

Arizona State University
Barrett, the Honors College
Wireless Mecanum Wheel Vehicle
Severin J. Davis

Approved:

Kevin Burger, Director

Greg Vannoni, Second Committee Member

Accepted:

Dean, Barrett, the Honors College

Contents

| | |
|------------------------------|----|
| Abstract..... | 2 |
| Project Goals | 3 |
| Components | 4 |
| Hardware | 4 |
| Software | 5 |
| Vehicle Diagram | 7 |
| Vehicle Diagram Key | 8 |
| Block Diagram..... | 9 |
| Tools..... | 10 |
| Hardware | 10 |
| Software | 10 |
| Design | 12 |
| Hardware | 12 |
| Software | 14 |
| Challenges and Changes | 20 |
| Conclusion | 22 |
| References..... | 23 |
| Appendix | 24 |

Abstract

The purpose of this project was to construct and write code for a vehicle to take advantage of the benefits of combining stepper motors with mecanum wheels. This process involved building the physical vehicle, designing a custom PCB for the vehicle, writing code for the onboard microprocessor, and implementing motor control algorithms.

Project Goals

The project goals can be roughly divided into the following components:

- Design and construct a physical vehicle that takes advantage of mecanum wheel functionality
- Create the schematic and layout for a custom PCB and have this manufactured
- Communicate with PCB chips from Kinetis Freedom Board
- Use Kinetis hardware modules to efficiently control the vehicle
- Communicate with the vehicle wirelessly
- Control the vehicle remotely with input on a controller
- Document the process
- Publish the code as a starting point for CSE325 at ASU

Components

Hardware

Control

Xbox 360 Controller

A wireless Xbox 360 controller was chosen to provide the user with an easy-to-use input device to control the vehicle. This particular controller was chosen because of the easy programming interface and its ergonomic design. Additionally, the controller was already owned, so no additional funds were spent on it.

Vehicle

Freescale FRDM-KL46Z

Freescale's FRDM-KL46Z is a small Arduino-style prototype and evaluation board that incorporates an ARM Cortex-M0+ microcontroller. I/O pins exit the board with standard Arduino pin spacing. This board was chosen due to its low cost (~\$20), the large number of available pins (64), and its speed (48 MHz). The board runs at and outputs 3.3V, allowing it to power several other external components. In this project the KL46Z provided 3.3V to 8 shift registers and 4 stepper motor driver chips.

ON Semiconductor LV8727 Stepping Motor Driver

The LV8727 chip was chosen due the fact that free samples were available from ON Semiconductor. This chip allowed the stepper motors to be driven in a fairly easy, automated fashion by simply providing a periodic step direction inputs.

Digi XBee

XBee modules by Digi were chosen to provide wireless remote control communication between the host PC and the vehicle. The module on the transmitter side attaches to the host PC via a USB adapter, and appears in Windows as a serial port. Thus, other than communicating with a standard Windows serial port, no setup needs to be performed. The module on the receiver side plugs into the custom PCB created for this project and connects to the UART module located in the KL46 microcontroller.

Software

Control

XInput API

Microsoft's XInput API is used to poll the Xbox controller's state. The Xbox controller is polled and the state of all the buttons and the positions of the two triggers and analog sticks is read and interpreted.

Vehicle

Universal Asynchronous Receiver/Transmitter (UART)

The KL46 contains 3 UART modules. For this project UART0 is used to interface with the XBee module. Since the communication between the control PC and the vehicle is relatively simple, and congestion was not an expected issue, flow-control lines were omitted, and only RX and TX lines are connected. Once the XBee receives a byte of data, the data is forwarded to the microcontroller, and an interrupt is generated via the KL46's UART module. This interrupt retrieves and stores the received data for processing.

Serial Peripheral Interface (SPI)

Both the LEDs and the stepper motor driver chips (LV8727) used in this project are interfaced to SPI by attaching shift registers to their input pins. Since the LEDs have three control pins (RGB) each and the motor chips have even more, using SPI to communicate with these peripherals becomes a necessity as it reduces the number of signals on the board. This simplifies the programming and the routing portion of this project significantly. Time sensitive signals and analog signal, such as the direction and step input to the motors and the current limiting analog value are fed directly by the microcontroller.

Timer/PWM Module (TPM)

The TPM on the KL46 microcontroller is a major component of this project as it provides a relatively convenient and precise way to generate pulses to drive the vehicle's stepper motors at varying speeds. Rather than using this module as a typical PWM generator, the spacing between pulses varies and is manually set and computed by an interrupt service routine.

Miscellaneous Modules

The KL46 processor and the FRDM-KL46Z contain many other modules that are used for this project.

Programmable Interrupt Timers (PIT) are used to provide a periodic check on whether UART messages are still being received. If the vehicle loses contact with the host PC, the PIT service routine initiates a vehicle shutdown process.

A **Digital to Analog Converter (DAC)** is used to generate the analog signal that provides the current limiting setting to the motor drivers.

The FRDM-KL46Z contains two **LEDs** (red and green) that haven proven to be extremely useful for fast, initial debugging.

Additionally, the FRDM-KL46Z includes two **switches** that were used in testing, debugging, and in the final product.

Vehicle Diagram

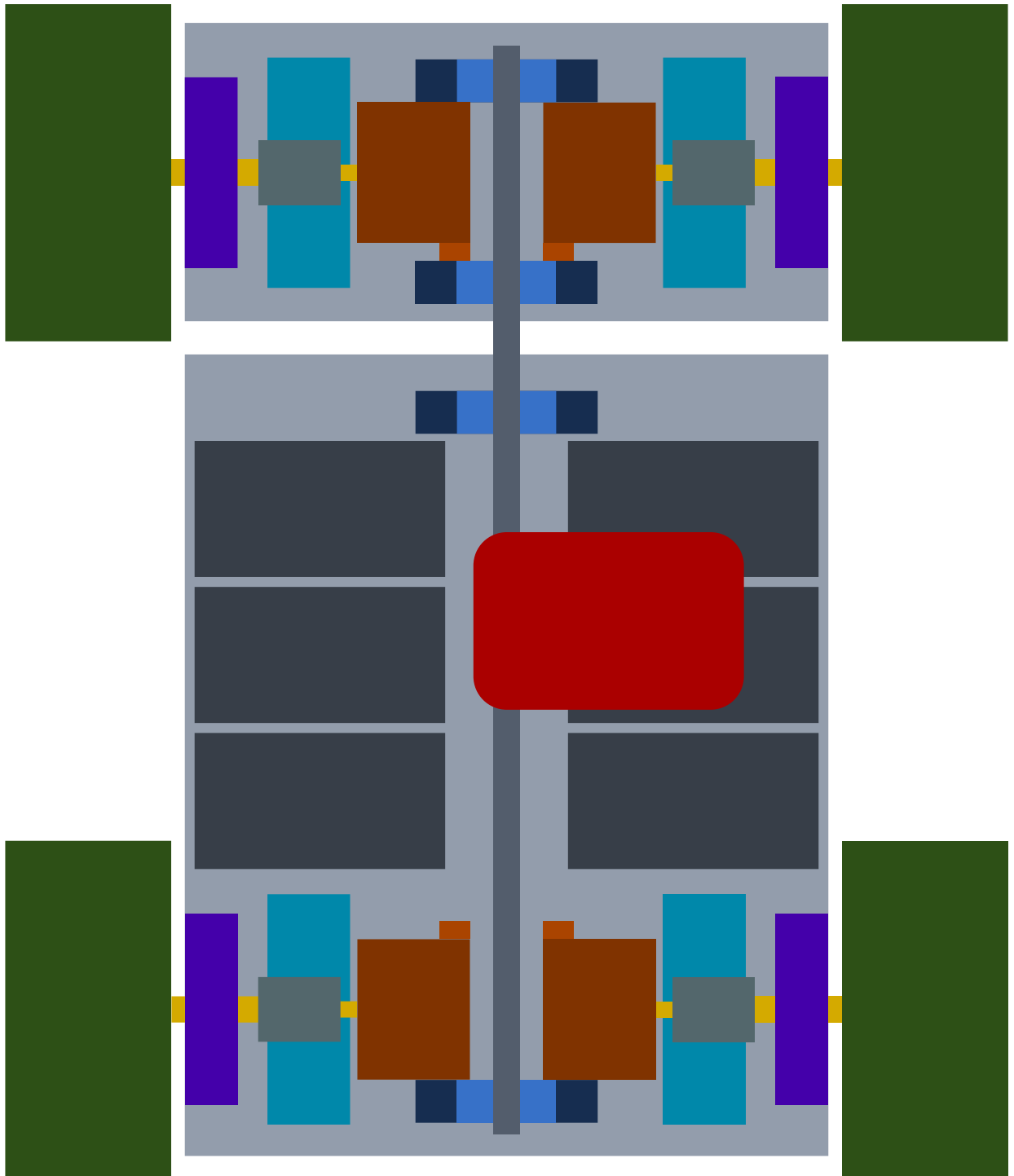


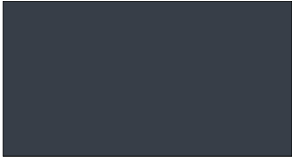





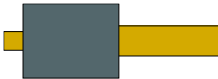


Figure 1: Vehicle Diagram

Vehicle Diagram Key

| | |
|---|-------------------------|
|  | Mecanum Wheel |
|  | Vehicle Spine |
|  | Battery Pack |
|  | Aluminum Motor Bracket |
|  | Stepper Motor |
|  | Shaft Bearing |
|  | Spine Bearing |
|  | Freedom Board |
|  | Motor Shaft and Coupler |

Block Diagram

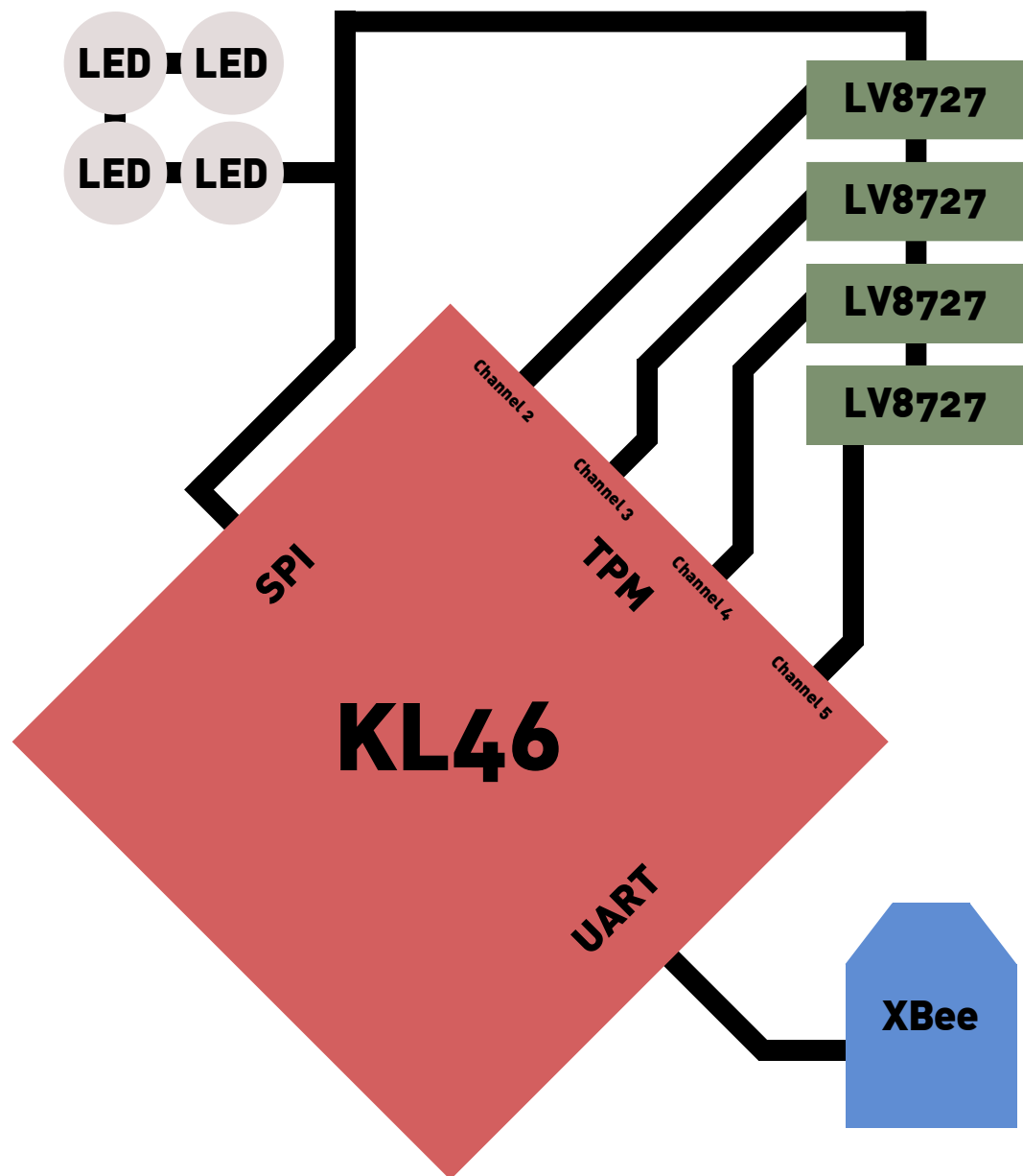


Figure 2: Block Diagram

Tools

Hardware

Multimeter

A low cost multimeter was frequently used simply to verify battery charge levels via voltage readings, ensure connectivity via impedance readings, and to verify correct operation of the microcontroller and shift register outputs early in the project.

Saleae 8 Channel Logic Analyzer

The Saleae 8 Channel Logic Analyzer proved invaluable for debugging signal issues on the microcontroller pins. While the Multimeter could be used to verify proper behavior of signals that were held at a constant value, the logic analyzer was necessary to verify correct timing of signals at the microsecond level.

Software

Freescale CodeWarrior

CodeWarrior 10.6 was used to develop the C code for the KL46 microcontroller. This is an Eclipse-based editor that provides an integrated debugger. This enables the user to step through the embedded code line by line to isolate and correct issues.

Microsoft Visual Studio Express

Visual Studio was used to develop the C++ code for the Host PC. Since interfacing with an Xbox controller was required for this project, the Microsoft XInput API was necessary. Visual Studio was chosen because of its ease of use, debugging capabilities, and it's extensive and relatively seamless integration in the Windows environment, and by extension, the Xbox controller interfacing.

Open-Source KiCad

KiCad is an open source PCB design suite that was used to create the schematics, component footprints, and PCB files necessary to order a custom PCB for this project. KiCad streamlines the PCB design process by linking the various steps in the PCB workflow into a self-contained, intuitive software package.

Open-Source Inkscape

Inkscape is an open source graphic design tool that was used to draw diagrams to determine correct spacing and fit of physical components of the vehicle. Inkscape can generate shapes with precise dimensions on Cartesian coordinates, making it ideal for physical layout purposes. Additionally, it was used to generate block diagrams for this report.

Design

Hardware

Vehicle

As shown in *Figure 1*, the vehicle consists of two Plexiglas platforms, each supporting a motor pair. Each motor's axle extends through a pillow-block bearing so that the sideways force applied on the wheel axle is mitigated and does not directly affect the motor.

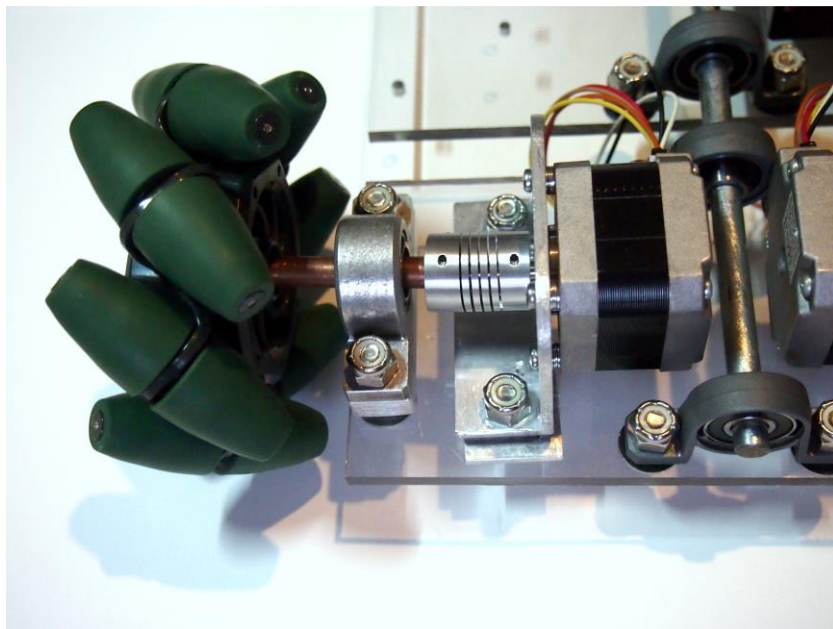


Figure 3: Shaft Assembly

The two platforms are connected via an 8mm metal rod that runs through a total of 4 pillow-block bearings at the center of the vehicle. This ensures that the front wheel pair can rotate independently of the rear pair. This configuration allows the vehicle to keep all four wheels in contact with the ground at all times, adjusting automatically for small bumps in its path.

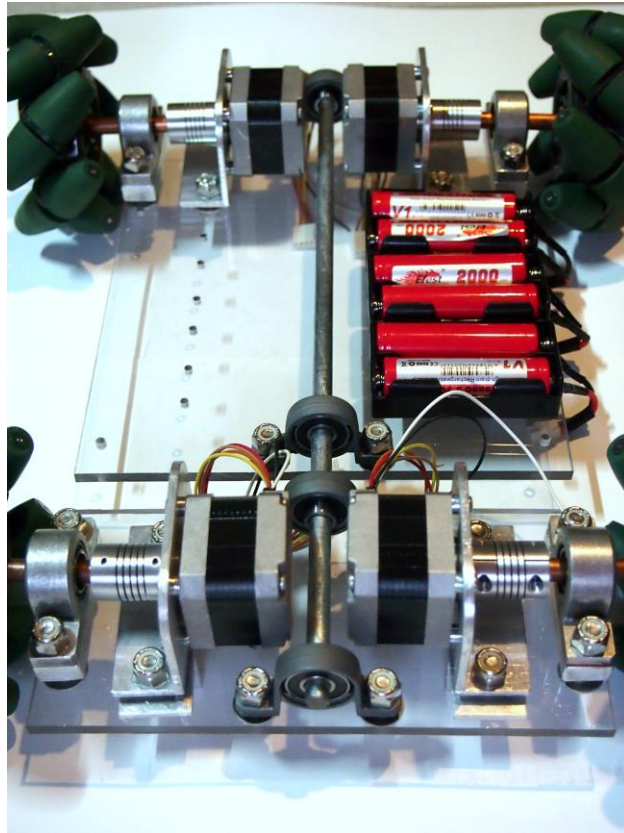


Figure 4: Vehicle Spine

Mecanum Wheels

Mecanum wheels allow translational and rotational motion if driven individually. Obviously, if all four wheels turn in one direction, the vehicle moves forward or backwards. However if a pair of motors spins in opposite direction, the vehicle moves sideways. Additionally, if the front and rear wheel pairs spin to move the front of the car sideways and the back of the car in the other direction, the car gains rotational motion.

Stepper Motors

Unlike DC motors which spin faster the more voltage is applied, stepper motors can step a precise angle and will hold that position. As such, counting the number of steps applied to a motor will guarantee a corresponding change in angle. By extension, this can be used to guarantee motor velocity. This is important for this project as all motors must spin at accurate velocities to result in correct motion and control of the vehicle. Microstepping of these motors allows an increased number of steps to occur between the physical steps in the motor. The stepper motors chosen for this project run off the ~8V supply and pull 1.2 A per coil. Since each motor has two coils, the minimum current

draw for one motor is when only one coil is activate (1.2 A), and the maximum current draw is when the motor is held directly between two physical steps ($0.707 \cdot 1.2 \cdot 2 \text{ coils} = 1.69 \text{ A}$).

As such, the combined current for the motors is 6.78 A maximum and 4.8 A minimum.

PCB

The custom PCB provides the communication channels between the various required chips for this project. The microcontroller board, the xbee, the stepper motor chips, and the transistors for the LEDs are the main components. The stepper motor chips require a number of additional components such as resistors and capacitors on some of their inputs that physically set certain behaviors of the stepper motor chips.

Due to the high number of pins on the stepper motor chips and the signals required to drive the LEDs directly, shift registers are used to interface these peripherals via the SPI protocol. With the exception of the analog current limiting input, the step signals, and the direction signals for each, the remaining required inputs on the motor chips are fed with four 8bit shift registers. The same was done for the LEDs: two 8bit shift registers feed six dual Nchannel transistors to control all twelve RGB LED signals.

Software

The vehicle software is built as modules that are integrated together to provide the needed functionality. Additionally, with one or two exceptions, the code is entirely interrupt driven.

The TPM module is used to step the motors. The TPM hardware module consists of a counter and a value. When the counter reaches the value, a pulse and an interrupt are automatically generated. The pulse is used to automatically step the motor while the interrupt is used to compute when the next step should occur. As such, the code that is called when an interrupt occurs compares the current direction of the motor and the current period of the motor to either maintain, accelerate, or decelerate to meet the target direction and period.

Since the expression for calculating the exact new period involved a lengthy square root operation, a look up table was created and from this a linear approximation was generated. These look up tables are shown in *Curve and Approximation*. On these

graphs, the X- axis shows the old period and the Y-axis shows the new period. If a vehicle has a current period of 60000, which is nearly stopped, and the target period is updated to 40, an interrupt will insert 60000 as the old period into the lookup table, and the table will return about 1400 as the next period for reasonably smooth acceleration. This process of looking up values continues until the target velocity is overshoot. The same process, but in reverse occurs for the deceleration table. The resulting pulse and direction waveforms from these continues lookups for various motions are shown in *Figure 5, Figure 6, and Figure 7.*



Figure 5: Translational Motion Pulses

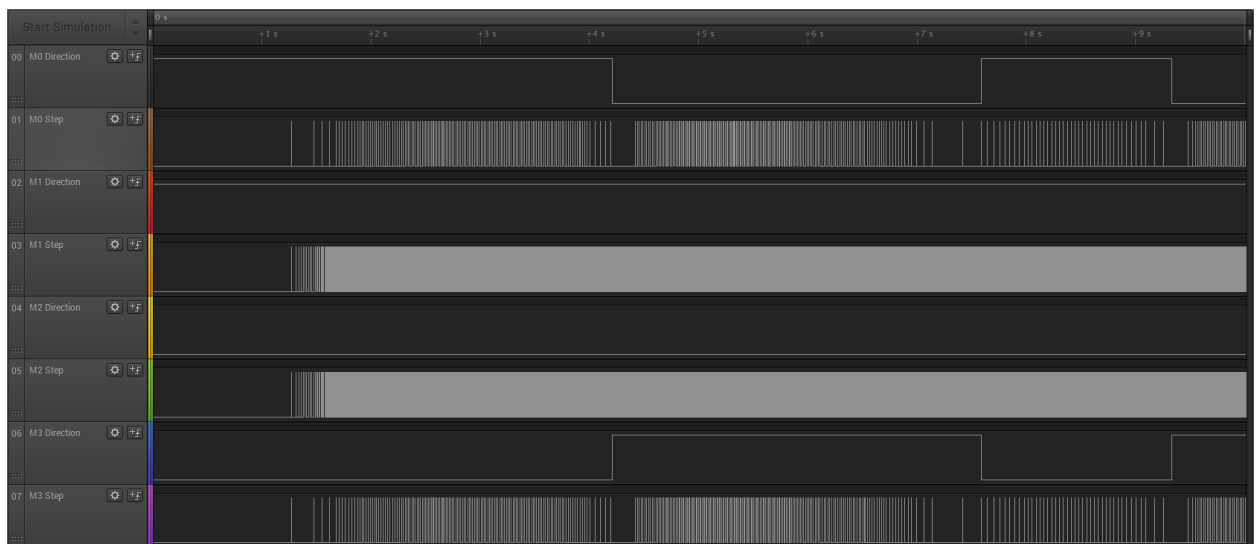


Figure 6: Diagonal Motion Pulses

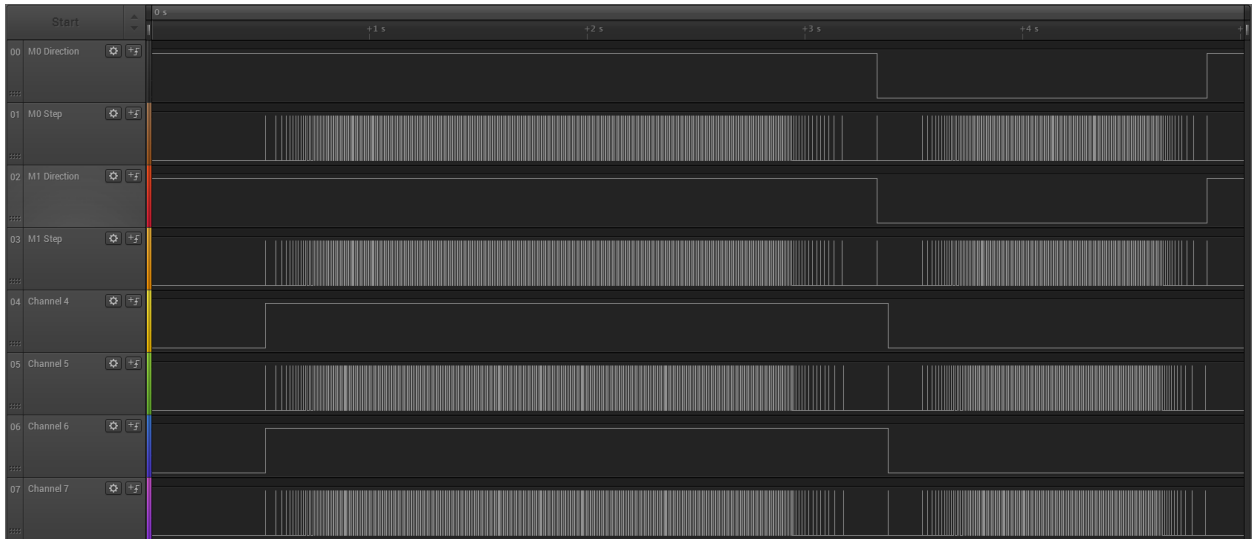


Figure 7: Rotational Motion Pulses

The XBee interacts with the UART module, providing wireless communication to the vehicle. This is also purely interrupt driven. Once the final 8bit packet of an entire message is received, the target direction, and target period is updated for each motor. Because the target direction and target period are separate variables and the update must occur as an atomic operation, interrupts from the TPM module are briefly disabled to allow the update to run atomically.

To account for XBee packet loss, due to a temporary power loss issue or interference, it became very obvious that some kind of loss detection and synchronization scheme was needed for the wireless communication. Initially the assumption was made that the vehicle would always remain in range of the transmitter module, and as a result, all packets would arrive successfully. However, interference or temporary power loss result in a single dropped packet, and, due to the lack of synchronization, all subsequent messages are offset by 1, resulting in incorrect translation of message contents, and, by extension, incorrect and extreme vehicle motion.

It was determined that two packets with 0x00 contents should never occur directly after each other in a valid message payload. To ensure this, the four unused bits in the direction byte were filled with 0xA, to force a non 0x00 direction packet. Naturally, the motors cannot physically have a 0 period. Thus, two 0x00 packets in a row were chosen as the synchronization indicator. The vehicle simply has to check if two packets like this arrived in a row, and this indicates the beginning of a message.

This scheme simply provides for synchronization but does not provide detection of a lost packet within the presumably valid packets following the 2 0x0 packet synchronization indicator. To solve this, an additional byte was added to the end of the message. The contents of this byte are simply an XOR of all the bytes in the preceding message. After all the bytes have been received on the vehicle side, the vehicle recomputes an XOR of all the message bytes and compares it to the last received byte. If it matches, there is a very high chance that no packets were lost in the transfer and the XBee interrupt can commit and update the target periods and directions for the motors.

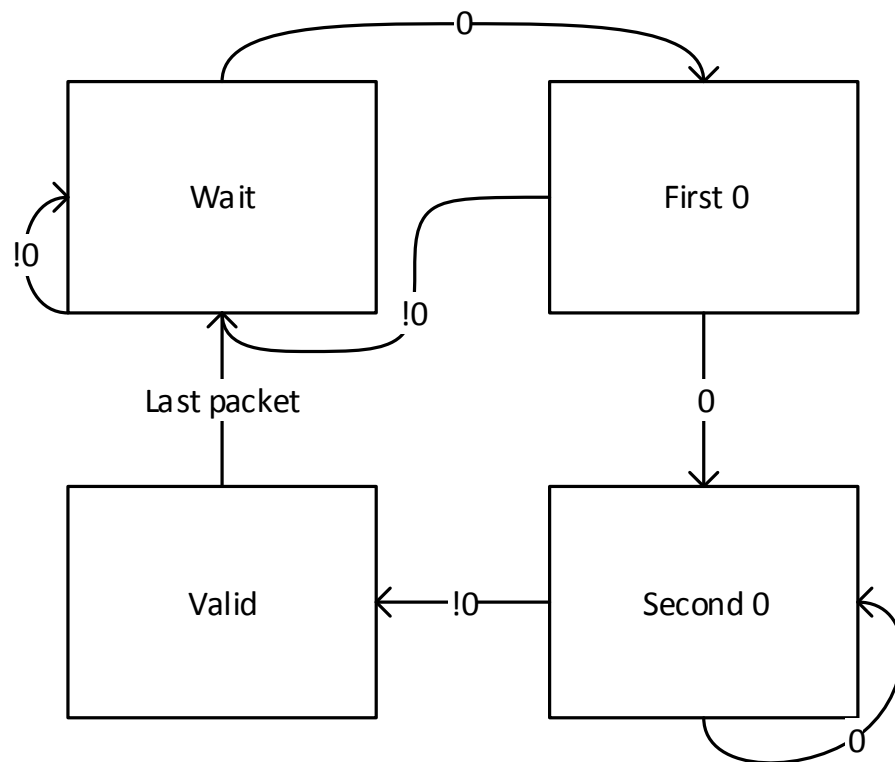


Figure 8: XBee State Diagram

XBee Transfer

0. Synchronization Byte 1

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|-----|---|---|---|---|---|---|---|
| Contents | 0x0 | | | | | | | |

1. Synchronization Byte 2

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|-----|---|---|---|---|---|---|---|
| Contents | 0x0 | | | | | | | |

2. Direction Byte

| | | | | | | | | |
|-----|-----|---|---|---|-------|-------|-------|-------|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | 0xA | | | | Dir 3 | Dir 2 | Dir 1 | Dir 0 |

3. Motor 0 Lower Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Lower 8 bits of motor 0 period | | | | | | | | |

4. Motor 0 Upper Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Upper 8 bits of motor 0 period | | | | | | | | |

5. Motor 1 Lower Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Lower 8 bits of motor 1 period | | | | | | | | |

6. Motor 1 Upper Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Upper 8 bits of motor 1 period | | | | | | | | |

7. Motor 2 Lower Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Lower 8 bits of motor 2 period | | | | | | | | |

8. Motor 2 Upper Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Upper 8 bits of motor 2 period | | | | | | | | |

9. Motor 3 Lower Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Lower 8 bits of motor 3 period | | | | | | | | |

10. Motor 3 Upper Byte

| | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Upper 8 bits of motor 3 period | | | | | | | | |

11. XOR Byte

| | | | | | | | | |
|-------------------------------|---|---|---|---|---|---|---|---|
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Result of XOR of Bytes 2 - 10 | | | | | | | | |

The fact that the TPM module recomputes on each pulse, means that periods and directions may not react to the XBee updates fast enough. This is particularly an issue if the motor is at 0 velocity, meaning no pulses or interrupts occur whatsoever. A similar scenario occurs whenever the target period or direction is updated. Because the new pulse time is only calculated on each pulse of the TPM, the TPM channel may react much too slow.

Below is an example that illustrates this issue:

Current Period = 1 second and Target Period = 1 second

The motor pulses at $t = 0$ and computes its next pulse to occur at $t = 1$, which matches its Target Period. Shortly after $t = 0$, the UART interrupt updates the Target Period to 0.5 seconds. Ideally, the motor would recompute the next pulse and update for the next pulse to occur at sooner than the scheduled pulse at 1 second, so that the motor immediately reacts to the new, smaller period and accelerates.

However, the motor will only realize the update has occurred on the next pulse at $t = 1$ second, putting it behind, and delaying the acceleration.

This requires some kind of wake-up functionality, allowing the motor to update earlier than normally scheduled.

This was accomplished by utilizing an extra "update" TPM channel.

When the UART receives its last packet, it updates all target directions and target periods, sets "update flags," and sets the update TPM channel to interrupt as soon as possible.

This allows the other channels a chance to pulse and update in the case that their pulses

happen in between the update and the update channel pulse. In this event, that interrupting channel will clear its update flag, indicating the update event was caught before the update channel had a chance to perform the update.

When the update channel interrupts, it updates all four channels based on whether the update flag for that channel is still set. If it is set, indicating an update on that channel needs to occur, it checks to make sure there is enough time to perform the update on a specific channel. If the update channel interrupt occurs at $t = 0$, and the pulse of channel 1 is scheduled for $t = 1, 2$, or 3 , there is not enough time to perform the update, the flag is cleared, and the update and period recomputation is left to the regularly scheduled pulse.

If there is enough time, the update channel will recompute the period based on the previous period and direction of the relevant TPM channel, and the new, updated target period and direction for that channel. If the new period is greater than the old period, the pulse is automatically rescheduled to occur at the correct later time. If the new period is less than the old period, the update channel verifies that the TPM counter has not passed the time where the new pulse should have occurred. If the counter has not passed that point, the new pulse is scheduled. If the counter has passed that point, the pulse the counter the update occurred too late and the pulse is scheduled to occur as soon as possible.

The PIT module is used to implement a kind of deadman switch. This simply involves a flag that is set every time a message from is received on the UART. The PIT periodically generates interrupts at a lower rate than the UART is receiving messages that clear the flag. This means that, if the PIT interrupt ever runs and the flag has not been cleared, no UART messages have been received between the current PIT interrupt and the last. This indicates a communication failure and the vehicle initiates a shutdown sequence.

Challenges and Changes

Due to the lack of knowledge regarding stepper motor behavior starting this project, it was incorrectly assumed that the stepper motor could be driven with the driver chips set at their half step configuration. This would have meant that for every pulse, the motor would move half of a stepper motor step. Since the stepper motor contains 200 physical

steps, a half step per pulse would have resulted in a full rotation occurring every 400 pulses. Unfortunately stepper motors, unlike DC motors, lose torque at high velocities. As such, the original idea to use half steps with a certain velocity was not realistic as the wheels refused to spin up due to lack of torque.

Solutions to this would have been to simply decrease the maximum speed, increase the maximum allowed current on the motor driver chips, or, perhaps, to modify the acceleration curve to be less steep. The motor driver chips however, also allow for increased stepping resolution. As such they can range from half steps, as was originally planned, to as small as 1/128 of a step. Increasing the resolution of the step in this way has a combined effect of decreasing the maximum wheel velocity and scales down the acceleration curve. This could have been achieved simply by the aforementioned possible fix, implemented in code. However, implementing this fix via an increased resolution resulted in smoother wheel spinning and subsequently car movement, because each pulse only advanced the wheel at a fraction of the angle resulting from the previous half step setting.

Known Issues

If less than five battery packs are connected, the current draw can apparently become high enough to either disable the XBee or kill the received message. Obviously the remedy for this is to ensure that more than four battery packs are connected and the batteries are properly charged.

Another issue, is that periodically, particularly when a motor is stepping very slowly, the motor will freeze for about 2.1 seconds. This indicates that the new pulse value was written after the counter, and the counter must count and overflow to reach that value once again. The bug likely exists somewhere in the extra “update” TPM channel code, but as of this writing, it has not been isolated.

Conclusion

Overall the project was a success. The custom PCB worked flawlessly and I was able to learn the PCB design process and workflow in the process. The programming portion of this project was also successful as I was able to write large amount of code that, for the most part, worked as it should. In total the code amounted to about 1700 lines. This code, particularly the code that enables and controls the Kinetis hardware modules, is fairly modular, and can be easily reused for other projects or for CSE325 at ASU. The hardware modules within the processor were each assigned to control a specific part of the vehicle in a way that would offload work, free up cycles, and make the vehicle more efficient. Throughout this process I also learned quite a bit about stepper motors, their behavior, and most relevantly, their limitations. The major challenges in this project revolved around these motor limitations and the weight of the vehicle. As with most technology issues, these could likely be relatively easily solved with higher financial, time, or power investment.

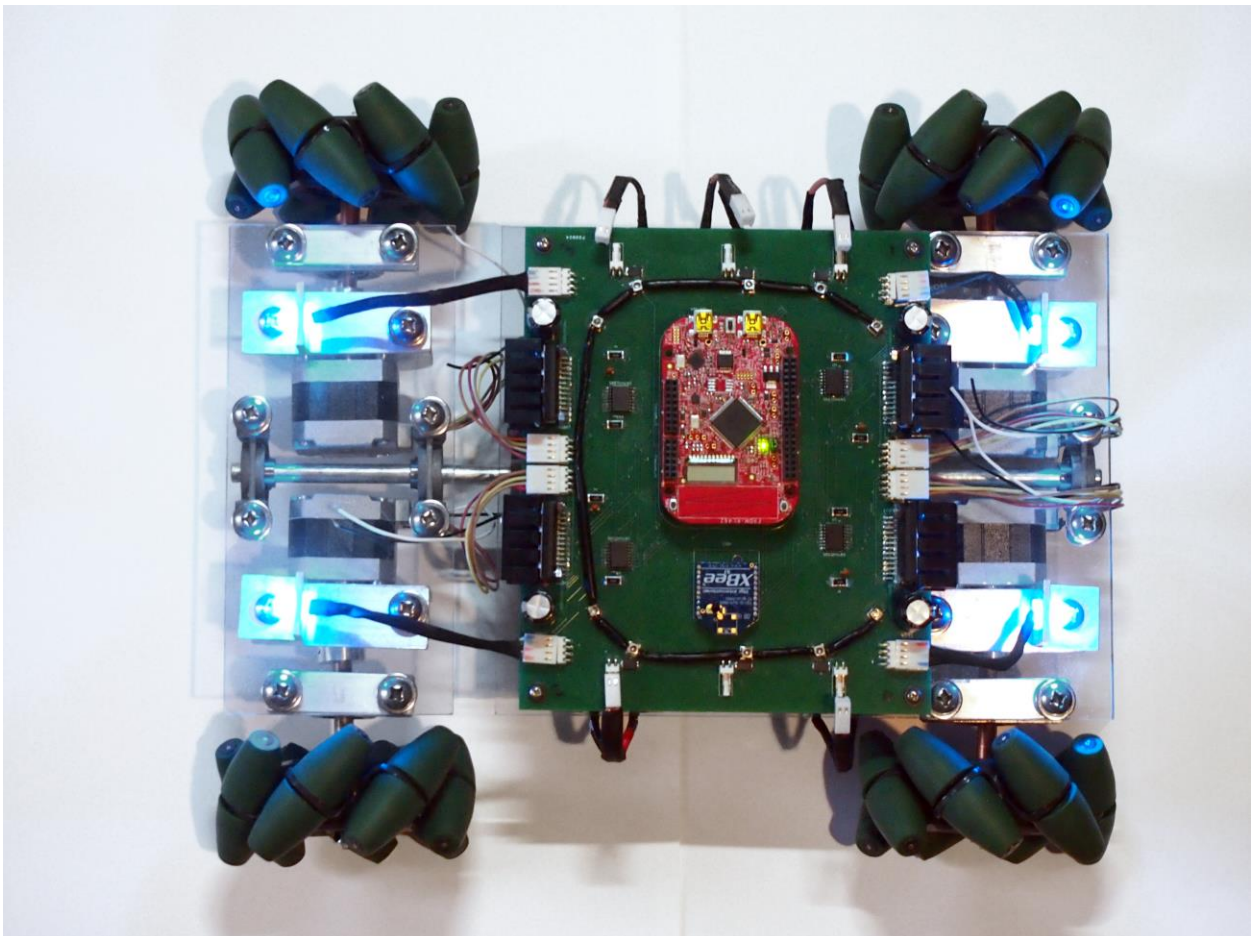


Figure 9: Final Vehicle

References

- Freescale Semiconductor. (2013). *KL46 Sub-Family Reference Manual*. Retrieved from http://cache.freescale.com/files/microcontrollers/doc/ref_manual/KL46P121M48SF4RM.pdf
- Freescale Semiconductor. (n.d.). *FRDM-KL46Z Schematic*. Retrieved from http://cache.freescale.com/files/microcontrollers/hardware_tools/schematics/FRDM-KL46Z_SCH.pdf
- Freescale Semiconductor. (n.d.). *FRDM-KL46Z User's Manual*. Retrieved from http://cache.freescale.com/files/microcontrollers/doc/user_guide/FRDM-KL46Z_UM.pdf
- ON Semiconductor. (2013). *PWM Constant-Current Control Stepper Motor Driver Application Note*. Retrieved from http://www.onsemi.com/pub_link/Collateral/ANDLV8727-D.PDF