clearly demonstrates the capabilities of inverse kinematics. Figure 4.4 shows the decisions made in the movement execution portion of the main control loop.

Because the control loop includes several components such as parsing receiver data, computing inverse kinematics servo angles, and sending commands to the servo controller, operational smoothness was a continually addressed concern. Any unnecessarily slow subroutine could possibly result in choppy movement.

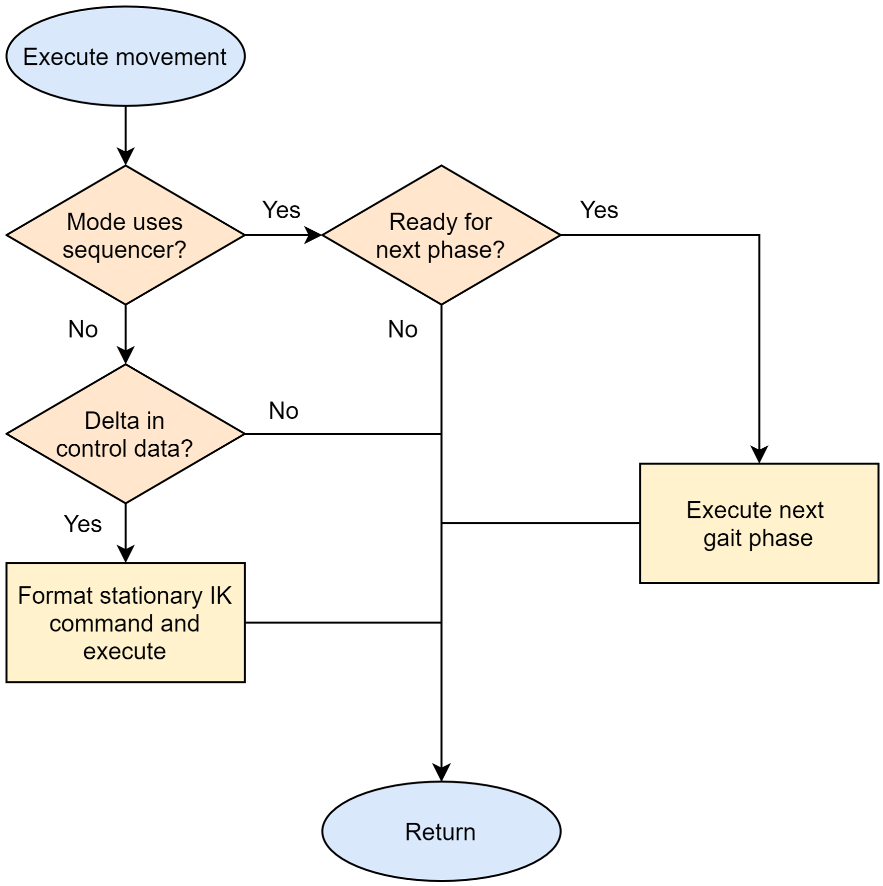


Figure 4.4: Flowchart for movement execution

4.4 TX/RX Protocol and Radio Control

SBUS is a serial communication protocol derived from RS232 and is used in RC receivers manufactured by FrSky. After the RX signal is inverted, it can be processed with UART at a non-standard baud rate of 100000 bits per second. The UART peripheral on the STM32F303 has configuration options that allow for receiving SBUS data without an external inverting circuit, which is common in other RC systems using SBUS receivers.

One SBUS packet contains 16 channels of 11-bit data and one byte of flags. The 25-byte packet format is shown in Figure 4.5. The flags include two digital channels and a failsafe bit, but are not currently used for the hexapod. A UART peripheral was configured on the MCU to receive data and transfer to memory with DMA, interrupting every 25 bytes to signal a complete refresh of control data.

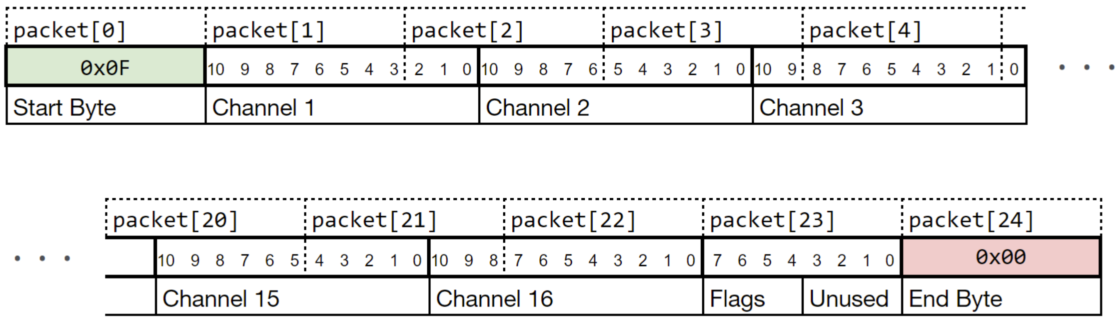


Figure 4.5: SBUS packet format

Transmitters can configure and assign switches and sticks to receiver channels. Figure 4.6 shows the Taranis Q X7's available control options. This transmitter allowed for fine tuning such as changing sensitivity or adjusting the curve that the raw data follows when moving a stick or potentiometer. This allowed for flexible configuration on the transmitter side that did not have to be done in software. Table 4.1 shows the final configuration for the Q X7 that was decided to effectively switch between hexapod modes and features. Stationary mode is split into Roll/Pitch and X/Y mode in order to demonstrate all six inverse kinematic command arguments, which are discussed in Section 4.5. S1 and S2 are unused potentiometers available for use in future features. Table 4.2 shows the functions of stick axis movement when the hexapod is in each mode.

Table 4.1: Radio switch functions

Name	Function	Position			Notes
		1	2	3†	
SA	Reserved	—	—	—	
SB	Reserved	—	—	—	
SC	Gait Select	Tripod	Ripple	Wave	Crawl mode only.
SD	Mode	Roll/Pitch	X/Y	Crawl	
SF	Arm	Disarmed	Armed	—	
SH	Rotate Mode	Off	On	—	Overrides SD if on.

† Switches SA, SB, SC, and SD are three-position switches. SF and SH are two-position switches.

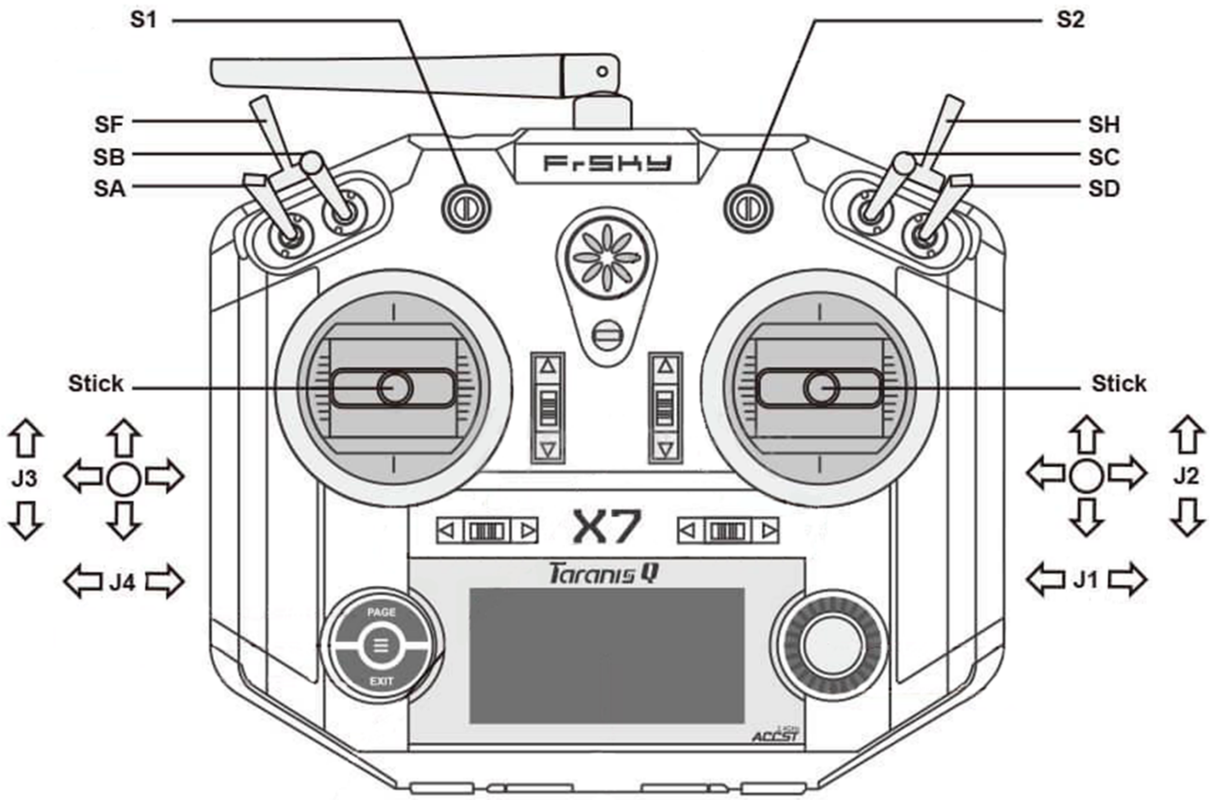


Figure 4.6: Q X7 channel labels

Table 4.2: Radio stick functions

Name	Mode			
	Roll/Pitch	**X/Y**	**Crawl**	**Rotate**
J1	Roll	Body x	Crawl x	—
J2	Pitch	Body y	Crawl y	—
J3	Body z	Body z	—	—
J4	Yaw	Yaw	—	Rotation

4.5 Inverse Kinematics

Inverse kinematics is important to the fluidity of multi-jointed movement. In forward kinematics, an input of several joint angles outputs a final position of the end effector or the end of the robotic arm. The process of inverse kinematics reverses this relationship such that providing the cartesian coordinates of the desired end effector position returns possible sets of joint angles to achieve such a position. In anticipation of the task of gait coordination, using inverse kinematics was deemed essential for smooth movement; providing cartesian coordinates for foot positions would be much more effective than a guess-and-check method of supplying individual servo angles in a forward kinematics approach.

Solving inverse kinematic problems can be complicated when dealing with higher numbers of joints, but fortunately each leg in an 18-DoF hexapod only have three joints moving in three-dimensional space. To derive the inverse kinematic equations for one hexapod leg, the three dimensional axes must be defined: the x-axis spans left to right of the hexapod, the y-axis spans backward to forward, and z-axis from below to above. First, let's view the hexapod from above, using Figure 4.7 to observe the hip movement of the coxa servo in order to achieve foot position $p = (x, y, z)$.

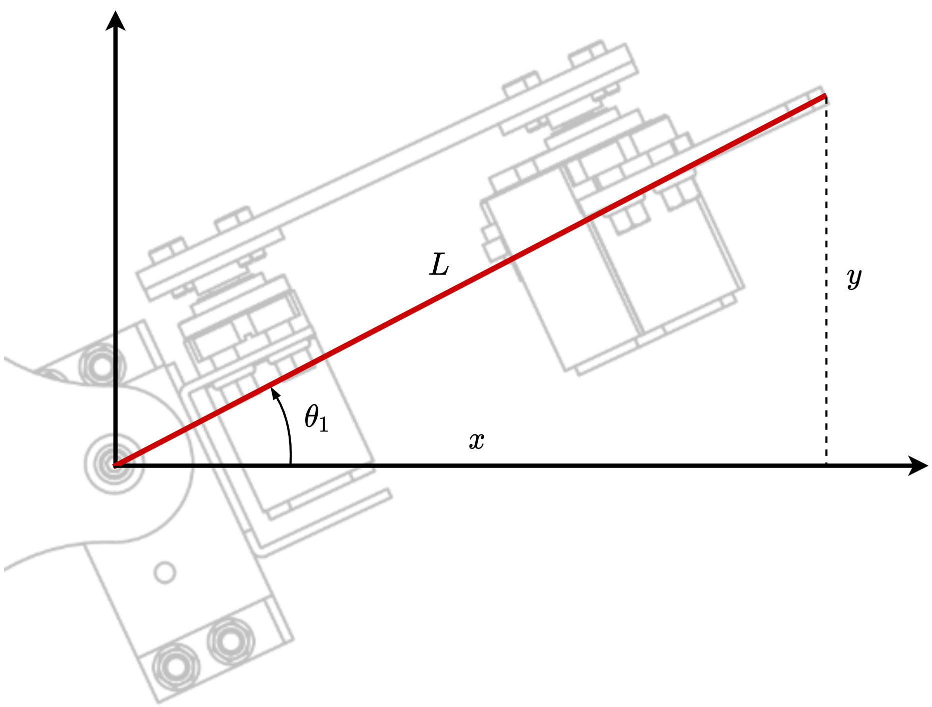


Figure 4.7: Inverse kinematics derivation on the coxa plane

Right away, coxa angle θ_1 can be determined:

$$\tan\theta_1 = \frac{y}{x}$$
$$\theta_1 = \tan^{-1}\left(\frac{y}{x}\right)$$

For the next steps, the total length L of the leg from this perspective is obtained by the Pythagorean theorem:

$$L = \sqrt{x^2 + y^2}$$

The perspective in Figure 4.8 observes the plane of rotation of the femur and tibia servos.

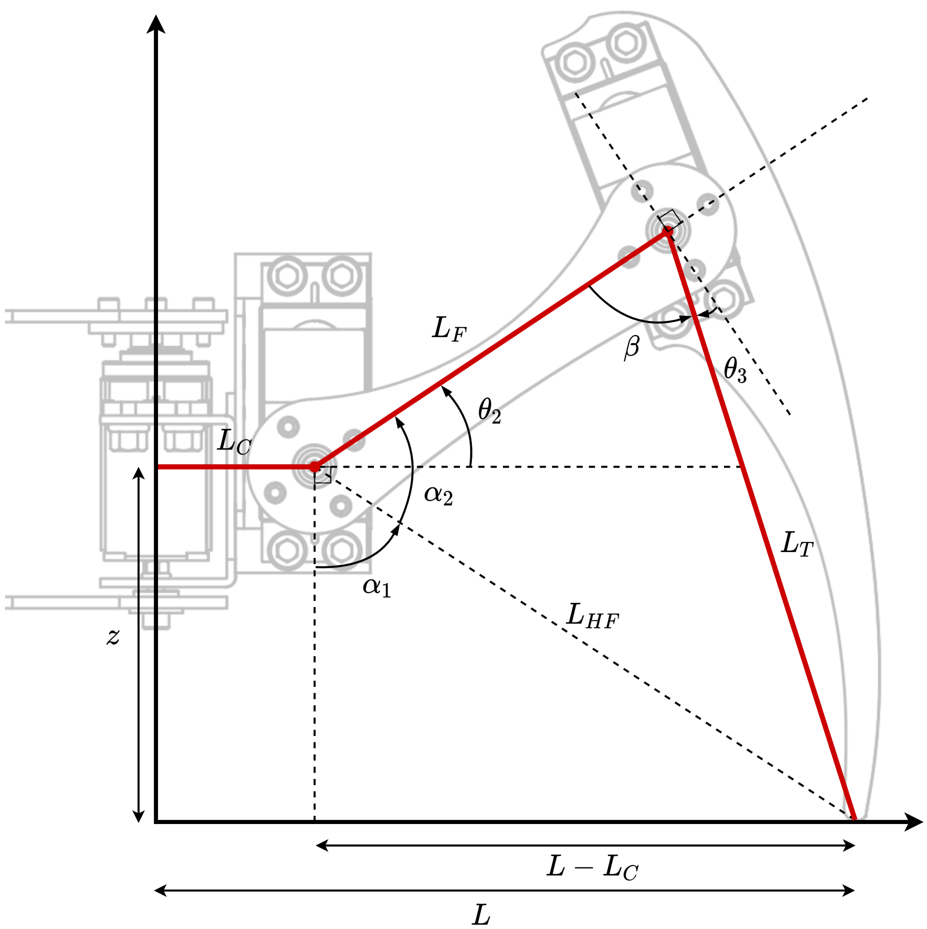


Figure 4.8: Inverse kinematics derivation on the femur-tibia plane

L_C, L_F, and L_T are physical constants, respectively representing the lengths of the coxa, femur, and tibia. The triangles formed by these lengths will be analyzed to find the femur and tibia angles θ_2 and θ_3.

To split the current geometry into two triangles, find L_{HF}, the length from the hip to the foot using Pythagorean theorem:

$$L_{HF} = \sqrt{(L - L_C)^2 + z^2}$$

Angle α_1 of the lower triangle can be found:

$$\begin{aligned} \tan \alpha_1 &= \frac{L - L_C}{z} \\ \alpha_1 &= \tan^{-1}\left(\frac{L - L_C}{z}\right) \end{aligned}$$

Now that three side lengths of upper triangle $L_F L_T L_{HF}$ are known, the law of cosines can be used to obtain angles α_2 and β:

$$\begin{aligned} L_T^2 &= L_F^2 + L_{HF}^2 - 2L_F L_{HF} \cos \alpha_2 \\ \alpha_2 &= \cos^{-1}\left(\frac{L_F^2 + L_{HF}^2 - L_T^2}{2L_F L_{HF}}\right) \end{aligned}$$

$$\begin{aligned} L_{HF}^2 &= L_F^2 + L_T^2 - 2L_F L_T \cos \beta \\ \beta &= \cos^{-1}\left(\frac{L_F^2 + L_T^2 - L_{HF}^2}{2L_F L_T}\right) \end{aligned}$$

Servo angles for the femur and tibia, θ_2 and θ_3, are calculated using the calculated angles and their relation to right angles:

$$\begin{aligned} \theta_2 &= \alpha_1 + \alpha_2 - 90° \\ \theta_3 &= 90° - \beta \end{aligned}$$

The final three servo angles $q = (\theta_1, \theta_2, \theta_3)$ needed to achieve foot position $p = (x, y, z)$ are:

$$\begin{aligned} \theta_1 &= \tan^{-1}\left(\frac{y}{x}\right) \\ \theta_2 &= \tan^{-1}\left(\frac{L - L_C}{z}\right) + \cos^{-1}\left(\frac{L_F^2 + L_{HF}^2 - L_T^2}{2L_F L_{HF}}\right) - 90° \\ \theta_3 &= 90° - \cos^{-1}\left(\frac{L_F^2 + L_T^2 - L_{HF}^2}{2L_F L_T}\right) \end{aligned}$$

Total control requires applying this set of equations to all six legs with the appropriate offsets from the center of the body. For roll, pitch, and yaw, the change in position is added after rotational trigonometry is applied. In doing so, a singular command of three-dimensional coordinates (x, y, z), roll, pitch, and yaw is used to control the position and orientation of the hexapod body.

Using inverse kinematics involves using trigonometric operations on a microcontroller and results in a tradeoff of performance and smoothness of hexapod motion. Anticipating slow computation in the overall process cycle, trigonometry function lookup tables were prepared. However, the `math.h` library functions and the FPU on the STM32F303 were fast enough such that these improvements were not necessary. See section 5.1 for the timing measurements and analysis.

4.6 Gait Sequencing

The idea of gait coordination stems from applying different inverse kinematics commands to subsets of legs as opposed to applying the same command to all six legs in the stationary modes. The hexapod gaits are implemented as finite state machines with each state representing a set of leg positions, so faster cycling through these states results in faster hexapod movement. The gait sequencer uses a timer on the STM32 for state changes, interrupting at a frequency proportional to the magnitude of the receiver channels assigned to movement.

Three common hexapod gaits were implemented: tripod, ripple, and wave gait. The rotation gait is derived from the tripod gait, using yaw instead of lateral movement. In a gait cycle or stride, a leg is in one of two phases: the stance phase and swing phase. Figure 4.9 shows diagrams of the states and the corresponding phases of each leg in each gait. Red segments represent the stance phase, during which the foot is in contact with the ground supports weight. Blue segments represent the swing phase, during which the foot is not in contact with the ground to return to a position to begin the stance phase again.

The states are labelled A through F to cover the lengths of stance and swing phases. Each lettered state is split into three sub-states because the swing phase requires three steps of raising a foot off of the ground, moving it across in mid-air, and placing it back down. The stance phase is divided evenly between the rest of the sub-states in the cycle with equidistant foot positions between the end and start of the swing phase to achieve a smooth transition.

Ripple gait is slightly more complicated, as the beginning of the swing phases are offset between the left and right legs of the hexapod. Twice the amount of states are used such that a left leg can begin its swing phase precisely in the middle of its counterpart's swing phase. There are many possibilities in six-legged locomotion aside from these three gaits, so this state-based model can be used to plan and implement other movement patterns.

Swing phase
Stance phase

Tripod/Rotate

Leg	A1	A2	A3	B1	B2	B3	A1	A2	A3	B1	B2	B3	A1	A2	A3	B1	B2	B3
1: Right front				1	2	3												
2: Right middle	1	2	3															
3: Right back				1	2	3												
4: Left back	1	2	3															
5: Left middle				1	2	3												
6: Left front	1	2	3															

Ripple

Leg	A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	D3	E1	E2	E3	F1	F2	F3
1: Right front		6								1		2		3		4		5
2: Right middle				1		2		3		4		5		6				
3: Right back		3		4		5		6								1		2
4: Left back							1		2		3		4		5		6	
5: Left middle	4		5		6								1		2		3	
6: Left front	1		2		3		4		5		6							

Wave

Leg	A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	D3	E1	E2	E3	F1	F2	F3
1: Right front				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2: Right middle	10	11	12	13	14	15				1	2	3	4	5	6	7	8	9
3: Right back	4	5	6	7	8	9	10	11	12	13	14	15				1	2	3
4: Left back	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15			
5: Left middle	7	8	9	10	11	12	13	14	15				1	2	3	4	5	6
6: Left front	13	14	15				1	2	3	4	5	6	7	8	9	10	11	12

Figure 4.9: Sequencing diagrams for the tripod, ripple, and wave gaits

5 System Testing and Analysis

Throughout the design process, subsystems were tested and demonstrated with the milestones listed in Table 1.1. Some engineering requirements were fulfilled from component and design decisions earlier in the process, but further tests needed to be conducted to verify the requirements specific to the end product's functionality. Figure 5.1 shows the fully-built hexapod. All planned features were successfully implemented, including body motion in stationary mode and the three basic gaits chosen for crawl mode. Some additional small features were added: an LED to indicate armed status and a rotate-in-place mode.

Figure 5.1: The completed hexapod build

5.1 Timing and Calibration Tests

Reactiveness and smoothness of hexapod motion was adjusted by optimizing timing-critical sections in software. One important metric specific to stationary mode is the length of the loop in which receiver data is processed, servo angles are calculated with inverse kinemematics, and the 18 servo commands are sent to the servo controller. Table 5.1 shows the measured execution times that each of these subroutines. The receiver sends a data packet every 14 ms, which is less than 24.65 ms it takes to send all of the SSC-32U commands through UART. Therefore, accounting for this period is not necessary as the start of every loop will have a new packet ready for processing.

Table 5.1: Control loop timing for stationary mode

Subroutine	**Execution Time**
Conversion of receiver data to IK command	0.015 ms
IK calculations	1.776 ms
Send all servo commands to SSC-32U	24.650 ms
Total	26.441 ms

The time for inverse kinematic calculations was lower than expected when `math.h` trigonometric functions were used instead of predefined lookup-tables. The STM32F303 FPU helped performance with the large amount of single-precision floating point arithmetic. Since the bulk of the control loop is dedicated to sending servo commands through UART, despite using the SSC-32U's maximum supported baud rate of 115200 bits per second, any optimizations made in the other subroutines are marginal. If any significant improvement is to be made, the method of sending commands to manage the PWM signals of 18 servos must be faster, possibly by designing another servo controller board that uses a communication protocol such as SPI. Nevertheless, a total loop time of 26.441 ms results in a refresh rate of approximately 37.8 Hz in stationary mode, fulfilling the refresh rate requirement.

In crawl mode, the frequency of sent commands to the SSC-32U is directly tied to the sequencer frequency. The sequencer uses a 16-bit timer sourced at 4 MHz with a prescaler value of 92. The minimum sequencer frequency uses the full range of the timer between interrupts, or a CCR of `0xFFFF`:

$$\begin{aligned}\text{Min. sequencer frequency} &= \frac{\text{Timer clock frequency}}{(\text{CCR}+1)*(\text{Prescaler}+1)} \\ &= \frac{4000000}{(65535+1)*(92+1)} \\ &= 0.656 \text{ Hz}\end{aligned}$$

The maximum sequencer frequency uses a CCR of `0xFFFF` minus the maximum magnitude of control data multiplied by a constant scalar:

$$\begin{aligned}\text{CCR}' &= 65535 - \text{Max. magnitude} * \text{Speed scalar} \\ &= 65535 - 820 * 64 \\ &= 13055\end{aligned}$$

$$\begin{aligned}\text{Max. sequencer frequency} &= \frac{\text{Timer clock frequency}}{(\text{CCR}'+1)*(\text{Prescaler}+1)} \\ &= \frac{4000000}{(13055+1)*(92+1)} \\ &= 3.294 \text{ Hz}\end{aligned}$$

With these specific timer settings, the sequencer frequency ranges from 0.656 to 3.294 Hz. Therefore, servo commands sent in crawl mode are always sent at a much slower rate than in stationary mode. If the servo movements between leg positions are too fast at these low sequencer frequencies, then the hexapod's crawl gaits will appear to be choppy. If the servos move too slow, then commanded positions will not be fully reached before initiating movement to the next one. According to the above frequency range, legs will need to use from 1.656 to 0.303 seconds to get to their next position in the gait sequence. Fortunately, the SSC-32U supports an optional command argument to specify the speed at which servos transition from their current angle to the given position. This was used to tune the servo transition speeds to make gait motion as fluid as possible.

Crawl velocity was measured by timing the hexapod travelling over a set distance. Tripod, ripple, and wave gait crawl speeds were measured at 0.190, 0.046, and 0.029 m/s, respectively. Ripple and wave gait are naturally slower, sacrificing speed for stability. Rotation speed was measured at 24 deg/s. These movement speed metrics can vary as they are dependent on many aspects such as IK precision, tuning values for timers, parameters for stroke length, etc.

5.2 Battery Life

Battery life was tested by using the hexapod with a fully charged 6 V, 2800 mAh NiMH battery pack until the battery voltage dropped below a 5.6 V threshold, at which the receiver would disconnect occasionally most likely due to lack of operating current. Cycles lasted between approximately a 10 to 15 minute range; times varied due to the health of available batteries. There was no notable difference in battery life observed between tests in which the hexapod stood idle or constantly moved.

6 Conclusion and Future Work

The radio-controlled hexapod was completed, satisfying the defined user and engineering requirements. Despite the end product having hobbyist-level functionality that is not incredibly revolutionary to the field of legged robotics, a solid foundation was created for further development. The ease of tuning various values and adding features at the end of the project demonstrates an extensible code base and readiness for future work.

The vast possibilities of mobile robotics allow for further development in several ways. One feature that was not implemented is stacking commands used in stationary mode onto the crawl mode commands to allow for gaits with a tilted body. Legged robots can navigate complex terrain much better than those with wheels or treads, but require more advanced sensing and control systems. Challenging motions for a hexapod such as climbing stairs, rappelling down steep faces, or swinging across monkey bars seem possible by adding sensors and having a sufficient understanding of physics.

7 Reflection

After learning about embedded systems design with the MSP432, I wanted to familiarize myself with the STM32, another popular ARM MCU. This was a good lesson in using my available resources (prior knowledge, an advisor, online forums, several datasheets and reference manuals) to learn about an unfamiliar board and use it in a product.

Knowing that the bulk of this project would be time spent writing and debugging embedded software, I chose the readily available SSC-32U to manage the 18 servo PWM signals. The SSC-32U did its job, but I would design and use my own servo controller board given more time.

In terms of project management, I gained lots of experience using tools like Trello to organize task lists, datasheets, research, and documents effectively. I very much appreciated having Dr. Hummel as my advisor to discuss design approaches, concerns, and logistics.

Bibliography

[1] Bruce A. Aikenhead, Robert G. Daniell, and Frederick M. Davis. Canadarm and the space shuttle. *Journal of Vacuum Science & Technology A*, 1(2):126–132, 1983.

[2] DG Caldwell and N Tsagarakis. Biomimetic actuators in prosthetic and rehabilitation applications. *Technology and Health Care*, 10(2):107–120, 2002.

[3] Erico Guizzo. Boston dynamics' spot robot dog goes on sale, Sep 2019.

Appendix A: Bill of Materials

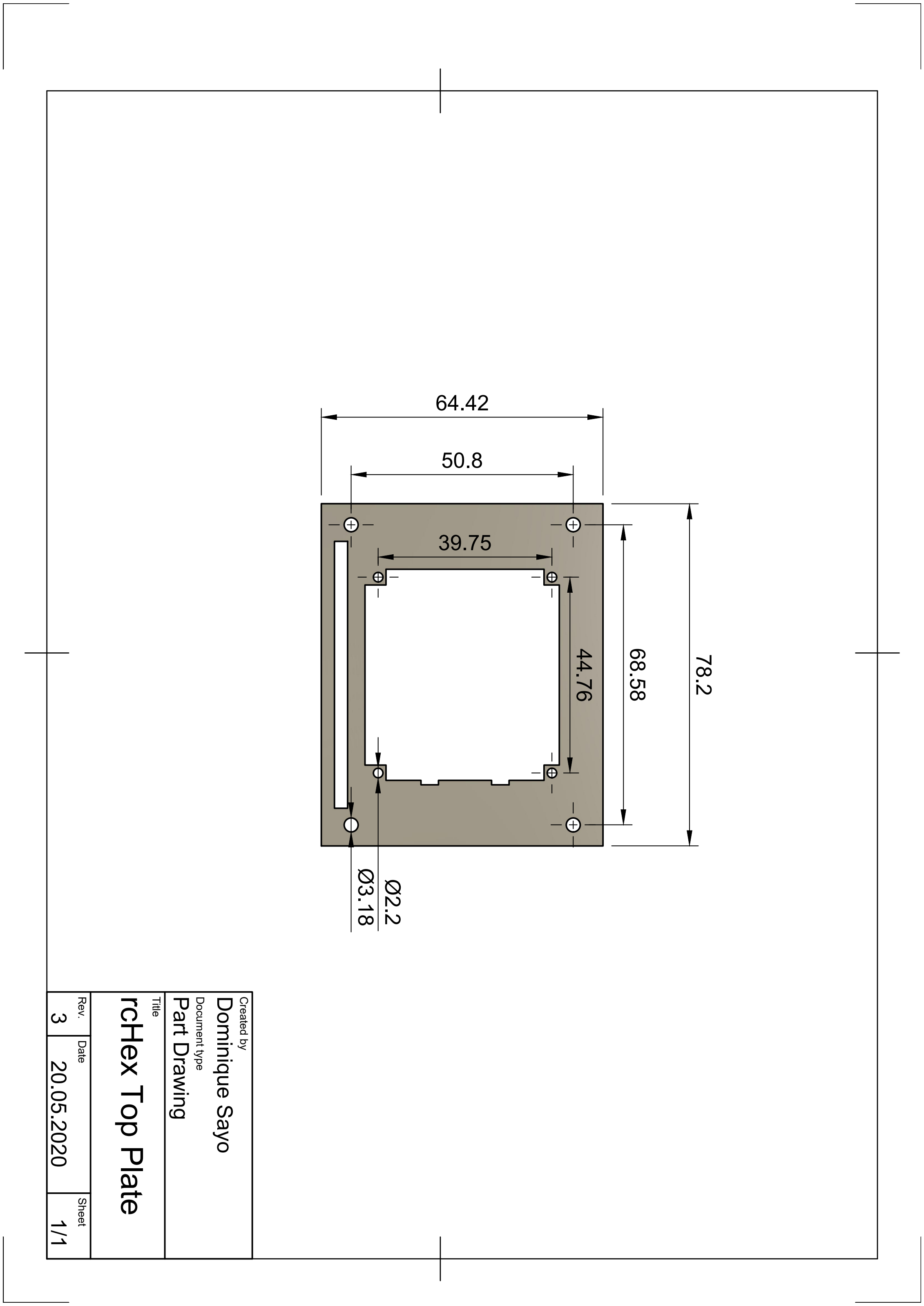
Revised: 2 Jun 2020

Revision: 4

rcHex Bill Of Materials Page 1 of 1

Item	Quantity	Unit	Part No.	Description	Manufacturer	Cost per unit	Extended Cost
1	1	ea.	Q X7	Taranis 2.4G Transmitter	FrSky	$107.99	$107.99
2	1	ea.	XM+	SBUS Micro Receiver	FrSky	$13.99	$13.99
3	1	ea.	STM32F303K8	Nucleo Dev Board	STMicro	$10.99	$10.99
4	1	ea.	SSC-32U	Servo Controller	LynxMotion	$44.95	$44.95
5	18	ea.	HS-645MG	High Torque Metal Gear Servo	Hitec	$28.99	$521.82
6	1	ea.	Phoenix	3DOF Hexapod Frame	LynxMotion	$248.90	$248.90
7	2	ea.	n/a	6V 2800 mAh NiMH Battery Pack	LynxMotion	$26.95	$53.90
8	1	ea.	IMAX B6	Balance Charger	SkyRC	$30.99	$30.99
9	1	ea.	UBEC3A2-6S	5V 3A UBEC	Hobbywing	$7.06	$7.06
10	1	ea.	n/a	1/8" PET Braided Sleeving	Alex Tech	$10.50	$10.50
11	2	ea.	n/a	XT60 Connector Pair	AMASS	$1.50	$3.00
12	1	ea.	n/a	Perfboard	ElectroCookie	$0.99	$0.99
13	1	ea.	LTL-1CHG	Green LED	Lite-On Inc.	$0.32	$0.32
14	1	ea.	CFM12JT47R0	RESISTOR 47 OHM 1/2W 5%	Stackpole Electronics Inc.	$0.10	$0.10
15	0.006	kg	n/a	Gun Metal Gray PETG Filament	Atomic USA	$32.99	$0.20
						Total	***$1,055.70***

Appendix B: Top Plate Drawing



Appendix C: Code Listing

Only a selection of code is shown here due to length. See https://github.com/dsayo/rcHex for the complete repository.

main.h

```c
/******************************************************************************
 * main.h
 *
 * RC Hexapod
 *
 * California Polytechnic State University, San Luis Obispo
 * Dominique Sayo
 * 19 May 2020
 ******************************************************************************
 */
#ifndef __MAIN_H
#define __MAIN_H

#include "stm32f3xx_hal.h"

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);
void Error_Handler(void);

#endif /* __MAIN_H */
```

main.c

```c
/*****************************************************************************
 * main.c
 *
 * RC Hexapod
 *
 * California Polytechnic State University, San Luis Obispo
 * Dominique Sayo
 * 19 May 2020
 *****************************************************************************
 */
#include <string.h>
#include <math.h>
#include "main.h"
#include "sbus.h"
#include "ssc.h"
#include "term.h"
#include "controls.h"
#include "ik.h"

TIM_HandleTypeDef htim3;

UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart1_rx;

volatile uint8_t ready = 0;       /* Flag: ready to process RX data    */
volatile uint8_t delta = 0;       /* Flag: change in RX data           */
volatile uint8_t phase_ready = 0; /* Flag: ready for next crawl phase  */
uint16_t seq_speed;               /* CCR subtractor for sequence speed */
Phase max_phase;                  /* Maximum phase in sequence cycle   */
float angle_delta[NUM_LEGS][NUM_SERVO_PER_LEG]; /* Servo degree changes */

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
```

```c
static void MX_DMA_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM3_Init(void);

int main(void)
{
   uint8_t packet[PACKET_SZ]; /* SBUS packet data            */
   RXData rx_data;            /* Formatted ctrl data         */
   RXData old_rx_data;        /* Previous ctrl data          */
   Command cmd;               /* Stationary command          */
   uint8_t armed = 0;         /* Flag: is armed              */
   Mode mode = MODE_RPY;      /* Movement mode               */
   CrawlMode cmod = TRIPOD;   /* Gait type / rotate          */
   Phase phase = A1;          /* Current phase in sequence   */
   float crawl_angle;         /* Crawl direction in radians  */
   uint16_t rot_dir;          /* Rotation direction +CW -CCW */

   /* Reset peripherals, Initializes the Flash interface and the Systick. */
   HAL_Init();

   /* Configure the system clock */
   SystemClock_Config();

   /* Initialize all configured peripherals */
   MX_GPIO_Init();
   MX_DMA_Init();
   MX_USART1_UART_Init();
   MX_USART2_UART_Init();
   MX_TIM3_Init();

   HAL_Delay(1000); /* One second startup */

   /* Start reading incoming RX data over UART */
   HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_1);
```

```
    __HAL_UART_FLUSH_DRREGISTER(&huart1);
    HAL_UART_Receive_DMA(&huart1, packet, PACKET_SZ);

    powerup_stance();    /* Fast stance on powerup       */
    neutral_stance();    /* Transition to neutral stance */

    while (1)
    {
       /* UART Error checking */
       if (HAL_UART_GetError(&huart1))
       {
          /* Overrun error, flush and restart */
          huart1.ErrorCode = HAL_UART_ERROR_NONE;
          __HAL_UART_FLUSH_DRREGISTER(&huart1);
          HAL_UART_Receive_DMA(&huart1, packet, PACKET_SZ);
       }

       /* Parse control data when ready */
       if (ready)
       {
          ready = 0;

          /* Save prev rx data and get new rx data*/
          memcpy(&old_rx_data, &rx_data, sizeof(RXData));
          sbus_format(packet, &rx_data);

          /* Get armed switch and operating mode */
          armed = get_arm(rx_data);
          mode = get_mode(rx_data);

          if (armed)
          {
             switch (mode)
             {
                case MODE_CRAWL:
```

```
            /* Parse control data into gait and sequencer info */
            cmod = get_cmod(rx_data);
            seq_speed = get_speed(rx_data, cmod);
            crawl_angle = get_angle(rx_data);
            rot_dir = get_rot_dir(rx_data);
            break;

         default: /* Stationary modes */
            /* Check deltas (if rx data changed) */
            delta = ctrl_delta(&old_rx_data, &rx_data);
            seq_speed = 0; /* Don't use sequencer */
            break;
      }
   }
}

/* Enable control if armed */
if (armed)
{
   switch (mode)
   {
      case MODE_CRAWL:
         /* When sequencer signals next phase */
         if (phase_ready)
         {
            phase_ready = 0;

            /* Execute phase movement */
            exec_phase(phase, cmod, seq_speed, crawl_angle, rot_dir);
            phase++;
            if (phase > max_phase)
            {
               phase = A1;
            }
         }
```

```
                break;

            default:
                /* Stationary mode */
                if (delta)
                {
                    /* Only run new calculations if rx data changed enough */
                    delta = 0;

                    /* Convert rx data to command & calculate inv. kinematics */
                    cmd = to_command(rx_data, mode);
                    ik(cmd, ALL_LEGS, angle_delta);
                    set_angles(ALL_LEGS, angle_delta, STATIONARY_SERVO_SPEED);
                    ssc_cmd_cr();   /* Send new servo pwm */
                }
                break;
            }
        }
        else
        {
            /* Stand still */
            neutral_stance();
        }

    }
}

/* System Clock Configuration
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
    RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

```

```c
  /* Initializes the CPU, AHB and APB busses clocks */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /* Initializes the CPU, AHB and APB busses clocks */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
    |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART1;
  PeriphClkInit.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK1;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    Error_Handler();
  }
}

/* TIM3 Initialization Function
 */
static void MX_TIM3_Init(void)
{
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
```

```
TIM_OC_InitTypeDef sConfigOC = {0};

htim3.Instance = TIM3;
htim3.Init.Prescaler = 92;
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = 0xFFFF;
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
{
   Error_Handler();
}
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
{
   Error_Handler();
}
if (HAL_TIM_OC_Init(&htim3) != HAL_OK)
{
   Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
{
   Error_Handler();
}
sConfigOC.OCMode = TIM_OCMODE_ACTIVE;
sConfigOC.Pulse = 0;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
{
   Error_Handler();
}
```

```c

    HAL_TIM_MspPostInit(&htim3);
}

/* USART1 Initialization Function
 */
static void MX_USART1_UART_Init(void)
{
    huart1.Instance = USART1;
    huart1.Init.BaudRate = 100000;
    huart1.Init.WordLength = UART_WORDLENGTH_9B;
    huart1.Init.StopBits = UART_STOPBITS_2;
    huart1.Init.Parity = UART_PARITY_EVEN;
    huart1.Init.Mode = UART_MODE_RX;
    huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart1.Init.OverSampling = UART_OVERSAMPLING_16;
    huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_RXINVERT_INIT |
        UART_ADVFEATURE_DMADISABLEONERROR_INIT;
    huart1.AdvancedInit.RxPinLevelInvert = UART_ADVFEATURE_RXINV_ENABLE;
    huart1.AdvancedInit.DMADisableonRxError =
        UART_ADVFEATURE_DMA_DISABLEONRXERROR;
    if (HAL_UART_Init(&huart1) != HAL_OK)
    {
        Error_Handler();
    }
}

/* USART2 Initialization Function
 */
static void MX_USART2_UART_Init(void)
{
    huart2.Instance = USART2;
    huart2.Init.BaudRate = 115200;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
```

```c
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
}

/* Enable DMA controller clock
 */
static void MX_DMA_Init(void)
{
  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel5_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel5_IRQn);

}

/* GPIO Initialization Function
 */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
```

```
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /* Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOF, GPIO_PIN_0 | GPIO_PIN_1, GPIO_PIN_RESET);

    /* Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4 | GPIO_PIN_5, GPIO_PIN_RESET);

    /* Configure GPIO pins : PF0 PF1 */
    GPIO_InitStruct.Pin = GPIO_PIN_0 | GPIO_PIN_1;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);

    /* Configure GPIO pins : PB4 PB5 */
    GPIO_InitStruct.Pin = GPIO_PIN_4 | GPIO_PIN_5;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}

/* Callback for complete UART receive
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART1)
    {
        ready = 1;
    }
}
```