

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't 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

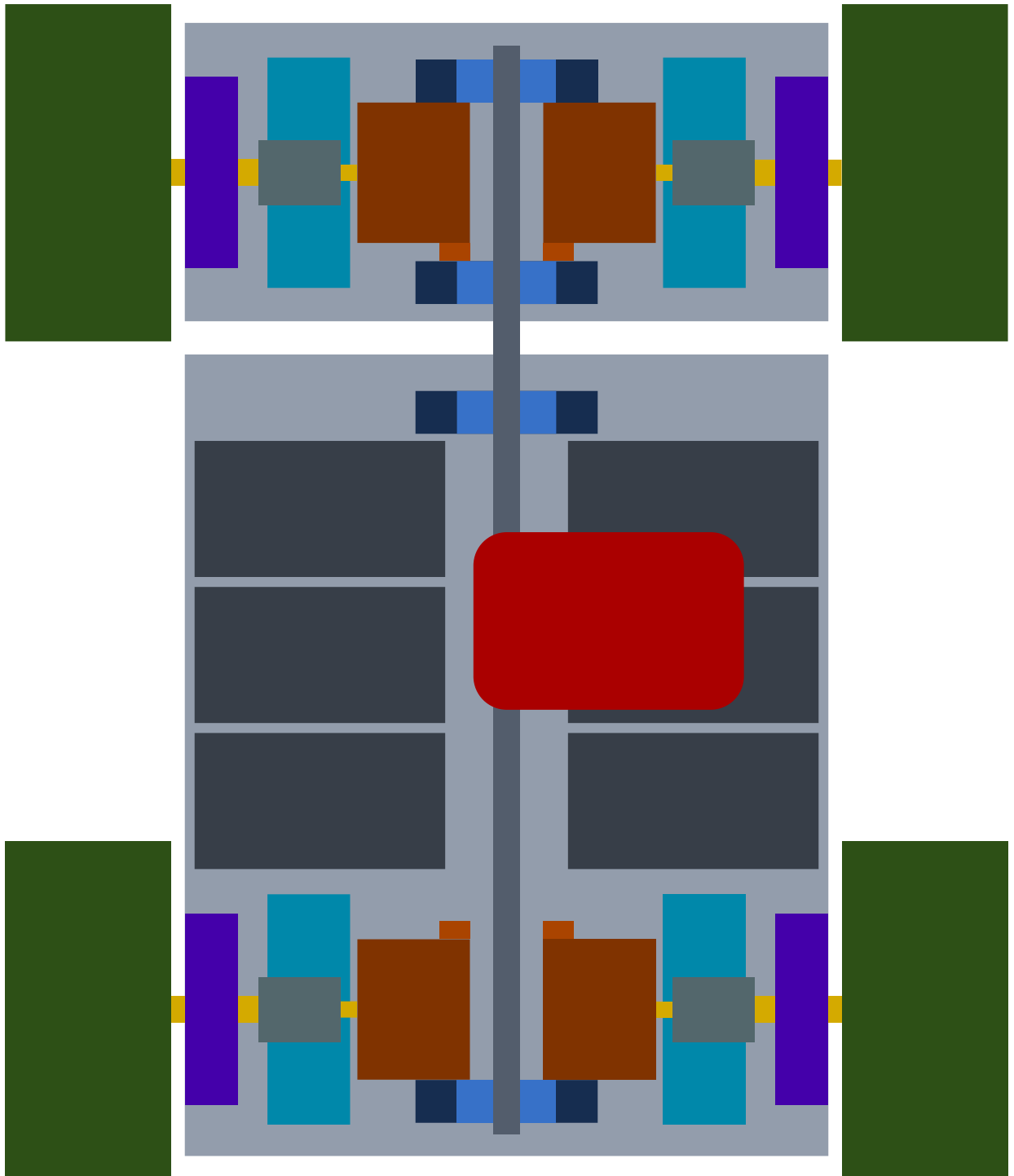


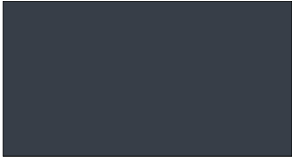





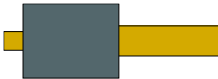


Figure 1: Vehicle Diagram

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

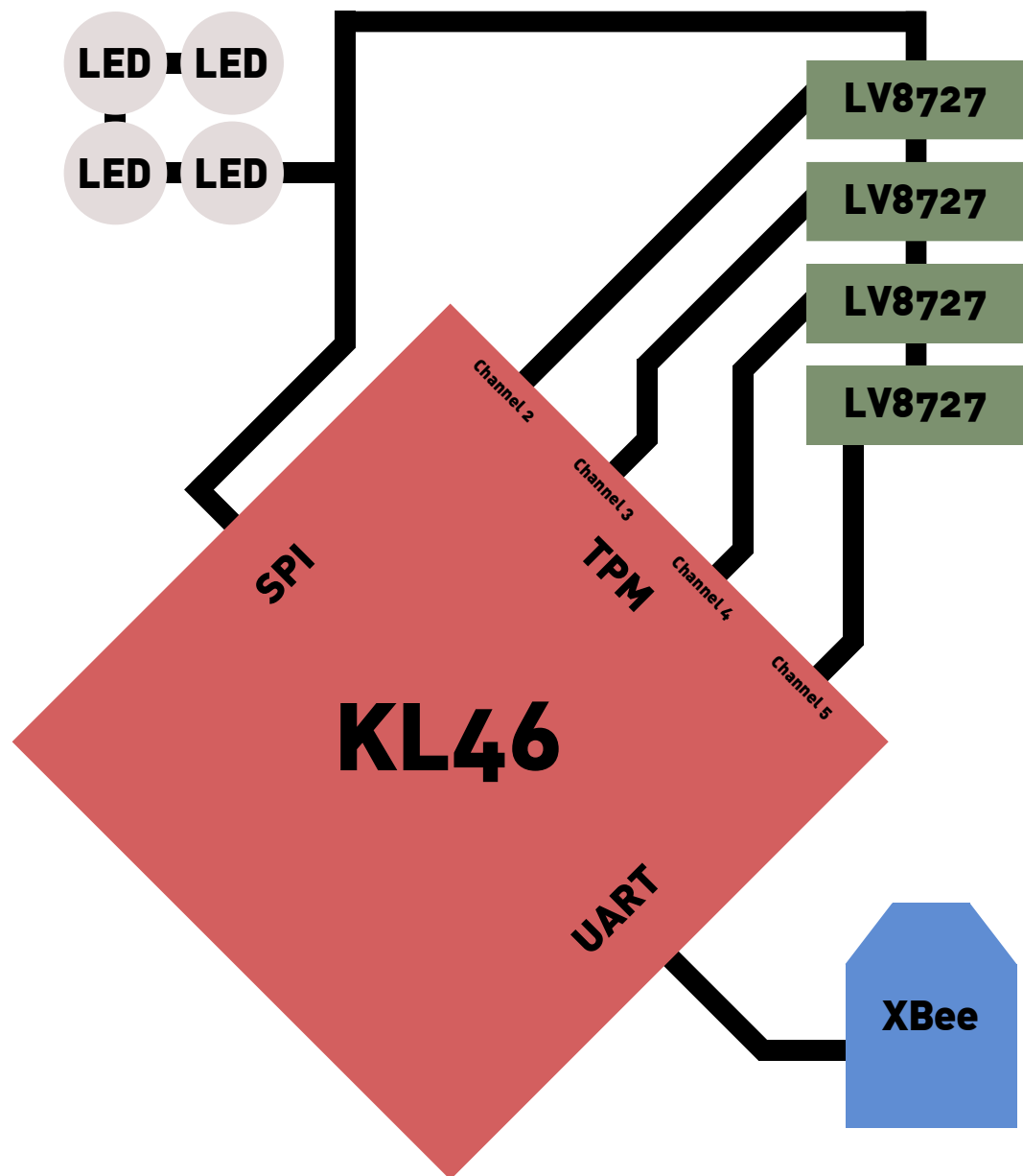


Figure 2: Block Diagram

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.

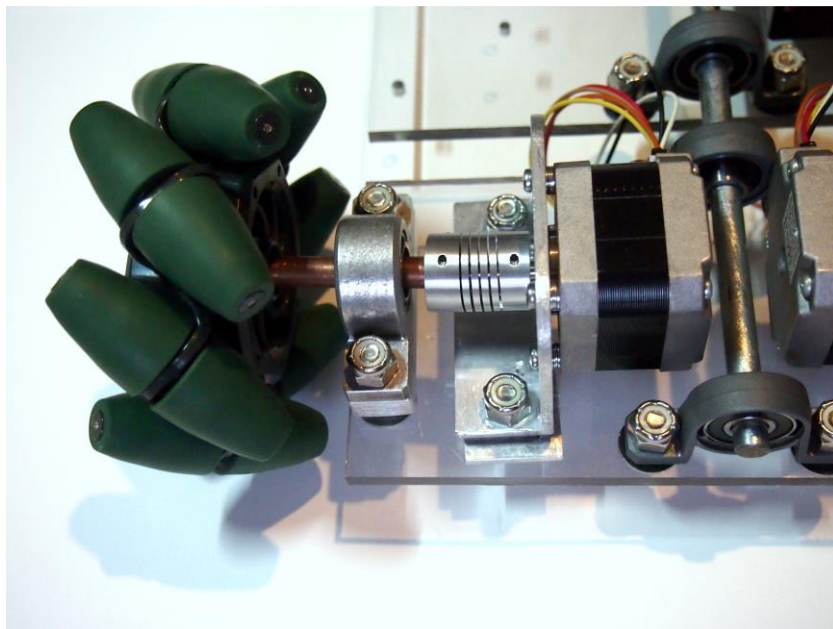


Figure 3: Shaft Assembly

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.

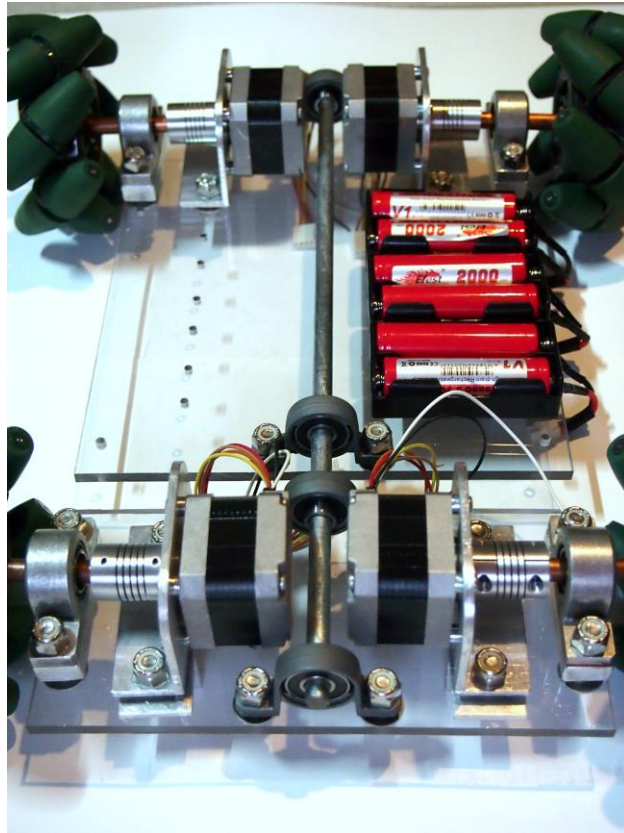


Figure 4: Vehicle Spine

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

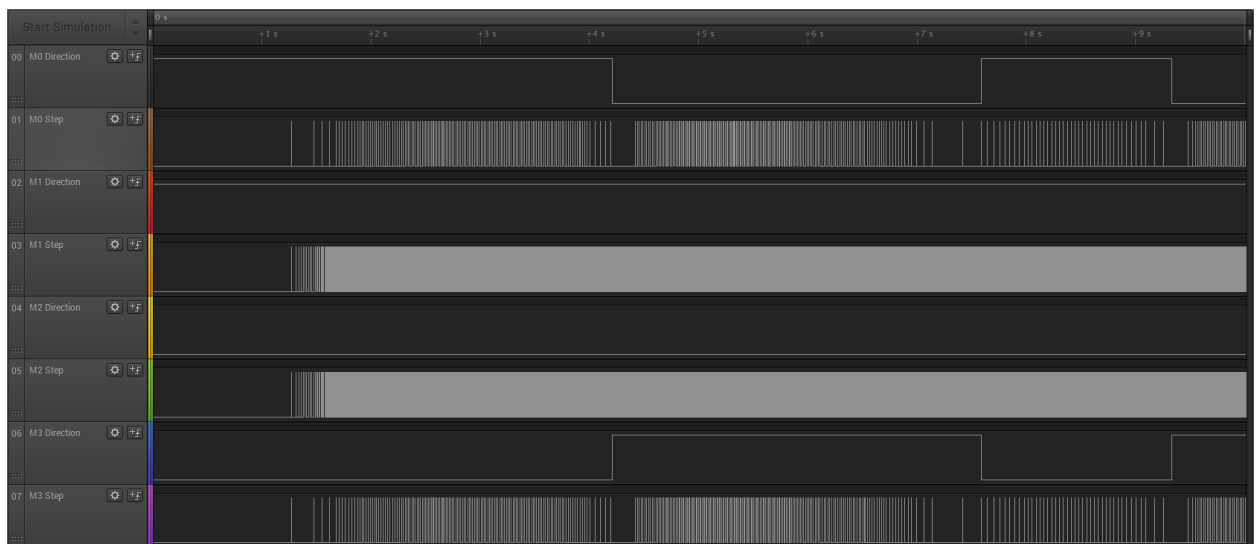


Figure 6: Diagonal Motion Pulses

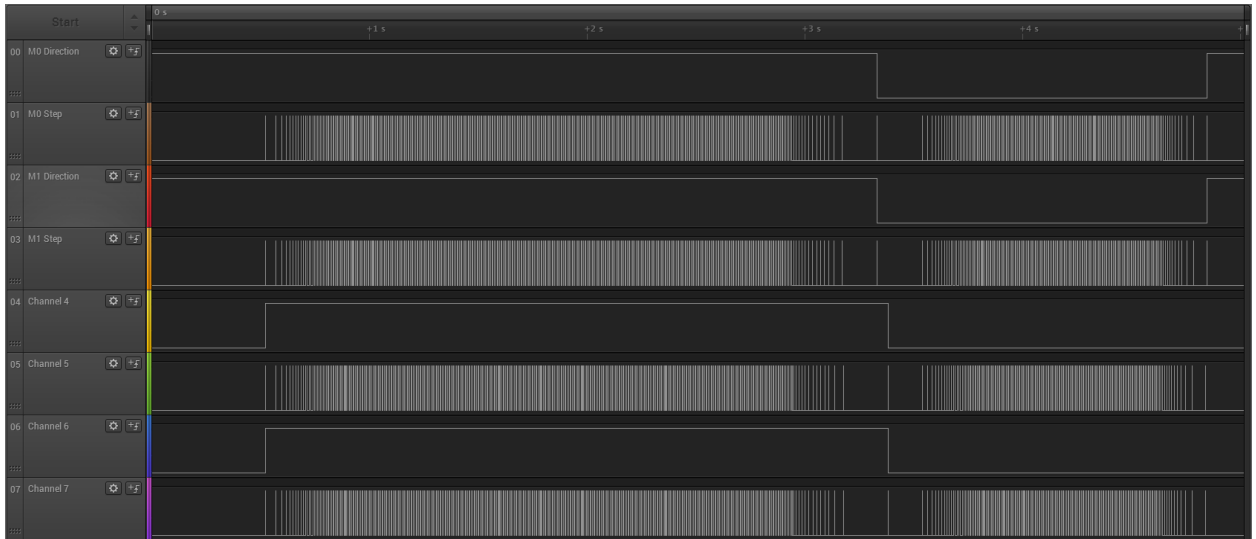


Figure 7: Rotational Motion Pulses

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.

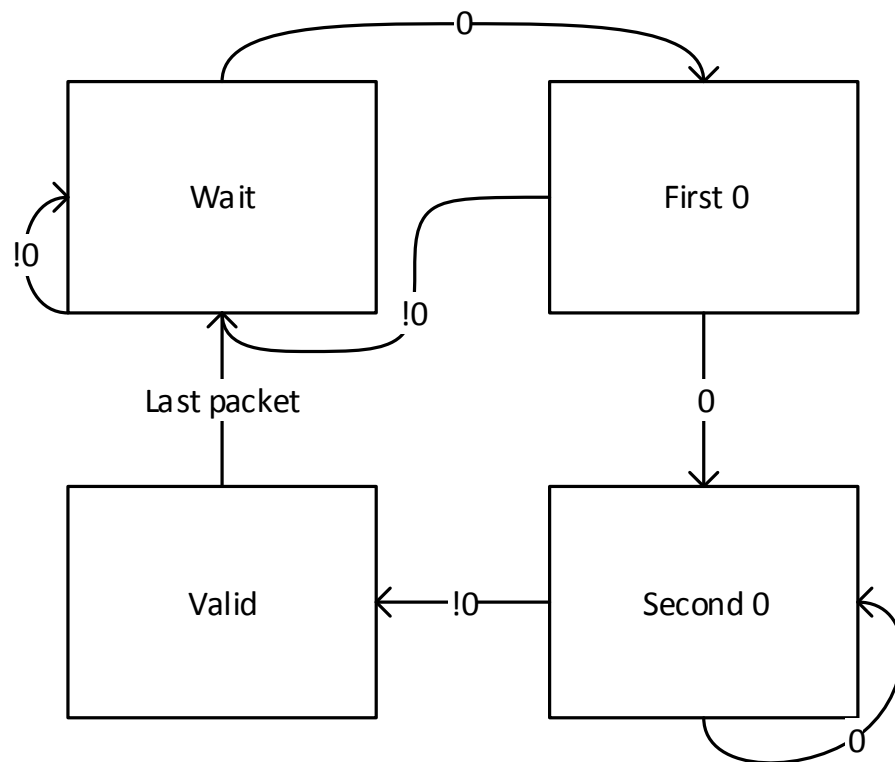


Figure 8: XBee State Diagram

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.

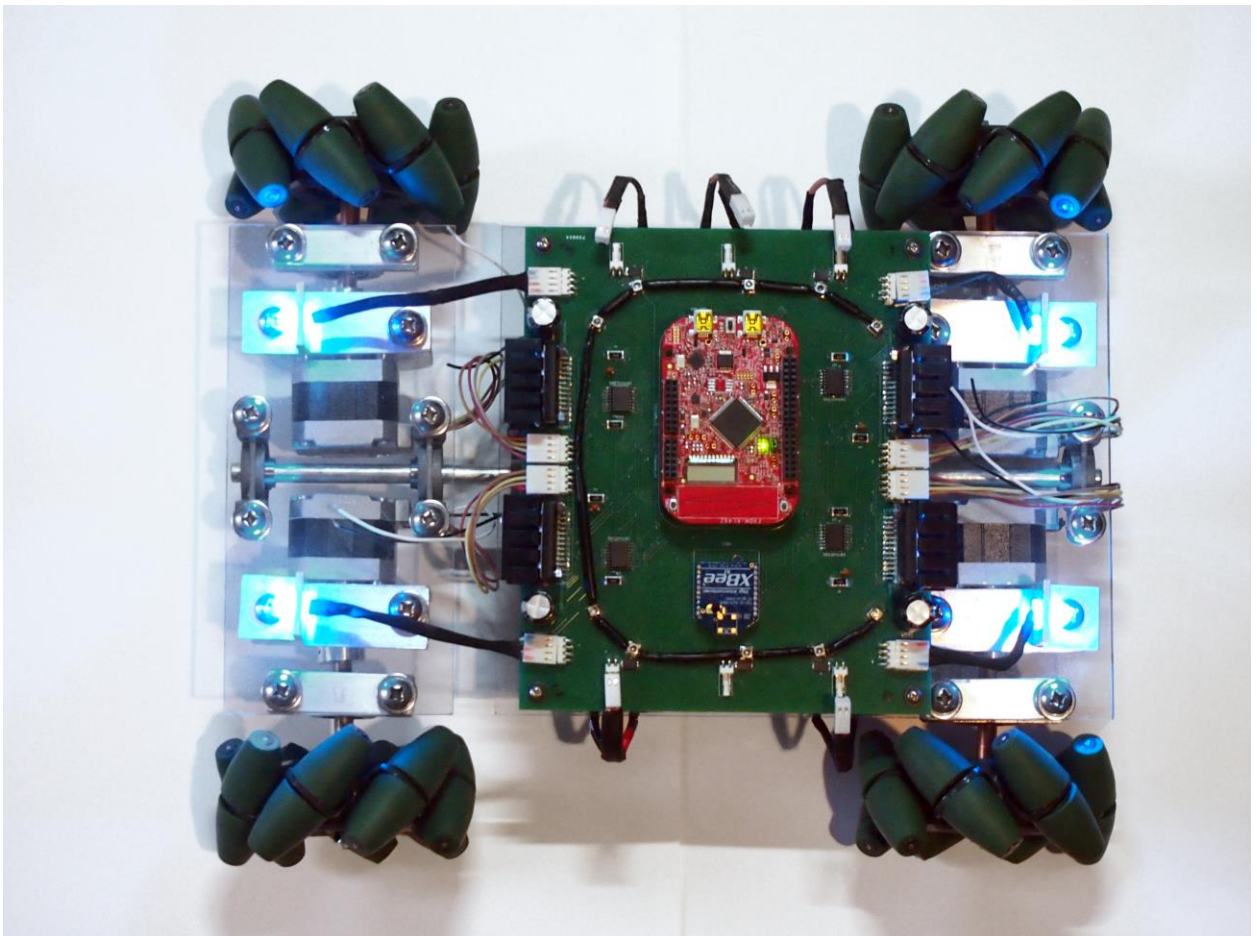


Figure 9: Final Vehicle

References

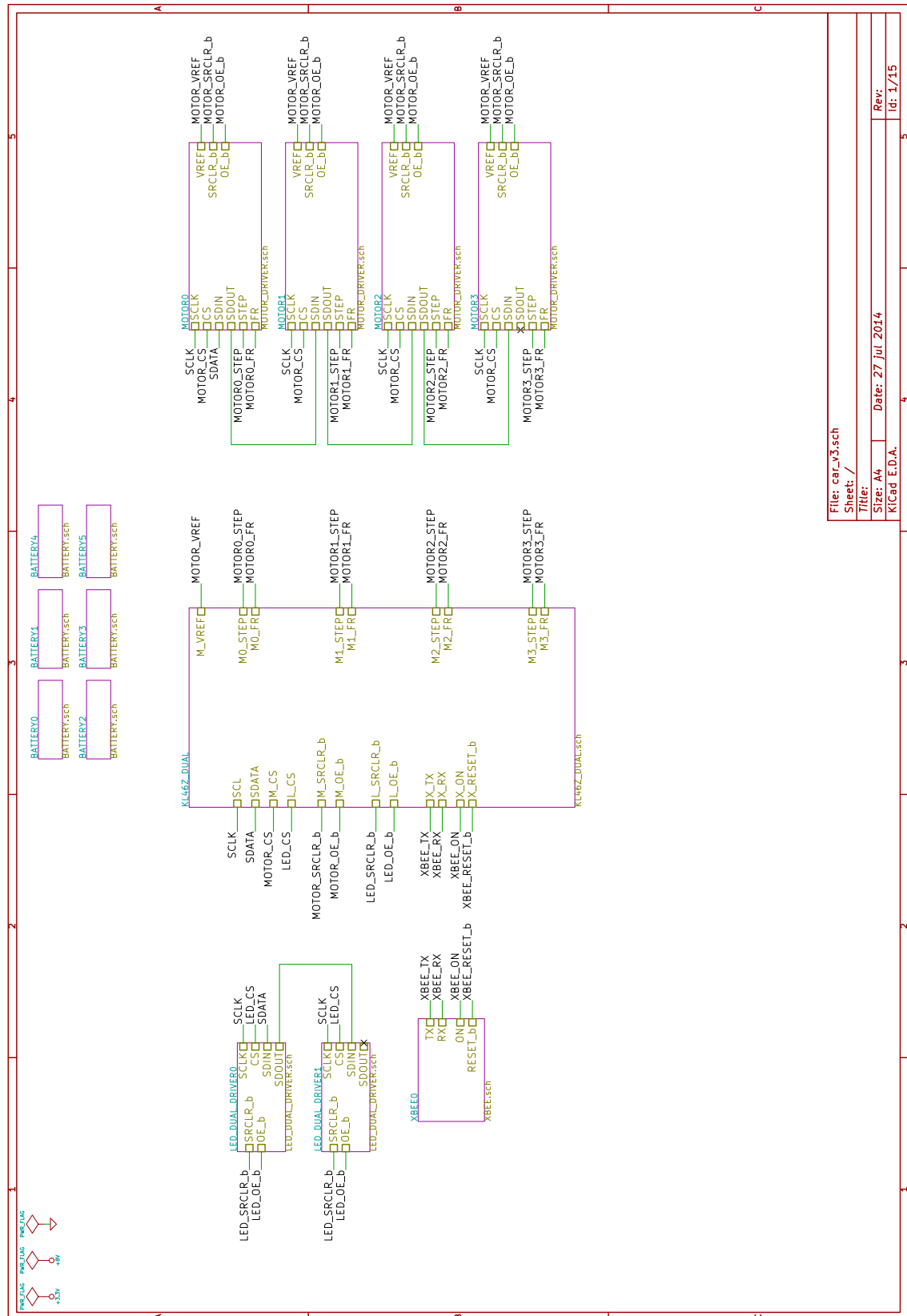
- Freescale Semiconductor. (2013). *KL46 Sub-Family Reference Manual*. Retrieved from http://cache.freescale.com/files/microcontrollers/doc/ref_manual/KL46P121M48SF4RM.pdf
- Freescale Semiconductor. (n.d.). *FRDM-KL46Z Schematic*. Retrieved from http://cache.freescale.com/files/microcontrollers/hardware_tools/schematics/FRDM-KL46Z_SCH.pdf
- 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

Appendix

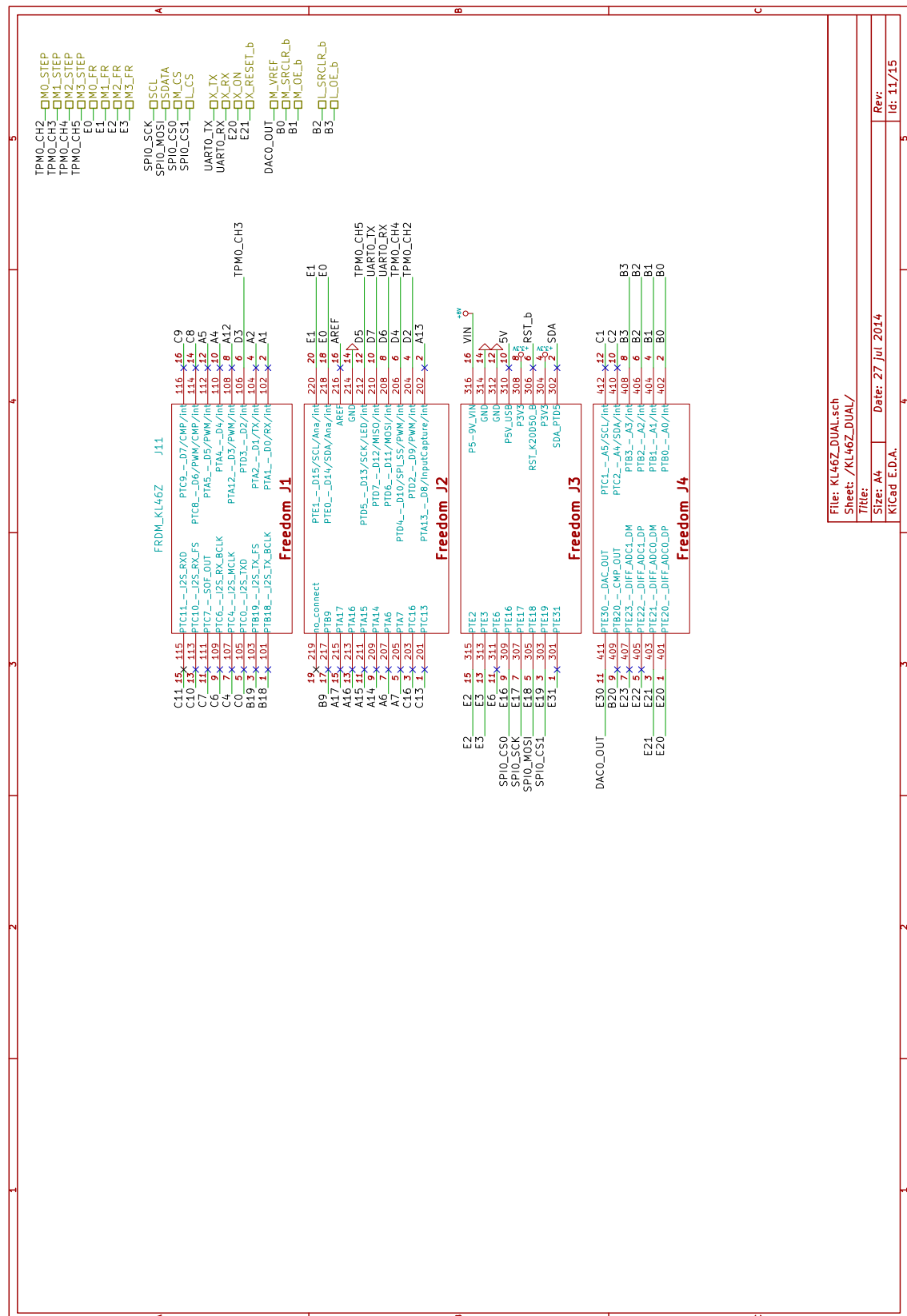
Contents

Appendix	24
Top Level Schematic	25
Microcontroller Schematic	26
Motor Driver Schematic	27
LED Driver Schematic	28
XBee Schematic.....	29
Power Schematic.....	30
Front Copper.....	31
Back Copper	32
Front Silkscreen	33
PCB Parts	34
Vehicle Parts.....	35
Curve and Approximation	36
Kinetis	37
Code Repository.....	39
Control Code	39
Vehicle Code	46

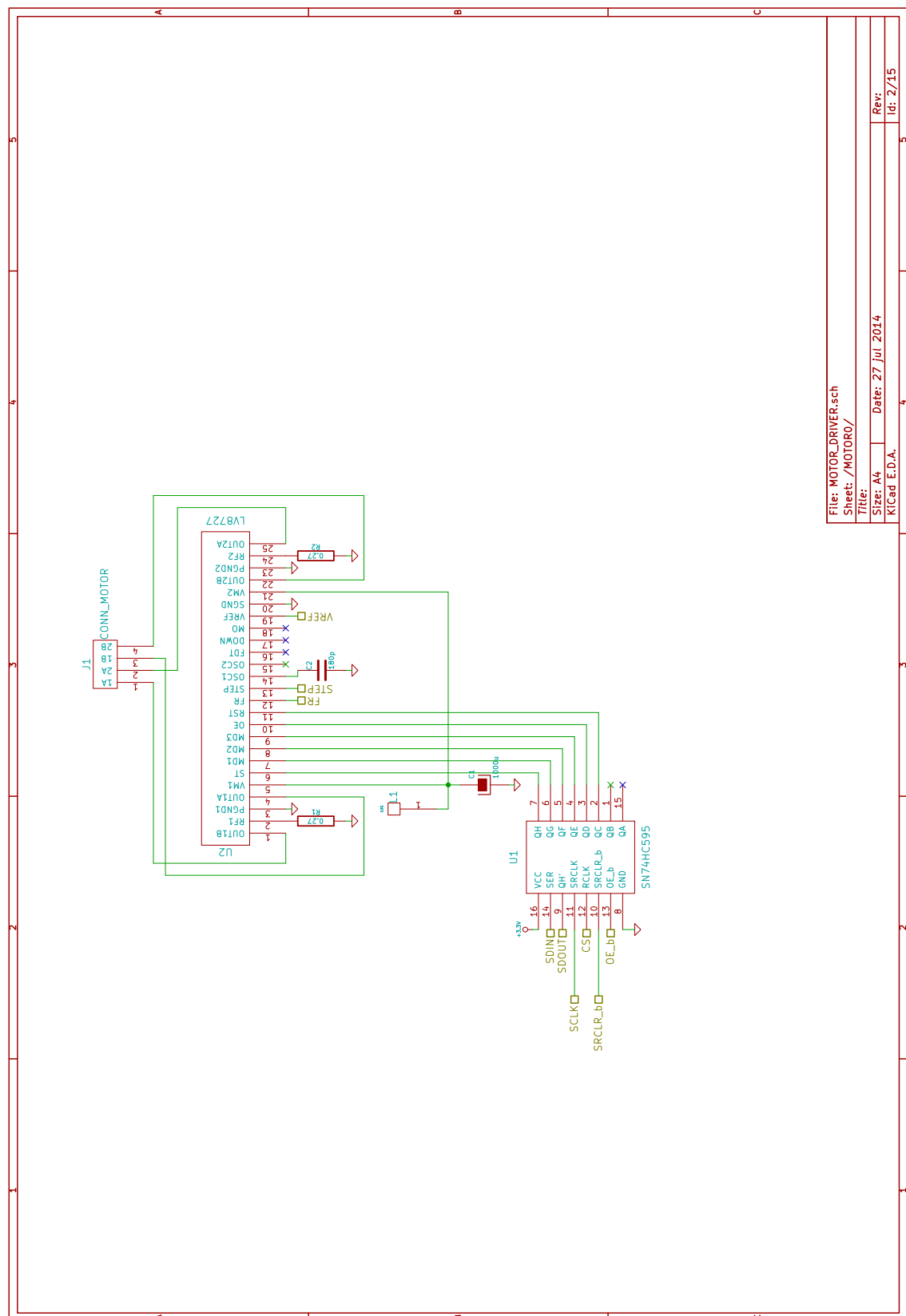
Top Level Schematic



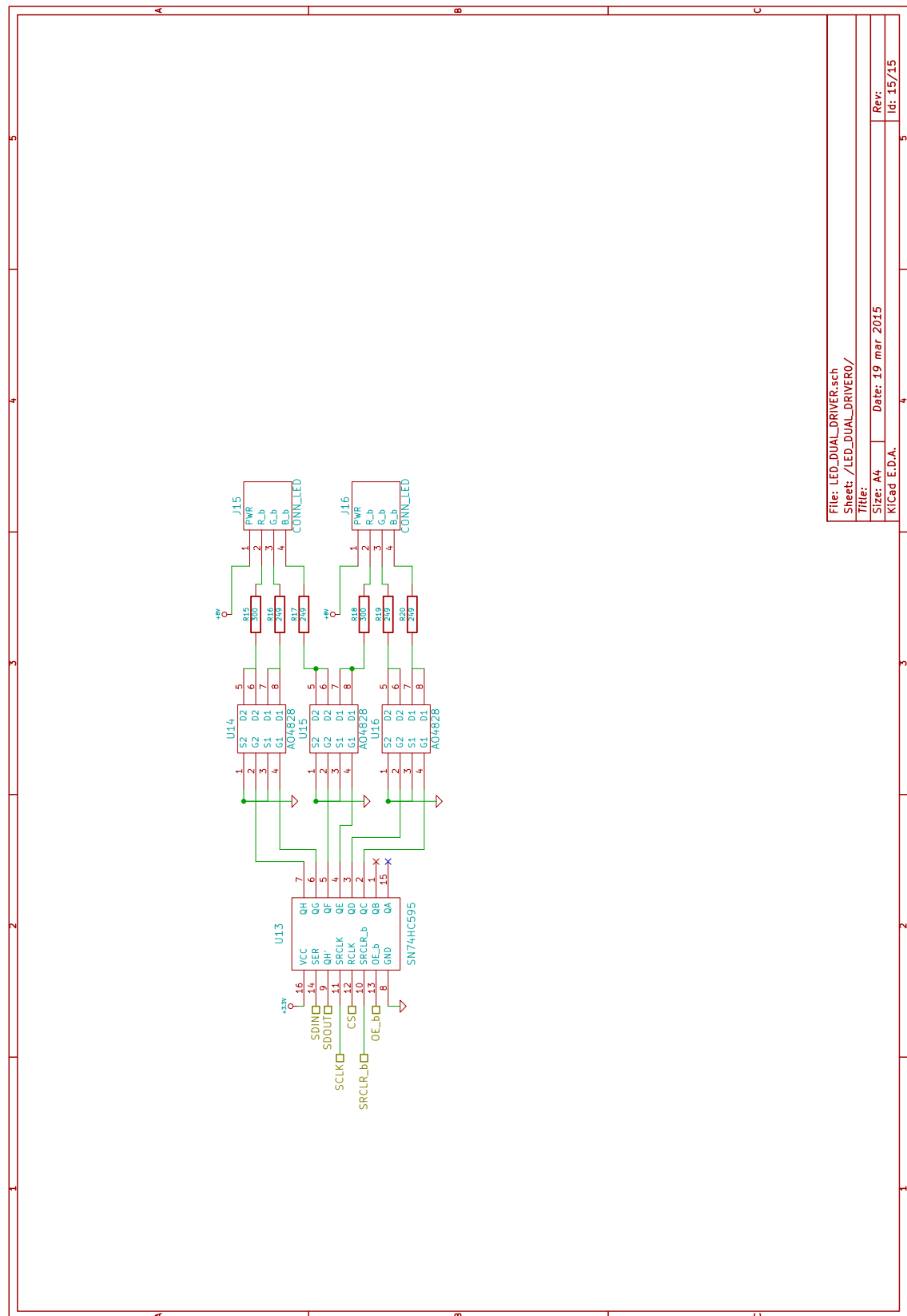
Microcontroller Schematic



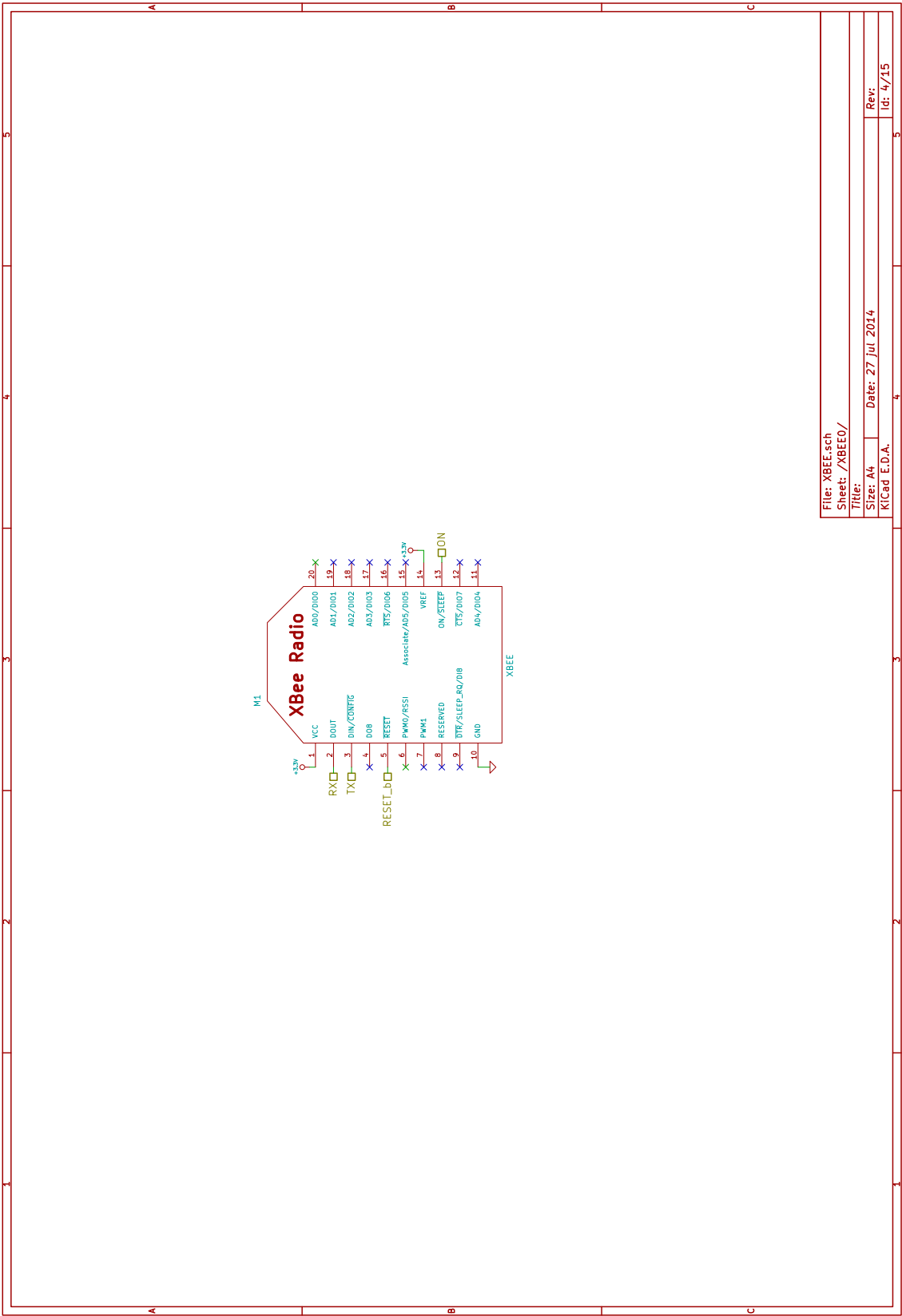
Motor Driver Schematic



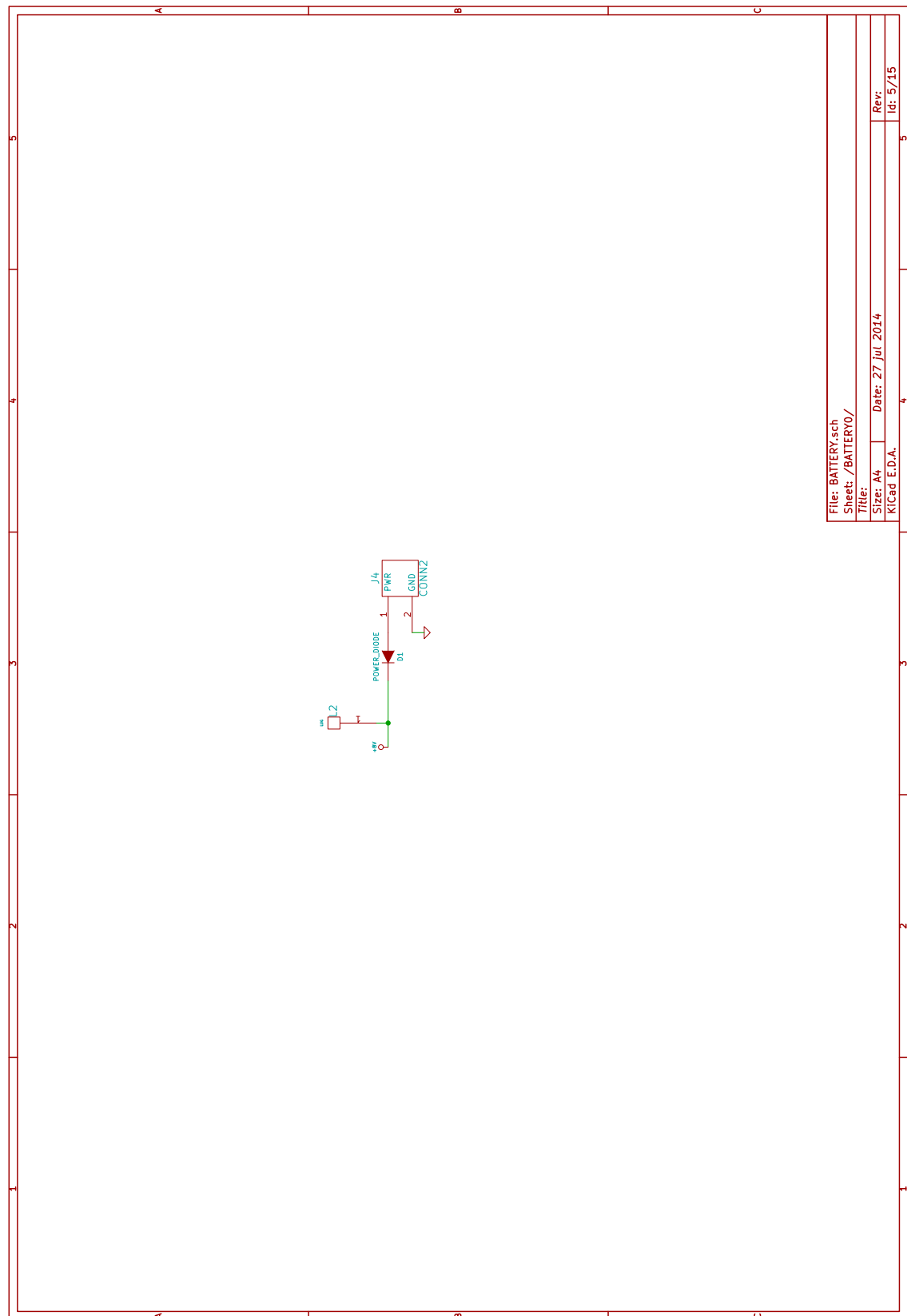
LED Driver Schematic



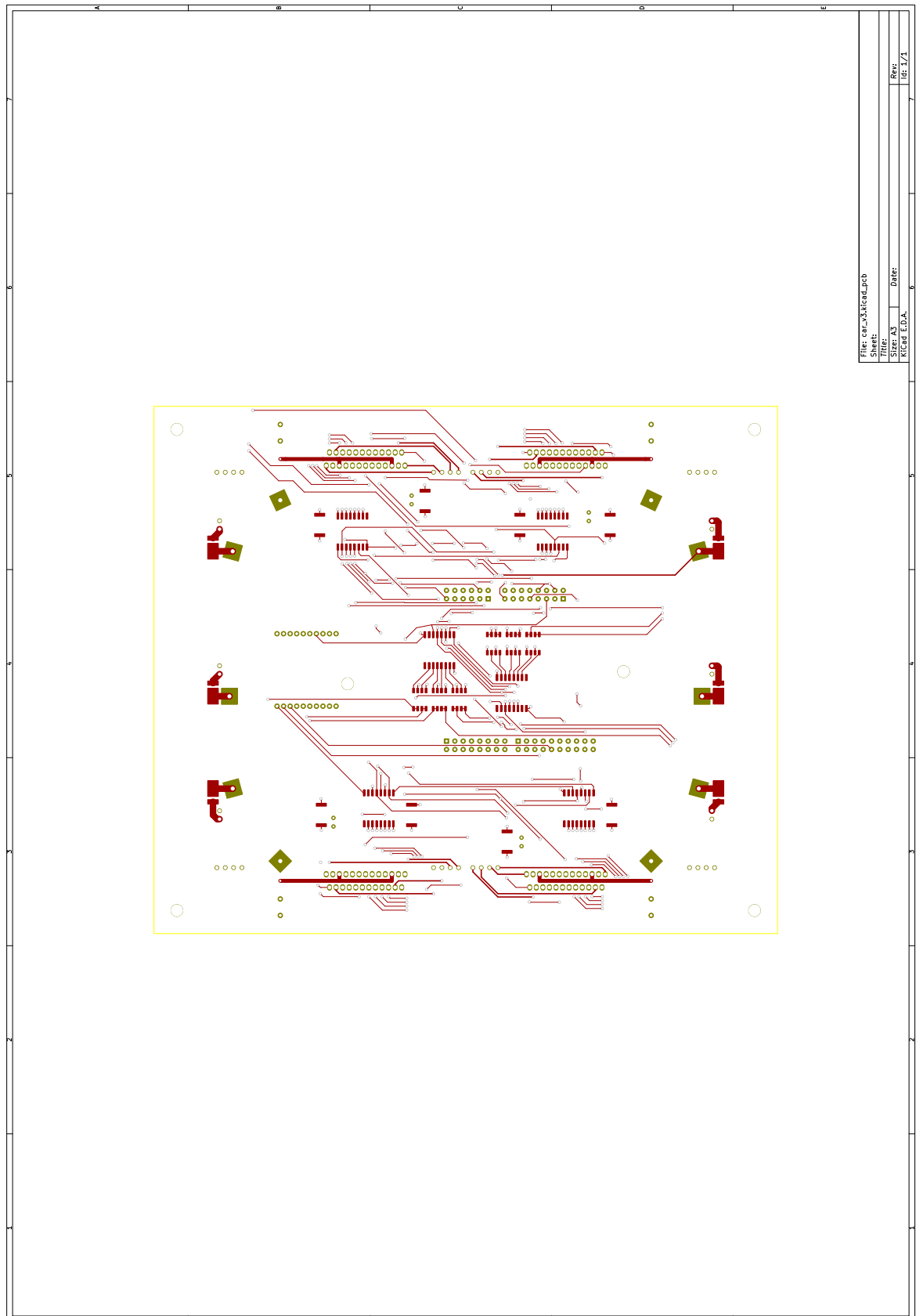
XBee Schematic



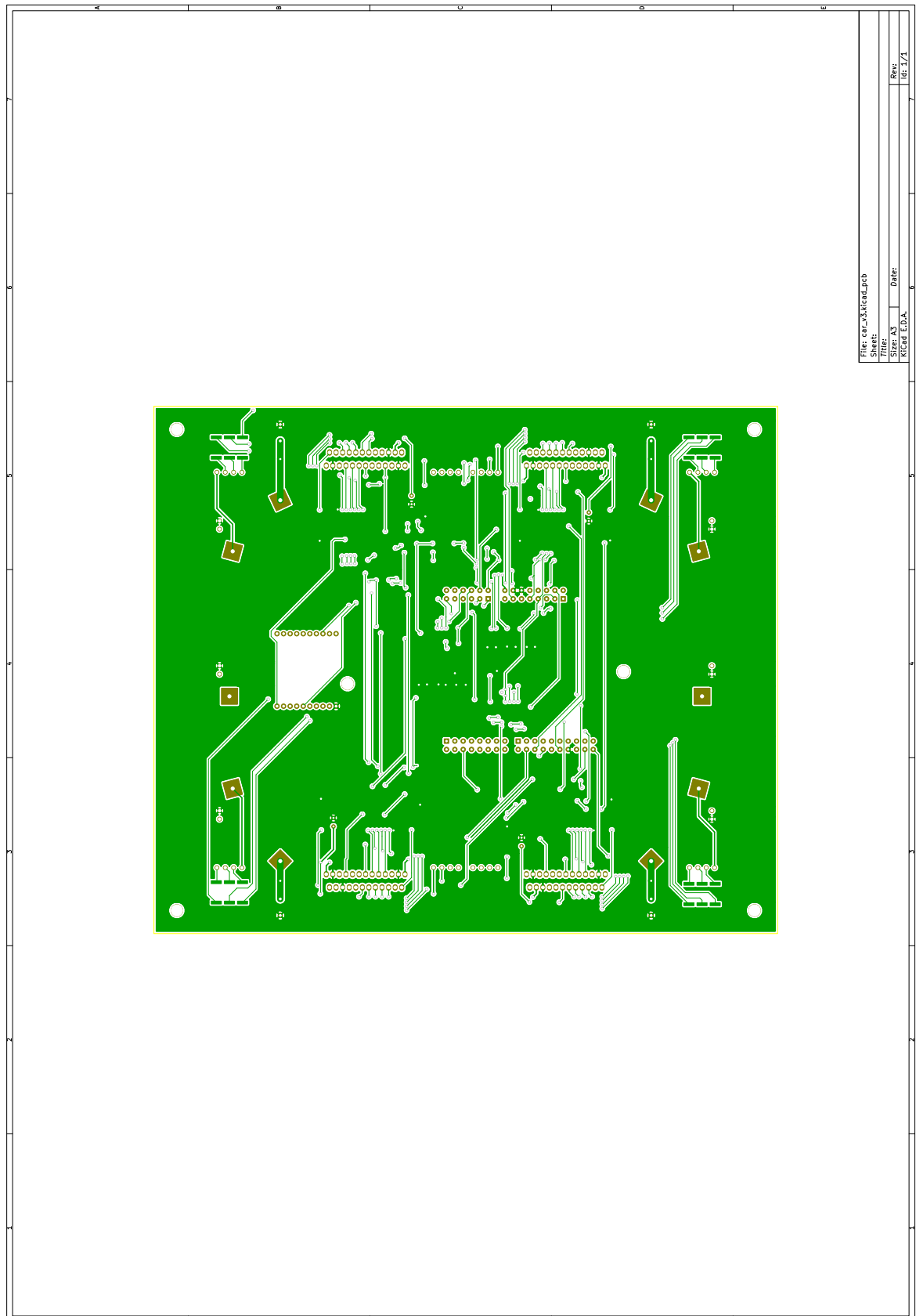
Power Schematic



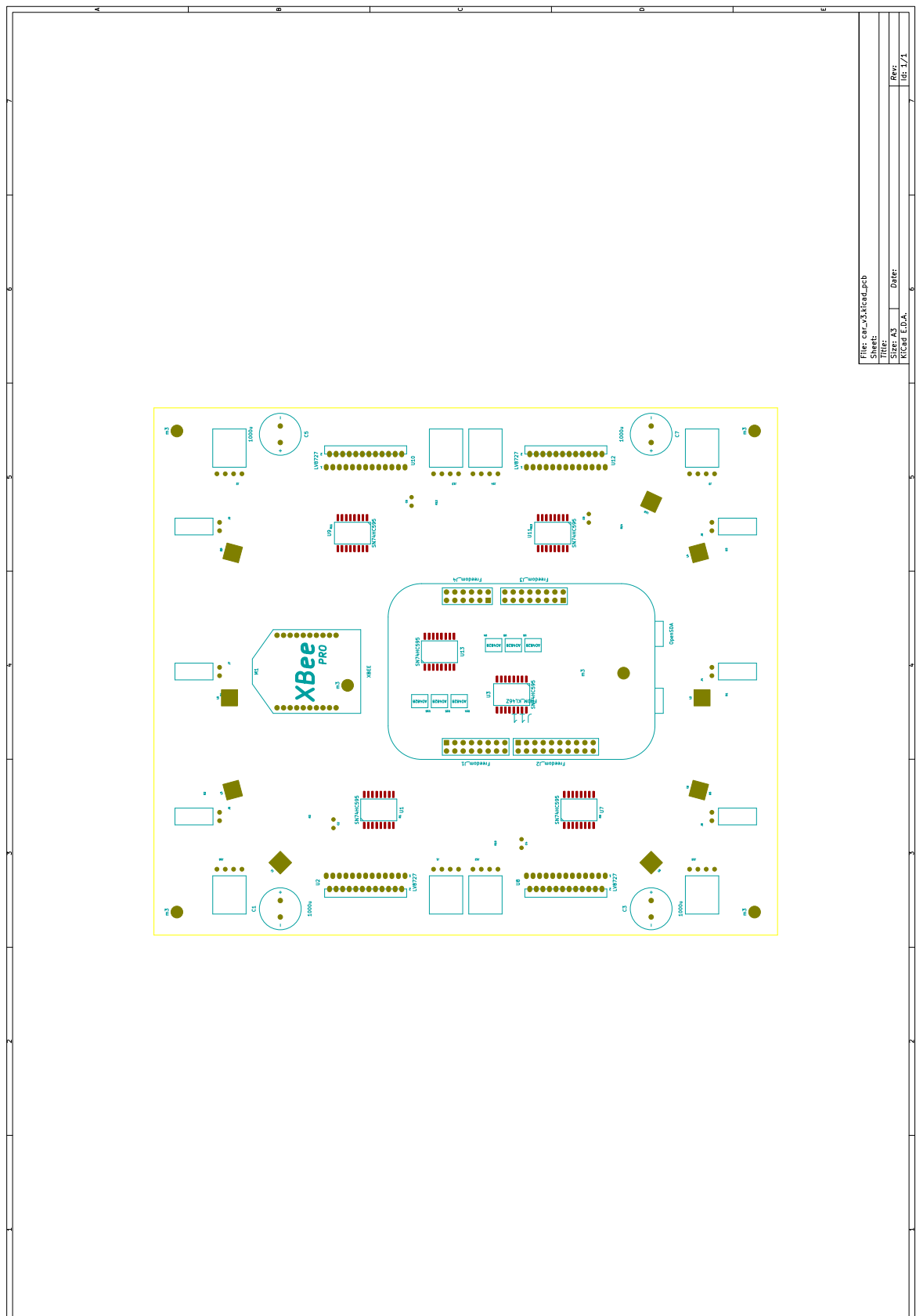
Front Copper



Back Copper



Front Silkscreen



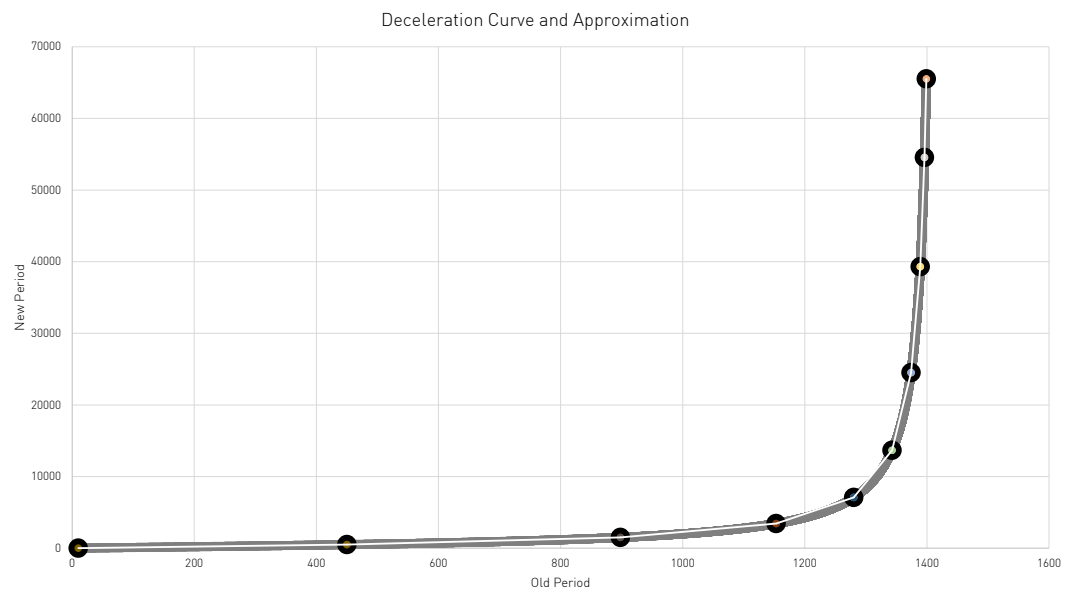
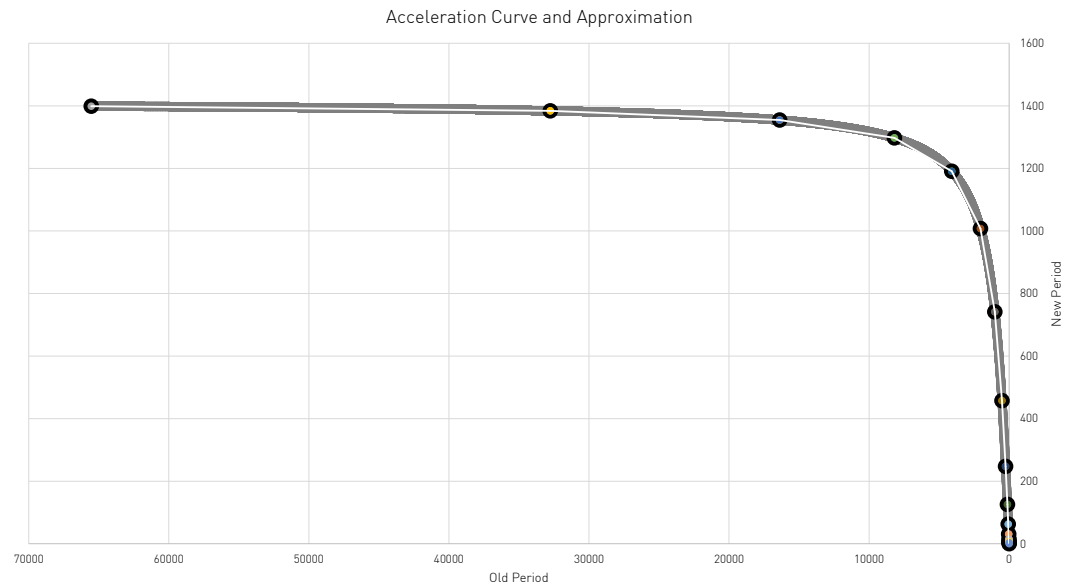
PCB Parts

Quantity	Part	Purpose	Source
1	FRDM-KL46Z	Microcontroller Board	Digikey
4	10-pin 0.1" header	Connects MC to PCB	Ebay
8	8-pin 0.1" header	Connects MC to PCB	Ebay
4	6-pin 0.1" header	Connects MC to PCB	Ebay
4	180pF Capacitor	Motor Driver input	Ebay
4	1000uF Capacitor	Power Buffer	Ebay
8	0.270hm Resistor	Motor Driver input	Digikey
6	Power Diode	Battery protection	Digikey
4	Heatsink	Attaches to motor drivers	Digikey
6	Shift Register	Feeds motor driver and LEDs	Digikey
6	Dual N-CH Mosfet	Feeds LEDs	Digikey
8	2490hm Resistor	Limits LED current	Digikey
4	3000hm Resistor	Limits LED current	Digikey
4	4 pin Connector	Connects to LEDs	Ebay
6	2 pin Connector	Connects to batteries	Ebay
8	10mm m3 panhead machine screw	Attaches heatsinks	Commonly available
12	12mm m3 flathead machine screw	Attaches battery packs	Commonly available
4	18mm m3 panhead machine screw	Attaches board	Commonly available
12	m3 star washers	Reduces nut loosening	Commonly available
24	m3 hex nut	Heatsinks and Bat. Packs	Commonly available
8	m3 washer	Applies flat pressure on heatsink mount	Commonly available

Vehicle Parts

Quantity	Part	Purpose	Source
4	4" Mecanum Wheels	Multidirectional wheels	VexRobotics
4	Stepper Motors	Powers wheels	Ebay
4	5mm to 8mm Coupler	Couples motors to wheels	
6	Battery Pack	Holds batteries	Ebay
1	¼" Plexiglass Sheet	Vehicle body	Commonly available
12	LI-MN 2000mAh 3.7V Battery	Powersv	Commonly available
1	5/16" Rod	Vehicle spine	Home Depot
4	8mm Bearing	Holds spine	Amazon
8	1/4"X 1-1/2" Phillip Machine Screw	Holds motor bearings	AlbanyCountyFasteners
20	1/4" Washers	Protects plexiglass	AlbanyCountyFasteners
16	1/4" X 5/8" Phillip Machine Screw	Holds bearings, motors, LEDs	AlbanyCountyFasteners

Curve and Approximation



Kinetis

This section is a small collection of tips to ease the learning curve of Kinetis development.

Interrupt Controller

Codewarrior 10.6 auto generates several files for the KL46Z. One of these includes function prototypes for all the functions called from the interrupt vector table (Project_Settings → Startup_Code → kinetis_sysinit.c). These are “weakly” tied to the “Default Handler,” so each of these has to be defined in code elsewhere to customize their behavior. This involves simply copying the function name into your relevant source file and filling in the contents of the function to clear whatever interrupt flag may be set and then typically call some kind of function that implements project specific functionality via a callback.

Additionally a header is generated that contains the typical register macros. In the case of KL46Z and CW10.6 this is called MKL46Z.h (Project_Settings → Project_Headers → MKL46Z4.h). The first section of this file has a helpful list of interrupt sources as a typedef enum. Thus, creating one manually is no longer required; you can just use the interrupt source names listed there.

Interrupts each have one of 4 priorities. The lower the number, the higher the priority. Likely all of the interrupt initialization code you might need is already all done in int.h.

Power

Since the Kinetis Cortex-M0+ focuses heavily on power usage, all GPIO and modules are disabled by default. The clocks to these have to be enabled manually. This occurs in all my init functions and typically involves something being written to an MCG (Multipurpose Clock Generator) register.

If single stepping through in debug your code suddenly jumps to the “Default Handler,” you probably forgot to enable the clock to a module.

By default the clock speed on the 46Z appears to be 24MHz. You’ll likely want full speed at 48MHz though. The code that does this is, is located in main.c in “clock_init”.

SPI

Although the 46Z includes a SPI module with multiple internal CS pins on each module, only 1 CS pin is available off the chip. As such, communicating with multiple peripheral requires manual toggling of CS pins. Unfortunately, there doesn't seem to be any way of receiving feedback from the module to know when a byte has been sent. As such, there is a small busy wait loop in the SPI send function that forces the processor to wait a certain amount of time for the transfer to complete before flipping the CS pin. This, of course depends on the baud rate (a faster transfer rate will require less wait time until flipping the CS pin). Currently this is hardcoded, but could obviously be changed to adjust to the configured baud rate. Alternatively, an interrupt -based approach could be used in conjunction with one of the PITs or the LPTMR to generate an interrupt that will toggle the CS pin. This would require an extra layer of software to prevent other calls to the SPI send function to occur before the message is done sending.

UART

UART0, which is the module used in this project, has an additional divider field on the incoming clock called OSR. This needs to be factored into the baud rate calculation. The OSR divider, as far as I know, does not exist for the other UART modules.

Switches

Since the Kinetis can generate interrupts directly off incoming signals, setting up the onboard switches is fairly simple. However, an internal pull up resistor must be enabled. If it isn't, the switch will only work for one push.

Code Repository

The code shown on the next pages is available online in my repository at:

[Wireless Mecanum Wheel Vehicle - Google Project Hosting](#)

Changes will likely be made after this document is submitted, so the link above will guarantee the latest version.

The code used to for serial communication can be found online at:

[CSerial - A C++ Class for Serial Communications](#)

The serial code is not included in this document but is also hosted in my repository.

Control Code

```
#include <Windows.h>
#include <xinput.h>
#include <stdio.h>
#include <time.h>
#include <string>
#include <iostream>
#include <math.h>

#include "Serial.h"

#define BAUDRATE 9600
#define STOP_PERIOD 65535
#define MAX_PERIOD 65534.0
#define MIN_PERIOD 40.0
#define RTH_SCALER 4.0
#define PI 3.14159265

// #define XBOX_DEBUG
// #define RAW_VAL_PRINT
// #define RAW_MOTOR_PRINT
// #define PERIOD_PRINT
// #define PCKG_PRINT

using namespace std;
void prompt_serial(CSerial * port_pntr);
void prompt_controller(XINPUT_STATE * state);
void poll(XINPUT_STATE * state, CSerial * port_pntr);
void sleep(unsigned int mseconds);
void xbox_debug_msg(XINPUT_STATE * p_state);
int compute_period(double raw_velocity);
void translate_and_send(int * m_period);

CSerial port;
double max_velocity;
double min_velocity;

void main()
{
    // connect to serial port
    prompt_serial(&port);
    XINPUT_STATE state;
```



```

    //connect to controller
    prompt_controller(&state);

    //get max and min velocities
    max_velocity = 1 / MIN_PERIOD;
    min_velocity = 1 / MAX_PERIOD;

    while (1)
    {
        poll(&state, &port);
        sleep(200);
    }

    /*
    gets the angle of the thumbstick
    */
    double get_theta(double x, double y)
    {
        double theta = 0;
        //check that nothign is 0
        if (!(x == 0) && (y == 0))
        {
            theta = atan2((double)y, (double)x);
        }

        if (y < 0)
        {
            theta = theta + (2 * PI);
        }

        return theta;
    }

    /*
    gets magnitude of the thumbstick
    */
    double get_mag(double x, double y)
    {
        double y_sqrd = ((double)y)*((double)y);
        double x_sqrd = ((double)x)*((double)x);

        double mag = sqrt(x_sqrd + y_sqrd);

        //cap value at max
        if (mag > 32767)
        {
            mag = 32767;
        }
        return mag;
    }

    /*
    polls the controller
    */
    void poll(XINPUT_STATE * state, CSerial * port_pntr)
    {
        //get controller state
        prompt_controller(state);

        #ifdef XBOX_DEBUG
            xbox_debug_msg(state);
        #endif
    }

```

```

double lmag = get_mag(state->Gamepad.sThumbLX, state->Gamepad.sThumbLY);
double ltheta = get_theta(state->Gamepad.sThumbLX, state->Gamepad.sThumbLY);
//only care about x coordinate on right thumbstick
double rmag = state->Gamepad.sThumbRX;

//check if lstick is within deadzone
if (lmag < XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE)
{
    lmag = 0;
    ltheta = 0;
}
else
{
    //scale left magnitude to value between 0 and 1
    lmag = (lmag - XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE) / (32767 -
        XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE);
}

//check if rstick is within deadzone
if (abs(rmag) < XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE)
{
    rmag = 0;
}
else
{
    if (rmag < -32767)
    {
        rmag = -32767;
    }
    //scale right magnitude o value between 0 and 1
    if (rmag > 0)
    {
        rmag = (rmag - XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE) / (32767 -
            XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE);
    }

    else if (rmag < 0)
    {
        rmag = (rmag + XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE) / (32767 -
            XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE);
    }
}

#ifdef RAW_VAL_PRINT
cout << "LTheta is: " << ltheta << endl;
cout << "LMag is:" << lmag << endl;
cout << "RMag is:" << rmag << endl << endl;
#endif

//scaled right thumb
rmag = rmag / RTH_SCALER;

double PI_div_4 = PI / 4;
double sin_term = sin(ltheta + (PI_div_4));
double cos_term = cos(ltheta + (PI_div_4));

//construct raw motor velocities
double raw_m0_vel = (lmag * cos_term) + rmag;
double raw_m1_vel = (lmag * sin_term) + rmag;
double raw_m2_vel = (lmag * sin_term) - rmag;
double raw_m3_vel = (lmag * cos_term) - rmag;

//scale raw velocities to between 0 and 1
raw_m0_vel = raw_m0_vel / (1 + (1/RTH_SCALER));

```

```

    raw_m1_vel = raw_m1_vel / (1 + (1/RTH_SCALER));
    raw_m2_vel = raw_m2_vel / (1 + (1/RTH_SCALER));
    raw_m3_vel = raw_m3_vel / (1 + (1/RTH_SCALER));

#ifdef RAW_MOTOR_PRINT
    cout << "m0: " << raw_m0_vel << endl;
    cout << "m1: " << raw_m1_vel << endl;
    cout << "m2: " << raw_m2_vel << endl;
    cout << "m3: " << raw_m3_vel << endl << endl;
#endif

    int m_period[4];

    //compute periods from raw
    m_period[0] = compute_period(raw_m0_vel);
    m_period[1] = compute_period(raw_m1_vel);
    m_period[2] = compute_period(raw_m2_vel);
    m_period[3] = compute_period(raw_m3_vel);

    //0 and 3 needs to be reversed
    m_period[0] *= -1;
    m_period[3] *= -1;

#ifdef PERIOD_PRINT
    cout << "m0_period: " << m_period[0] << endl;
    cout << "m1_period: " << m_period[1] << endl;
    cout << "m2_period: " << m_period[2] << endl;
    cout << "m3_period: " << m_period[3] << endl << endl;
#endif

    translate_and_send(m_period);
}

/*
sleeps for specified milliseconds
*/
void sleep(unsigned int mseconds)
{
    clock_t goal = mseconds + clock();
    while (goal > clock());
}

/*
Attempts to connect to a controller
*/
void prompt_controller(XINPUT_STATE * state)
{
    int flag = 0;
    //loops while no response from controller

    while( XInputGetState( 0, state ) != ERROR_SUCCESS )
    {
        if (flag == 0)
        {
            cout << "Controller not connected" << endl << endl;
            flag = 1;
        }
    }

    if (flag == 1)
    {
        cout << "Controller connected" << endl;
    }
}

```

```

}

/*
Attempts to connect to serial port
*/
void prompt_serial(CSerial * port_pntr)
{
    //print available ports
    cout << "Available COM ports" << endl;

    //scans ports
    for(int i = 1; i < 20; i++)
    {
        if(port_pntr->Open(i, BAUDRATE))
        {
            port_pntr->Close();
            cout << i << ". " << "COM" << i << endl;
        }
    }

    cout << endl << endl;

    cout << "Enter port number" << endl;

    //gets port number
    int port_num;
    cin >> port_num;

    //checks if port number was valid, loops if it wasn't
    while(!port_pntr->Open(port_num, BAUDRATE))
    {
        cout << endl << "Invalid port" << endl;
        cout << "Enter port number" << endl;
        cin >> port_num;
    }

    //rechecks if port is valid, prints message
    if(port_pntr->Open(port_num, BAUDRATE))
    {
        cout << "Connected to COM" << port_num << " with " << BAUDRATE << " baud"
        << endl << endl;
    }
}

/*
just prints the xbox controller state
*/
void xbox_debug_msg(XINPUT_STATE * p_state)
{
    if (p_state->Gamepad.wButtons & 0x0001)
    {
        cout << "UP is pressed" << endl;
    }
    if (p_state->Gamepad.wButtons & 0x0002)
    {
        cout << "DOWN is pressed" << endl;
    }
    if (p_state->Gamepad.wButtons & 0x0004)
    {
        cout << "LEFT is pressed" << endl;
    }
    if (p_state->Gamepad.wButtons & 0x0008)
    {
        cout << "RIGHT is pressed" << endl;
    }
}

```

```

}
if (p_state->Gamepad.wButtons & 0x0010)
{
    cout << "START is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x0020)
{
    cout << "BACK is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x0040)
{
    cout << "LTHMB is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x0080)
{
    cout << "RTHMB is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x0100)
{
    cout << "LSHLDR is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x0200)
{
    cout << "RSHLDR is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x1000)
{
    cout << "A is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x2000)
{
    cout << "B is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x4000)
{
    cout << "X is pressed" << endl;
}
if (p_state->Gamepad.wButtons & 0x8000)
{
    cout << "Y is pressed" << endl;
}
if (p_state->Gamepad.bLeftTrigger > XINPUT_GAMEPAD_TRIGGER_THRESHOLD)
{
    cout << "LT is pressed" << endl;
}
if (p_state->Gamepad.bRightTrigger > XINPUT_GAMEPAD_TRIGGER_THRESHOLD)
{
    cout << "RT is pressed" << endl;
}
if (abs(p_state->Gamepad.sThumbLX) > XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE)
{
    cout << "Left Thumbstick X Position is at" << p_state->Gamepad.sThumbLX <<
    endl;
}
if (abs(p_state->Gamepad.sThumbLY) > XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE)
{
    cout << "Left Thumbstick Y Position is at" << p_state->Gamepad.sThumbLY <<
    endl;
}
if (abs(p_state->Gamepad.sThumbRX) > XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE)
{
    cout << "Right Thumbstick X Position is at" << p_state->Gamepad.sThumbRX <<
    endl;
}
}

```

```

        if (abs(p_state->Gamepad.sThumbRY) > XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE)
        {
            cout << "Right Thumbstick Y Position is at" << p_state->Gamepad.sThumbRY <<
                endl;
        }
    }

    /*
    computes period from given velocity
    */
    int compute_period(double raw_velocity)
    {
        if (raw_velocity == 0)
        {
            return STOP_PERIOD;
        }
        else
        {
            double vel_range = max_velocity - min_velocity;

            double vel_scaled = (raw_velocity * vel_range) + min_velocity;
            double period = 1 / vel_scaled;
            return (int)period;
        }
    }

    /*
    packages up the four motor periods and sends them via the serial port
    */
    void translate_and_send(int * m_period)
    {
        char packets[12] = {};
        int converted_packet = 0;
        packets[0] = 0;
        packets[1] = 0;

        for (int i = 0; i < 4; i++)
        {
            int converted_packet = m_period[i];
            if (m_period[i] < 0)
            {
                converted_packet = m_period[i] * -1;
            }
            else
            {
                packets[2] |= (1 << i);
            }
            packets[3 + (i * 2)] = converted_packet & 0xFF;
            packets[3 + (i * 2) + 1] = (converted_packet >> 8) & 0xFF;
        }
        //mess with upper bits of direction packet
        packets[2] |= 0xA0;

        //compute xor
        packets[11] = packets[2] ^ packets[3] ^ packets[4] ^ packets[5] ^ packets[6] ^
            packets[7] ^ packets[8] ^ packets[9] ^ packets[10];

        for (int i = 0; i < 12; i++)
        {
            cout << i << " " << (int)packets[i] << endl;
        }
        cout << "sent " << port.SendData(packets, 12) << " packets" << endl;
    }

```

Vehicle Code

main.c

```
/*
 * main.c
 *
 * Created on: May 7, 2014
 * Author: Severin
 */

#include "derivative.h" /* include peripheral declarations */
#include "gpio.h"
#include "uc_pit.h"
#include "uc_spi.h"
#include "uc_sw.h"
#include "CAR_LED.h"
#include "CAR_MOTOR.h"
#include "CAR_XBEE.h"
#include "uc_uart.h"

#define PKT_NUM 10

typedef enum{
    wait,
    first,
    second,
    valid
} xbee_state;

int switch_3_push = 0;
int switch_1_push = 0;
int xbee_pkt_cnt = 0;
char raw_pkts[PKT_NUM] = {0,0,0,0,0,0,0,0,0,0};
int first_message = 0;
int deadman = 0;
int led_update = 0;
xbee_state xb_s = wait;

//has to run at lower rate than xbee callback
void PIT0_CALLBACK()
{
    //first message keeps car from shutting down on startup
    if(first_message == 1)
    {
        //xbee interrupt didn't occur to set deadman
        if(deadman == 0)
        {
            //shutdown XBee
            uc_uart_mask_int();
            CAR_XBEE_sleep();

            //set status LEDs
            CAR_LED_set_color(car_led_0, car_led_ylo);
            CAR_LED_set_color(car_led_1, car_led_ylo);
            CAR_LED_set_color(car_led_2, car_led_ylo);
            CAR_LED_set_color(car_led_3, car_led_ylo);
            CAR_LED_update();
            //shutdown motor
            CAR_MOTOR_shutdown();
        }
    }
}
```

```

        CAR_MOTOR_set_output_en(disable);
        CAR_MOTOR_update();
    }
    //xbee interrupt occurred. reset deadman
    else
    {
        deadman = 0;
    }
}

/*
 * Callback for switch 1
 */
void SW1_CALLBACK()
{
    //toggles motor output
    if(switch_1_push == 0){

        CAR_MOTOR_set_output_en(enable);
        CAR_MOTOR_update();
        switch_1_push = 1;
    }
    else
    {
        CAR_MOTOR_set_output_en(disable);
        CAR_MOTOR_update();
        switch_1_push = 0;
    }
}

/*
 * Callback for switch 3
 */
void SW3_CALLBACK()
{
    //currently unused
    if(switch_3_push == 0)
    {
        switch_3_push = 1;
    }
    else if(switch_3_push == 1)
    {
        switch_3_push = 0;
    }
}

/*
 * Callback for xbee
 */
void XBEE_CALLBACK()
{
    //switches lights when first message is received
    if(first_message == 0)
    {
        first_message = 1;
        CAR_LED_set_color(car_led_0, car_led_wht);
        CAR_LED_set_color(car_led_1, car_led_wht);
        CAR_LED_set_color(car_led_2, car_led_red);
        CAR_LED_set_color(car_led_3, car_led_red);
    }
}

```



```
        led_update = 1;
    }
    char rcv = uc_uart_get_data();

    //WAIT STATE
    if(xb_s == wait)
    {
        //if rcv is 0, move state to first
        if(rcv == 0x00)
        {
            xb_s = first;
        }
        //otherwise stay in wait state
    }

    //FIRST STATE
    else if(xb_s == first)
    {
        //if second 0 received move to second
        if(rcv == 0x00)
        {
            xb_s = second;
        }
        //else something went wrong and we wait again
        else
        {
            xb_s = wait;
        }
    }

    //SECOND STATE
    else if(xb_s == second)
    {
        //loop back to second state again if 0
        if(rcv == 0x00)
        {
            xb_s = second;
        }
        //we move into valid state
        else
        {
            xb_s = valid;
            //we set counter
            xbee_pkt_cnt = 0;
            //we move first valid character into raw rcv buffer
            raw_pkts[xbee_pkt_cnt] = rcv;
            xbee_pkt_cnt++;
        }
    }

    //VALID STATE
    else if(xb_s == valid)
    {
        raw_pkts[xbee_pkt_cnt] = rcv;
        xbee_pkt_cnt++;

        if(xbee_pkt_cnt == 10)
        {
            xb_s = wait;
        }
    }
}
```

```

char check = raw_pkts[0] ^ raw_pkts[1] ^ raw_pkts[2] ^
raw_pkts[3] ^ raw_pkts[4] ^ raw_pkts[5] ^ raw_pkts[6] ^
raw_pkts[7] ^ raw_pkts[8];

if(check == raw_pkts[9])
{
    //process these now so we keep TPM interrupt lockout as
    short as possible.
    unsigned int period0 = 0;
    unsigned int period1 = 0;
    unsigned int period2 = 0;
    unsigned int period3 = 0;

    period0 = (raw_pkts[2] << 8) | raw_pkts[1];
    period1 = (raw_pkts[4] << 8) | raw_pkts[3];
    period2 = (raw_pkts[6] << 8) | raw_pkts[5];
    period3 = (raw_pkts[8] << 8) | raw_pkts[7];

    int direction0 = raw_pkts[0] & 1;
    int direction1 = (raw_pkts[0] >> 1) & 1;
    int direction2 = (raw_pkts[0] >> 2) & 1;
    int direction3 = (raw_pkts[0] >> 3) & 1;

    //access control, because TPM uses these.
    //we can't modify them while TPM is using them.
    uc_tpm_mask_int();
    //update actuals
    CAR_MOTOR_set_t_period(motor_0, period0);
    CAR_MOTOR_set_t_period(motor_1, period1);
    CAR_MOTOR_set_t_period(motor_2, period2);
    CAR_MOTOR_set_t_period(motor_3, period3);
    CAR_MOTOR_set_t_direction(motor_0, direction0);
    CAR_MOTOR_set_t_direction(motor_1, direction1);
    CAR_MOTOR_set_t_direction(motor_2, direction2);
    CAR_MOTOR_set_t_direction(motor_3, direction3);
    //set update flags in tpm
    CAR_MOTOR_set_flags();

    uc_tpm_unmask_int();
    //sets deadman variable to check for active host
    deadman = 1;
}
}
}

/*
 * clock setup
 */
void clock_init()
{
    //set clock to blazing speed of 48Mhz.
    //Timers won't run at right frequency unless this is called first.
    unsigned int set_clock= (MCG_C4 & ~0xE0) | 0xA0;
    MCG_C4 = set_clock;
}

/*

```

```
* initializes modules
*/
void init()
{
    //mask all interrupts
    int_all_mask();

    //setup clock frequency
    clock_init();

    //setup spi
    uc_spi_init(spi_0); // CAR_LED and CAR_MOTOR dependencies

    //init car leds
    CAR_LED_init();

    //switches configured for interrupts
    uc_sw_init_int(switch_1, SW1_CALLBACK);
    uc_sw_init_int(switch_3, SW3_CALLBACK);

    CAR_MOTOR_init();

    CAR_XBEE_init();
    uc_uart_set_callback(XBEE_CALLBACK);

    //must be lower frequency than UART receive frequency
    pit_init(pit_0, priority_3, 1000000, PIT0_CALLBACK);
    pit_enable(pit_0);

    //enable interrupts
    int_all_unmask();

    //safe motor startup
    CAR_MOTOR_motor_startup();
    CAR_MOTOR_set_output_en(disable);
}

/*
 * main
 */
int main(void)
{
    //initialize hardware
    init();

    CAR_LED_set_color(car_led_0, car_led_trq);
    CAR_LED_set_color(car_led_1, car_led_trq);
    CAR_LED_set_color(car_led_2, car_led_trq);
    CAR_LED_set_color(car_led_3, car_led_trq);
    led_update = 1;

    while(1)
    {
        if(led_update == 1)
        {
            led_update = 0;
            CAR_LED_update();
        }
    }
}
```

```
    }  
    return 1;  
}
```

CAR_LED.h

```
/*
 * CAR_LED.h
 *
 * Created on: September 12, 2014
 * Author: Severin
 */

#ifndef CAR_LED_H
#define CAR_LED_H

#include "derivative.h"
#include "uc_spi.h"
#include "gpio.h"

typedef enum{
    car_led_off,
    car_led_blu,
    car_led_grn,
    car_led_trq,
    car_led_red,
    car_led_ppl,
    car_led_ylo, //#yolo!
    car_led_wht
} car_led_color_t;

typedef enum{
    car_led_0,
    car_led_1,
    car_led_2,
    car_led_3
} car_led_t;

void CAR_LED_init();
void CAR_LED_update();
void CAR_LED_set_color(car_led_t, car_led_color_t);

#endif
```

CAR_LED.c

```
/*
 * CAR_LED.c
 *
 * Created on: September 12, 2014
 * Author: Severin
 */

#include "CAR_LED.h"

//keeps track of current led states
static char car_led_0_1 = 0;
static char car_led_2_3 = 0;

/*
 * Initializes LED GPIOs.
 */
void CAR_LED_init()
{
    //CS/RCLK
    gpio_port_init(port_E, pin_19, alt_1, output);
    gpio_set_pin_state(port_E, pin_19, 1);
    //output enable
    gpio_port_init(port_B, pin_3, alt_1, output);
    gpio_set_pin_state(port_B, pin_3, 0);
    //clear and reset
    gpio_port_init(port_B, pin_2, alt_1, output);
    gpio_set_pin_state(port_B, pin_2, 0);
    gpio_set_pin_state(port_B, pin_2, 1);

    CAR_LED_set_color(car_led_0, car_led_off);
    CAR_LED_set_color(car_led_1, car_led_off);
    CAR_LED_set_color(car_led_2, car_led_off);
    CAR_LED_set_color(car_led_3, car_led_off);
    CAR_LED_update();
}

/*
 * Updates LEDs to whatever is currently in car_led_x_x when called.
 */
void CAR_LED_update()
{
    //CS/RCLK low
    gpio_set_pin_state(port_E, pin_19, 0);

    uc_spi_send(spi_0, car_led_2_3);
    uc_spi_send(spi_0, car_led_0_1);

    //CS/RCLK back up, yo
    gpio_set_pin_state(port_E, pin_19, 1);
}

/*
 * Sets specific car led color.
 */
void CAR_LED_set_color(car_led_t p_car_led, car_led_color_t p_car_led_color)
{
    //just clears and sets the relevant bits in the words sent to the 2 led shift
    regs
```

```
    if((p_car_led == car_led_0) || (p_car_led == car_led_1))
    {
        car_led_0_1 &= ~(0b111 << (2 + (p_car_led * 3)));
        car_led_0_1 |= (p_car_led_color) << (2 + (p_car_led * 3));
    }

    else
    {
        car_led_2_3 &= ~(0b111 << (2 + ((p_car_led - 2) * 3)));
        car_led_2_3 |= (p_car_led_color) << (2 + ((p_car_led - 2) * 3));
    }
}
```

CAR_MOTOR.h

```
/*
 * CAR_MOTOR.h
 *
 * Created on: October 13, 2014
 * Author: Severin
 */

#ifndef CAR_MOTOR_H
#define CAR_MOTOR_H

#include "derivative.h"
#include "uc_spi.h"
#include "uc_tpm.h"
#include "uc_dac.h"
#include "gpio.h"

typedef enum{
    CAR_MOTOR_dir_f,
    CAR_MOTOR_dir_b
}CAR_MOTOR_dir_t;

typedef enum{
    motor_0,
    motor_1,
    motor_2,
    motor_3
}CAR_MOTOR_motor_t;

//bit order md1, md2, md3
typedef enum{
    step_2    = 0b000,
    step_8    = 0b100,
    step_16   = 0b010,
    step_32   = 0b110,
    step_64   = 0b001,
    step_128  = 0b101,
    step_10   = 0b011,
    step_20   = 0b111
}CAR_MOTOR_step_size_t;

typedef enum{
    disable,
    enable
}CAR_MOTOR_state;

void CAR_MOTOR_init();
void CAR_MOTOR_manual_debug_init();
void CAR_MOTOR_update();
void CAR_MOTOR_set_MD(CAR_MOTOR_step_size_t);
void CAR_MOTOR_set_chip_en(CAR_MOTOR_state);
void CAR_MOTOR_set_output_en(CAR_MOTOR_state);
void CAR_MOTOR_set_rst(CAR_MOTOR_state);
void CAR_MOTOR_set_rst_cycle();
void CAR_MOTOR_set_current_limiter_en(CAR_MOTOR_state);
void CAR_MOTOR_CALLBACK_0();
void CAR_MOTOR_CALLBACK_1();
void CAR_MOTOR_CALLBACK_2();
```



```
void CAR_MOTOR_CALLBACK_3();  
void WAKEUP_CALLBACK();  
void CAR_MOTOR_motor_startup();  
void CAR_MOTOR_shutdown();  
void CAR_MOTOR_set_flags();  
void CAR_MOTOR_set_t_period(CAR_MOTOR_motor_t, unsigned int);  
void CAR_MOTOR_set_t_direction(CAR_MOTOR_motor_t, CAR_MOTOR_dir_t);  
unsigned int get_a_period(unsigned int, unsigned int);  
unsigned int get_d_period(unsigned int, unsigned int);  
#endif
```

CAR_MOTOR.c

```

/*
 * CAR_MOTOR.c
 *
 * Created on: October 13, 2014
 * Author: Severin
 */

#include "CAR_MOTOR.h"

/*MOTOR SETTING FIELDS
 *
 *
 * |ST          |MD1  |MD2  |MD3  |OE          |RST  |unused
 * |unused|
 * 1 active          check manual for MD 1 active          0 reset
 *
 *      1 normal
 *
 *
 */

#define A_TABLE_SZ 13
#define D_TABLE_SZ 10
#define A_D_CNT_MAX 20
#define FLIP_ZERO 5000
#define VEL_OFF 0xFFFF

//A bunch of globals to save states
static char car_motor = 0b00000000;

static volatile CAR_MOTOR_dir_t target_direction[4] = {1,1,1,1};
static volatile CAR_MOTOR_dir_t current_direction[4] = {1,1,1,1};
static volatile unsigned int current_period[4] = {VEL_OFF,VEL_OFF,VEL_OFF,VEL_OFF};
static volatile unsigned int target_period[4] = {VEL_OFF,VEL_OFF,VEL_OFF,VEL_OFF};
static volatile unsigned int previous_period[4] = {VEL_OFF,VEL_OFF,VEL_OFF,VEL_OFF};
static volatile int update_flag[4] = {0,0,0,0};

static int a_d_cnt[4] = {0, 0, 0, 0};

unsigned int get_a_period(unsigned int, unsigned int);
unsigned int get_d_period(unsigned int, unsigned int);
void CAR_MOTOR_set_direction(CAR_MOTOR_motor_t, CAR_MOTOR_dir_t);
unsigned int CAR_MOTOR_compute_period(CAR_MOTOR_motor_t, CAR_MOTOR_dir_t
c_direction, CAR_MOTOR_dir_t t_direction, unsigned int c_period, unsigned int
t_period);

//acceleration table
/*
65535 574.8127001
32767 572.286256
16383 567.2667428
8191 557.3611758
4095 538.0830159
2047 501.6431061
1023 436.9772255
511 336.9501875
255 218.4828964
127 121.3861211

```

```

63          62.26720821
31          30.91113864
15          14.98988865
*/

static const unsigned int a_bound[A_TABLE_SZ] = {15,      31,    63,    127,    255,
          511,    1023,    2047,    4095,    8191,    16383,      32767,    65535};
static const unsigned int a_table[A_TABLE_SZ] = {15,      31,    63,    125,    240,
          420,    624,    785,    885,    940,    969,    984,    992};

//deceleration table
/*
574    49602
572    31006
568    17680
560    9459
544    4848
512    2398
448    1126
320    462
64     65
2      2
*/

static const unsigned int d_bound[D_TABLE_SZ] = {10,      300,    647,    746,    873,
          936,    967,    982,    989,    992};
static const unsigned int d_table[D_TABLE_SZ] = {10,      330,    1113,    1683,    3671,
          7555,    14898,      27526,      45222,      65534};

/*
 * Initializes pins to control shift registers and configures "direction" pins as
GPIO
 * SPI needs to be initialized separately
 */
void CAR_MOTOR_init()
{
    //set up current-limiting digital - analog converter. still needs separate
call to set specific voltage
uc_dac_init();

    CAR_MOTOR_set_MD(step_128);
    CAR_MOTOR_update();

    //CS/RCLK
    gpio_port_init(port_E, pin_16, alt_1, output);
    gpio_set_pin_state(port_E, pin_16, 1);

    //output enable
    gpio_port_init(port_B, pin_1, alt_1, output);
    gpio_set_pin_state(port_B, pin_1, 0);

    //clear and reset
    gpio_port_init(port_B, pin_2, alt_1, output);
    gpio_set_pin_state(port_B, pin_0, 0);
    gpio_set_pin_state(port_B, pin_0, 1);

    //direction
    gpio_port_init(port_E, pin_0, alt_1, output);
    gpio_port_init(port_E, pin_1, alt_1, output);
    gpio_port_init(port_E, pin_2, alt_1, output);

```

```

    gpio_port_init(port_E, pin_3, alt_1, output);

    uc_tpm_init();
    uc_tpm_set_callback(tpm_chan_2, CAR_MOTOR_CALLBACK_0);
    uc_tpm_set_callback(tpm_chan_3, CAR_MOTOR_CALLBACK_1);
    uc_tpm_set_callback(tpm_chan_4, CAR_MOTOR_CALLBACK_2);
    uc_tpm_set_callback(tpm_chan_5, CAR_MOTOR_CALLBACK_3);
    uc_tpm_set_callback(tpm_chan_1, WAKEUP_CALLBACK);

    CAR_MOTOR_set_direction(motor_0, target_direction[motor_0]);
    CAR_MOTOR_set_direction(motor_1, target_direction[motor_1]);
    CAR_MOTOR_set_direction(motor_2, target_direction[motor_2]);
    CAR_MOTOR_set_direction(motor_3, target_direction[motor_3]);
}

/*
 * updates shift register outputs to whatever was set in global setting variable
 */
void CAR_MOTOR_update()
{
    //CS/RCLK low
    gpio_set_pin_state(port_E, pin_16, 0);

    //send four times to update all motors
    uc_spi_send(spi_0, car_motor);
    uc_spi_send(spi_0, car_motor);
    uc_spi_send(spi_0, car_motor);
    uc_spi_send(spi_0, car_motor);

    //CS/RCLK back up
    gpio_set_pin_state(port_E, pin_16, 1);
}

/*
 * sets MD on the LV8727
 */
void CAR_MOTOR_set_MD(CAR_MOTOR_step_size_t p_step_size)
{
    car_motor &= ~(7 << 4);
    car_motor |= p_step_size << 4;
}

/*
 * enables/disables the LV8727
 */
void CAR_MOTOR_set_chip_en(CAR_MOTOR_state p_state)
{
    car_motor &= ~(1 << 7);
    car_motor |= p_state << 7;
}

/*
 * enables/disables LV8727 output
 */
void CAR_MOTOR_set_output_en(CAR_MOTOR_state p_state)
{
    car_motor &= ~(1 << 3);
    car_motor |= p_state << 3;
}

```

```

/*
 * sets the reset on the LV8727
 */
void CAR_MOTOR_set_rst(CAR_MOTOR_state p_state)
{
    car_motor &= ~(1 << 2);
    car_motor |= p_state << 2;
}

/*
 * performs a reset cycle on LV8727 automatically
 */
void CAR_MOTOR_set_rst_cycle()
{
    CAR_MOTOR_set_rst(disable);
    CAR_MOTOR_update();
    CAR_MOTOR_set_rst(enable);
    CAR_MOTOR_update();
}

/*
 * enables enables current limiter pin for LV8727 current limiting
 */
void CAR_MOTOR_set_current_limiter_en(CAR_MOTOR_state p_state)
{
    if(p_state)
    {
        /*
            dac param      voltage (V)          output current (A)
            0                0.000805664         0.000994647
            200              0.161938477         0.199924045
            400              0.323071289         0.398853443
            600              0.484204102         0.597782841
            800              0.645336914         0.79671224
            1000             0.806469727         0.995641638
            1200             0.967602539         1.194571036
            1400             1.128735352         1.393500434
            1600             1.289868164         1.592429832
        */

        //needs tuning
        uc_dac_set_output(1100);
    }
    else
    {
        uc_dac_set_output(0);
    }
}

/*
 * get acceleration period based on old period and target
 */
unsigned int get_a_period(unsigned int c_period, unsigned int t_period)
{
    int i = 1;
    //loop through entire table. exit early if case matched and value was set.
    while(i <= A_TABLE_SZ)
    {
        if(c_period < a_bound[i])

```

```

        {
            unsigned int base_period = a_table[i-1];
            unsigned int period_adjust = c_period - a_bound[i-1];
            unsigned int numerator = period_adjust*(a_table[i] - a_table[i-1]);
            unsigned int offset = numerator/(a_bound[i] - a_bound[i-1]);
            return base_period + offset;
        }
        i++;
    }
    return a_table[A_TABLE_SZ-1];
}

/*
 * get deceleration period based on old period and target
 */
unsigned int get_d_period(unsigned int c_period, unsigned int t_period)
{
    int i = 1;
    //loop through entire table. exit early if case matched and value was set.
    while(i <= D_TABLE_SZ)
    {
        if(c_period < d_bound[i])
        {
            unsigned int base_period = d_table[i-1];
            unsigned int period_adjust = c_period - d_bound[i-1];
            unsigned int numerator = period_adjust*(d_table[i] - d_table[i-1]);

            unsigned int offset = numerator/(d_bound[i] - d_bound[i-1]);
            return base_period + offset;
        }
        i++;
    }
    return t_period;
}

/*
 * Should be called by TPM and performs motor 0 stepping
 */
void CAR_MOTOR_CALLBACK_0()
{
    //local copies. don't want the uart interrupt changing these while we're using them.
    //not sure if it matters, but it's safe.
    unsigned int c_period = current_period[0];
    unsigned int t_period = target_period[0];
    CAR_MOTOR_dir_t c_direction= current_direction[0];
    CAR_MOTOR_dir_t t_direction= target_direction[0];

    unsigned int final_period = CAR_MOTOR_compute_period(motor_0 , c_direction, t_direction, c_period, t_period);

    previous_period[0] = current_period[0];
    current_period[0] = final_period;

    uc_tpm_set_compare_val(tpm_chan_2, final_period);

    update_flag[0] = 0;
}

```

```
/*
 * Should be called by TPM and performs motor 1 stepping
 */
void CAR_MOTOR_CALLBACK_1()
{
    unsigned int c_period = current_period[1];
    unsigned int t_period = target_period[1];
    CAR_MOTOR_dir_t c_direction= current_direction[1];
    CAR_MOTOR_dir_t t_direction= target_direction[1];

    unsigned int final_period = CAR_MOTOR_compute_period(motor_1 , c_direction,
    t_direction, c_period, t_period);

    previous_period[1] =current_period[1];
    current_period[1] = final_period;

    uc_tpm_set_compare_val(tpm_chan_3, final_period);

    update_flag[1] = 0;
}

/*
 * Should be called by TPM and performs motor 2 stepping
 */
void CAR_MOTOR_CALLBACK_2()
{
    unsigned int c_period = current_period[2];
    unsigned int t_period = target_period[2];
    CAR_MOTOR_dir_t c_direction= current_direction[2];
    CAR_MOTOR_dir_t t_direction= target_direction[2];

    unsigned int final_period = CAR_MOTOR_compute_period(motor_2 , c_direction,
    t_direction, c_period, t_period);

    previous_period[2] =current_period[2];
    current_period[2] = final_period;

    uc_tpm_set_compare_val(tpm_chan_4, final_period);

    update_flag[2] = 0;
}

/*
 * Should be called by TPM and performs motor 3 stepping
 */
void CAR_MOTOR_CALLBACK_3()
{
    unsigned int c_period = current_period[3];
    unsigned int t_period = target_period[3];
    CAR_MOTOR_dir_t c_direction= current_direction[3];
    CAR_MOTOR_dir_t t_direction= target_direction[3];

    unsigned int final_period = CAR_MOTOR_compute_period(motor_3 , c_direction,
    t_direction, c_period, t_period);

    previous_period[3] =current_period[3];
    current_period[3] = final_period;

    uc_tpm_set_compare_val(tpm_chan_5, final_period);
```

```

        update_flag[3] = 0;
    }

    /*
     * called by channel 1 to update other channels preemptively. Triggered by UART
     */
    void WAKEUP_CALLBACK()
    {
        //set channel back to sleep mode
        uc_tpm_set_compare_val(tpm_chan_1, VEL_OFF);

        int i = 0;
        //check if motors need updating
        for(i = 0; i < 4; i++)
        {
            //check if the regular interrupt occurred after values were update.
            //If flag is 0, update already occurred on regular motor interrupt.
            if(update_flag[i] == 1)
            {
                //if current velocity is 0...
                if(current_period[i] == VEL_OFF)
                {
                    //...and we want to accelerate
                    if(target_period[i] != VEL_OFF)
                    {
                        //switch to new direction and update current
                        current_direction[i] = target_direction[i];
                        CAR_MOTOR_set_direction(i, target_direction[i]);
                        //interrupt ASAP
                        uc_tpm_pulse_asap(i+2);
                        //update current period to slowest period
                        current_period[i] = VEL_OFF-1;
                    }
                }
            }
            else
            {
                //check how much time is left
                unsigned int time_left = uc_tpm_time_left(i+2);

                if(time_left > 3)
                {
                    unsigned int final_period =
                        CAR_MOTOR_compute_period(i, current_direction[i],
                        target_direction[i], previous_period[i],
                        target_period[i]);

                    if(final_period == VEL_OFF)
                    {
                        uc_tpm_sleep(i+2);
                        current_period[i] = VEL_OFF;
                        previous_period[i] = VEL_OFF;
                    }
                    else
                    {
                        if(final_period > current_period[i])
                        {
                            //rewrite comparison value
                            uc_tpm_set_compare_val(i+2,
                                final_period - current_period[i]);
                        }
                    }
                }
            }
        }
    }
}

```



```

        else if(current_period[i] > final_period)
        {
            unsigned int time_difference =
                current_period[i] - final_period;
            //counter hasnt passed pulse point yet
            if(time_difference <= (time_left + 3))
            {
                uc_tpm_set_neg_compare_value(i+2, time_difference);
            }
            //counter passed pulse point...
            else
            {
                uc_tpm_pulse_asap(i+2);
            }
            current_period[i] = final_period;
        }
    }
    update_flag[i] = 0;
}

}

/*
 * sets the direction for a specific motor
 */
void CAR_MOTOR_set_direction(CAR_MOTOR_motor_t p_motor, CAR_MOTOR_dir_t p_dir)
{
    if(p_motor == motor_0 || p_motor == motor_2)
    {
        gpio_set_pin_state(port_E, p_motor, !p_dir);
    }
    else
    {
        gpio_set_pin_state(port_E, p_motor, p_dir);
    }
}

/*
 * sets new target period
 */
void CAR_MOTOR_set_t_period(CAR_MOTOR_motor_t p_motor, unsigned int p_target)
{
    target_period[p_motor]=p_target;
}

void CAR_MOTOR_set_t_direction(CAR_MOTOR_motor_t p_motor, CAR_MOTOR_dir_t p_dir)
{
    target_direction[p_motor]=p_dir;
}

/*
 * starts up LV8727s safely.
 */
void CAR_MOTOR_motor_startup()
{

```

```

    //motor startup sequence
    //set a md and turn off output. enable chip.
    //update motors

    CAR_MOTOR_set_MD(step_2);
    CAR_MOTOR_set_output_en(disable);
    CAR_MOTOR_set_chip_en(enable);
    CAR_MOTOR_update();

    //start current limiting analog value
    CAR_MOTOR_set_current_limiter_en(enable);

    //reset chip, since it just intialized
    //rst_cycle() auto-cycles and updates
    CAR_MOTOR_set_rst_cycle();
}

unsigned int CAR_MOTOR_compute_period(CAR_MOTOR_motor_t p_motor, CAR_MOTOR_dir_t
c_direction, CAR_MOTOR_dir_t t_direction, unsigned int c_period, unsigned int
t_period)
{
    unsigned int final_period = 0;
    //if same direction, simple a or d
    if(c_direction == t_direction)
    {
        //too fast, decelerate
        if(c_period < t_period)
        {
            //get decel value
            final_period = get_d_period(c_period, t_period);

            //if no progress is made, begin acceleration counter manual
            increment.
            if(final_period == c_period)
            {
                //end of acceleration counter reached, manually increment
                and reset counter for next iteration
                if(a_d_cnt[p_motor] == A_D_CNT_MAX)
                {
                    final_period++;
                    a_d_cnt[p_motor] = 0;
                }
                //increment counter
                else
                {
                    a_d_cnt[p_motor]++;
                }
            }
            //reset counter just in case.
            else
            {
                a_d_cnt[p_motor] = 0;
            }

            //check if target overshoot and correct
            if(final_period > t_period)
            {
                final_period = t_period;
            }
        }
    }
}

```

```
    }

    //too slow, accelerate
    else if(c_period > t_period)
    {
        final_period = get_a_period(c_period, t_period);

        //if no progress is made, begin acceleration counter manual
        increment.
        if(final_period == c_period)
        {
            //end of acceleration counter reached, manually increment
            and reset counter for next iteration
            if(a_d_cnt[p_motor] == A_D_CNT_MAX)
            {
                final_period--;
                a_d_cnt[p_motor] = 0;
            }
            //increment counter
            else
            {
                a_d_cnt[p_motor]++;
            }
        }
        //reset counter just in case.
        else
        {
            a_d_cnt[p_motor] = 0;
        }

        //check if target overshoot and correct
        if(final_period < t_period)
        {
            final_period = t_period;
        }
    }

    //target speed reached, continue
    else
    {
        final_period = t_period;
    }
}

//decelerate until direction flip
else
{
    //current period is already so low that we accelerate right away
    if(c_period >= FLIP_ZERO)
    {
        //flip direction
        current_direction[p_motor] = t_direction;
        CAR_MOTOR_set_direction(p_motor, t_direction);

        //get acceleration because we're speeding up now
        final_period = get_a_period(c_period, t_period);

        //check if target overshoot and correct for overshoot
        if(final_period < t_period)
        {
```

```

        final_period = t_period;
    }
}
//haven't slowed down enough yet
else
{
    //if we overshoot "0" speed via deceleration, set at "0" speed =
    2000 period
    final_period = get_d_period(c_period, 65535);

    if(final_period == c_period)
    {
        //end of acceleration counter reached,
        manually increment and reset counter
        for next iteration
        if(a_d_cnt[p_motor] == A_D_CNT_MAX)
        {
            final_period++;
            a_d_cnt[p_motor] = 0;
        }
        //increment counter
        else
        {
            a_d_cnt[p_motor]++;
        }
    }
    //reset counter just in case.
    else
    {
        a_d_cnt[p_motor] = 0;
    }

    if(final_period > FLIP_ZERO)
    {
        final_period = FLIP_ZERO;
        //flip direction
        current_direction[p_motor] = t_direction;
        CAR_MOTOR_set_direction(p_motor, t_direction);
    }
}

}

return final_period;
}

void CAR_MOTOR_shutdown()
{
    //stop all motors
    CAR_MOTOR_set_t_period(motor_0, VEL_OFF);
    CAR_MOTOR_set_t_period(motor_1, VEL_OFF);
    CAR_MOTOR_set_t_period(motor_2, VEL_OFF);
    CAR_MOTOR_set_t_period(motor_3, VEL_OFF);
    CAR_MOTOR_set_t_direction(motor_0, CAR_MOTOR_dir_b);
    CAR_MOTOR_set_t_direction(motor_1, CAR_MOTOR_dir_b);
    CAR_MOTOR_set_t_direction(motor_2, CAR_MOTOR_dir_b);
    CAR_MOTOR_set_t_direction(motor_3, CAR_MOTOR_dir_b);

    //spin until all motors are stopped
    int stop_flag = 0;

```

```
do{
    int i = 0;
    for(i = 0; i < 4; i++)
    {
        if(current_direction[i] != VEL_OFF)
        {
            stop_flag = 1;
        }
    }
}while(stop_flag == 1);
}

void CAR_MOTOR_set_flags()
{
    update_flag[0] = 1;
    update_flag[1] = 1;
    update_flag[2] = 1;
    update_flag[3] = 1;

    //set tpm to go off ASAP
    uc_tpm_pulse_asap(tpm_chan_1);
}
```

CAR_XBEE.h

```
/*
 * CAR_XBEE.h
 *
 * Created on: November 9, 2014
 * Author: Severin
 */

#ifndef CAR_XBEE_H
#define CAR_XBEE_H

#include "derivative.h"
#include "gpio.h"
#include "uc_uart.h"

void CAR_XBEE_init();
void CAR_XBEE_reset_cycle();
void CAR_XBEE_on();
void CAR_XBEE_sleep();

#endif
```

CAR_XBEE.c

```
/*
 * CAR_XBEE.c
 *
 * Created on: November 9, 2014
 * Author: Severin
 */

#include "CAR_XBEE.h"

/*
 * initializes pins for xbee
 */
void CAR_XBEE_init()
{
    //tx
    gpio_port_init(port_D, pin_7, alt_3, output);
    //rx
    gpio_port_init(port_D, pin_6, alt_3, input);

    //on. Initialized to sleep state
    gpio_port_init(port_E, pin_20, alt_1, output);
    gpio_set_pin_state(port_E, pin_20, 0);

    //reset cycle
    gpio_port_init(port_E, pin_21, alt_1, output);
    CAR_XBEE_reset_cycle();

    uc_uart_init();
}

/*
 * performs reset cycle on XBee
 */
void CAR_XBEE_reset_cycle()
{
    gpio_set_pin_state(port_E, pin_21, 1);
    gpio_set_pin_state(port_E, pin_21, 0);
    gpio_set_pin_state(port_E, pin_21, 1);
}

/*
 * wakes xbee
 */
void CAR_XBEE_on()
{
    gpio_set_pin_state(port_E, pin_20, 1);
}

/*
 * sleeps xbee
 */
void CAR_XBEE_sleep()
{
    gpio_set_pin_state(port_E, pin_20, 0);
}
```

gpio.h

```
/*
 * gpio.h
 *
 * Created on: May 7, 2014
 * Author: Severin
 */

#ifndef GPIO_H_
#define GPIO_H_

#include "derivative.h"
#include "int.h"

typedef enum{
    input,
    output
} dir_t;

typedef enum{
    port_A,
    port_B,
    port_C,
    port_D,
    port_E,
} port_t;

typedef enum{
    alt_0,
    alt_1,
    alt_2,
    alt_3,
    alt_4,
    alt_5,
    alt_6,
    alt_7
} alt_t;

typedef enum{
    pin_0,
    pin_1,
    pin_2,
    pin_3,
    pin_4,
    pin_5,
    pin_6,
    pin_7,
    pin_8,
    pin_9,
    pin_10,
    pin_11,
    pin_12,
    pin_13,
    pin_14,
    pin_15,
    pin_16,
    pin_17,
    pin_18,
    pin_19,
```



```
    pin_20,  
    pin_21,  
    pin_22,  
    pin_23,  
    pin_24,  
    pin_25,  
    pin_26,  
    pin_27,  
    pin_28,  
    pin_29,  
    pin_30,  
    pin_31,  
} pin_t;  
  
typedef enum{  
    trig_disable      = 0b0000,  
    trig_DMA_posedge  = 0b0001,  
    trig_DMA_negedge  = 0b0010,  
    trig_DMA_edge     = 0b0011,  
    trig_int_lo       = 0b1000,  
    trig_int_posedge  = 0b1001,  
    trig_int_negedge  = 0b1010,  
    trig_int_edge     = 0b1011,  
    trig_int_hi       = 0b1100  
} trig_t;  
  
void gpio_pin_set_dir(port_t, pin_t, dir_t);  
void gpio_pin_set_alt(port_t, pin_t, alt_t);  
void gpio_port_init(port_t, pin_t, alt_t, dir_t);  
  
int gpio_get_pin_state(port_t, pin_t);  
void gpio_set_pin_state(port_t, pin_t, int);  
void gpio_toggle_pin_state(port_t, pin_t);  
void gpio_enable_interrupt(port_t p_port, pin_t p_pin, trig_t p_trig, callback_t  
p_callback);  
  
#endif /* GPIO_H_ */
```

gpio.c

```

/*
 * gpio.c
 *
 * Created on: May 7, 2014
 * Author: Severin
 */

#include "gpio.h"
#include "uc_led.h"

#define PORTX_PCR_BASE          0x40049000

#define FGPIOX_PDOR_BASE        0xF80FF000
#define FGPIOX_PSOR_BASE        0xF80FF004
#define FGPIOX_PCOR_BASE        0xF80FF008
#define FGPIOX_PTOR_BASE        0xF80FF00C
#define FGPIOX_PDIR_BASE        0xF80FF010
#define FGPIOX_PDDR_BASE        0xF80FF014

static callback_t gpio_c_callback[32] = {0};
static callback_t gpio_d_callback[32] = {0};

/*
 * sets pin direction
 */
void gpio_pin_set_dir(port_t p_port, pin_t p_pin, dir_t p_dir)
{
    unsigned int * custom_PDDR = (unsigned int *) (FGPIOX_PDDR_BASE + (p_port *
0x40));

    *custom_PDDR &= ~(1 << p_pin);

    *custom_PDDR |= p_dir << p_pin;
}

/*
 * sets pin function (check reference manual for pin specific functions)
 */
void gpio_pin_set_alt(port_t p_port, pin_t p_pin, alt_t p_alt)
{
    unsigned int * custom_PCR = (unsigned int *) (PORTX_PCR_BASE + p_port * 0x1000 +
p_pin * 0x4);

    *custom_PCR &= ~0x700;

    *custom_PCR |= p_alt << 8;
}

/*
 * initializes a port/pin
 */
void gpio_port_init(port_t p_port, pin_t p_pin, alt_t p_alt, dir_t p_dir)
{
    SIM_SCGC5 |= 1 << (p_port + 9); // enable clock to port

    gpio_pin_set_alt(p_port, p_pin, p_alt);

    gpio_pin_set_dir(p_port, p_pin, p_dir);
}

```

```

}

/*
 * returns the state of a pin
 */
int gpio_get_pin_state(port_t p_port, pin_t p_pin)
{
    unsigned int * custom_PDIR = (unsigned int *) (FGPIOX_PDIR_BASE + (p_port *
0x40));

    return (*custom_PDIR >> p_pin) & 1;
}

/*
 * sets the state of a gpio pin
 */
void gpio_set_pin_state(port_t p_port, pin_t p_pin, int p_state)
{
    if(p_state)
    {
        unsigned int * custom_PSOR = (unsigned int *) (FGPIOX_PSOR_BASE + (p_port
* 0x40));

        *custom_PSOR = 1 << p_pin;
    }

    else
    {
        unsigned int * custom_PCOR = (unsigned int *) (FGPIOX_PCOR_BASE + (p_port
* 0x40));

        *custom_PCOR = 1 << p_pin;
    }
}

/*
 * toggles the state of a gpio pin
 */
void gpio_toggle_pin_state(port_t p_port, pin_t p_pin)
{
    unsigned int * custom_PTOR = (unsigned int *) (FGPIOX_PTOR_BASE + (p_port *
0x40));

    *custom_PTOR = 1 << p_pin;
}

/*
 * enables an interrupt on a pin. needs edge trigger and callback function specified
 */
void gpio_enable_interrupt(port_t p_port, pin_t p_pin, trig_t p_trig, callback_t
p_callback)
{
    unsigned int * custom_PCR = (unsigned int *) (PORTX_PCR_BASE + p_port *
0x1000 + p_pin * 0x4);

    if(p_port == port_C)
    {
        gpio_c_callback[p_pin] = p_callback;
    }
}

```

```

    }
    else if(p_port == port_D)
    {
        gpio_d_callback[p_pin] = p_callback;
    }
    //enable pin interrupt with trigger
    *custom_PCR &= ~PORT_PCR_IRQC_MASK;
    *custom_PCR |= PORT_PCR_IRQC(p_trig);

    //config priority
    int_init(INT_PORTC_PORTD, priority_1);
}

/*
 * interrupt handler for all pins on ports C and D
 */
void PORTCD_IRQHandler()
{
    int i = 0;

    //check all portc sources
    for(i = 0; i < 32; i++)
    {
        if((PORTC_PCR(i) & PORT_PCR_ISF_MASK) == PORT_PCR_ISF_MASK)
        {
            //clear interrupt
            PORTC_PCR(i) |= PORT_PCR_ISF_MASK;
            if(gpio_c_callback[i])
            {
                gpio_c_callback[i]();
            }
        }
    }

    //check all portd sources
    for(i = 0; i < 32; i++)
    {
        if((PORTD_PCR(i) & PORT_PCR_ISF_MASK) == PORT_PCR_ISF_MASK)
        {
            PORTD_PCR(i) |= PORT_PCR_ISF_MASK;
            if(gpio_d_callback[i])
            {
                gpio_d_callback[i]();
            }
        }
    }
}

```

int.h

```
/*
 * int.h
 *
 * Created on: May 10, 2014
 * Author: Severin
 */

#ifndef INT_H_
#define INT_H_

#include "derivative.h"

typedef void (*callback_t)();

typedef enum{
    priority_0,
    priority_1,
    priority_2,
    priority_3,
} priority_t;

void int_init(IRQInterruptIndex, priority_t);
void int_all_unmask();
void int_all_mask();
void int_unmask(IRQInterruptIndex);
void int_mask(IRQInterruptIndex);

#endif /* INT_H_ */
```

int.c

```
/*
 * int.c
 *
 * Created on: May 10, 2014
 * Author: Severin
 */

#include "int.h"

/*
 * set the priority for a specified interrupt
 */
void int_init(IRQInterruptIndex p_vector, priority_t p_priority)
{
    int IRQ = p_vector - 16;
    NVIC_IP(IRQ / 4) &= ~(3 << ((8 * (IRQ % 4)) + 6));
    NVIC_IP(IRQ / 4) |= (p_priority << ((8 * (IRQ % 4)) + 6));
}

/*
 * unmask all interrupts
 */
void int_all_unmask()
{
    NVIC_ISER = 0xFFFFFFFF;
}

/*
 * masks all interrupts
 */
void int_all_mask()
{
    NVIC_ICER = 0xFFFFFFFF;
}

/*
 * masks specified interrupt
 */
void int_mask(IRQInterruptIndex p_vector)
{
    int IRQ = p_vector - 16;
    NVIC_ICER = (1 << IRQ);
}

/*
 * unmask specified interrupt
 */
void int_unmask(IRQInterruptIndex p_vector)
{
    int IRQ = p_vector - 16;
    NVIC_ISER = (1 << IRQ);
}
```

uc_dac.h

```
/*
 * uc_dac.h
 *
 * Created on: June 8, 2013
 * Author: Severin
 */

#ifndef UC_DAC_H_
#define UC_DAC_H_

#include "derivative.h"
#include "gpio.h"

void uc_dac_init();
void uc_dac_set_output(int);

#endif /* UC_DAC_H_ */
```

uc_dac.c

```
/*
 * uc_dac.c
 *
 * Created on: June 8, 2013
 * Author: Severin
 */

#include "uc_dac.h"

/*
 * initialize d to a converter
 */
void uc_dac_init()
{
    gpio_port_init(port_E, pin_30, alt_0, output);
    SIM_SCGC6 |= 1 << 31;
    uc_dac_set_output(0);

    DAC0_C0 |= 1 << 6;
    DAC0_C0 |= 1 << 7;
}

/*
 * sets output of dac
 */
void uc_dac_set_output(int p_numerator)
{
    //p_numerator is out of 4095. VOUT = VIN * (1+p_numerator)/4096
    //MAX VOUT = VIN
    //MIN VOUT = (1/4096) * VIN
    DAC0_DAT0L = p_numerator;
    DAC0_DAT0H = (p_numerator >> 8);
}
```


uc_led.h

```
/*
 * uc_led.h
 *
 * Created on: Dec 24, 2013
 * Author: Severin
 */

#ifndef UC_LED_H_
#define UC_LED_H_

#include "derivative.h"
#include "gpio.h"

typedef enum{
    led_green,
    led_red
} led_t;

void uc_led_all_init();
void uc_led_init(led_t);
void uc_led_on(led_t);
void uc_led_off(led_t);
void uc_led_toggle(led_t);

#endif /* UC_LED_H_ */
```

uc_led.c

```
/*
 * uc_led.c
 *
 * Created on: Dec 24, 2013
 * Author: Severin
 */

#include "uc_led.h"

/*
 * initializes both on board LEDs
 */
void uc_led_all_init()
{
    uc_led_init(Led_red);
    uc_led_init(Led_green);
}

/*
 * initializes individual LEDs
 */
void uc_led_init(led_t p_led)
{
    if(p_led == Led_green)
    {
        gpio_port_init(port_D, pin_5, alt_1, output);
        gpio_set_pin_state(port_D, pin_5, 1);
    }
    else if(p_led == Led_red)
    {
        gpio_port_init(port_E, pin_29, alt_1, output);
        gpio_set_pin_state(port_E, pin_29, 1);
    }
}

/*
 * turns specific LED on
 */
void uc_led_on(led_t p_led)
{
    if(p_led == Led_green)
    {
        gpio_set_pin_state(port_D, pin_5, 0);
    }
    else if(p_led == Led_red)
    {
        gpio_set_pin_state(port_E, pin_29, 0);
    }
}

/*
 * turns specific LED off
 */
void uc_led_off(led_t p_led)
{
    if(p_led == Led_green)
    {
        gpio_set_pin_state(port_D, pin_5, 1);
    }
}
```

```
    }
    else if(p_led == Led_red)
    {
        gpio_set_pin_state(port_E, pin_29, 1);
    }
}

/*
 * toggles specific LED
 */
void uc_led_toggle(led_t p_led)
{
    if(p_led == Led_green)
    {
        gpio_toggle_pin_state(port_D, pin_5);
    }
    else if(p_led == Led_red)
    {
        gpio_toggle_pin_state(port_E, pin_29);
    }
}
```

uc_lptmr.h

```
/*
 * uc_lptmr.h
 *
 * Created on: Dec 25, 2013
 * Author: Severin
 */

#ifndef UC_LPTMR_H_
#define UC_LPTMR_H_

#include "derivative.h"

void uc_lptmr_init();
void uc_lptmr_delay(int p_msec);

#endif /* UC_LPTMR_H_ */
```

uc_lptmr.c

```
/*
 * uc_lptmr.c
 *
 * Created on: Dec 25, 2013
 * Author: Severin
 */

#include "uc_lptmr.h"

/*
 * initializes the low power timer
 */
void uc_lptmr_init()
{
    //Timer is still wonky. Only works consistently with a hard power cycle
    //I lost the working code, but it involved writing to another register to
    reset something
    //I didn't need this for the project, so fixing this wasn't a priority
    SIM_SCGC5 |= 1;
    SIM_SOPT1 = ~(3 << 18);
    LPTMR0_PSR = 0x5;
}

/*
 * busy delay in milliseconds
 */
void uc_lptmr_delay(int p_msec)
{
    LPTMR0_CMR = p_msec;
    LPTMR0_CSR |= 1;
    while((LPTMR0_CSR & 0x80) == 0){} //spin while compare isn't equal
    LPTMR0_CSR &= ~1;
}
```

uc_pit.h

```
/*
 * uc_pit.h
 *
 * Created on: Jan 18, 2014
 * Author: Severin
 */

#ifndef UC_PIT_H_
#define UC_PIT_H_

#include "derivative.h"
#include "int.h"

typedef enum{
    pit_0,
    pit_1
} pit_t;

void pit_enable(pit_t);
void pit_disable(pit_t);
void pit_init(pit_t, priority_t, int, callback_t);
void pit_set_callback(pit_t, callback_t);

#endif /* UC_PIT_H_ */
```

uc_pit.c

```

/*
 * uc_pit.c
 *
 * Created on: May 11, 2014
 * Author: Severin
 */

#include "uc_pit.h"
#include "uc_led.h"

static callback_t pit0_callback = 0;
static callback_t pit1_callback = 0;

/*
 * enables pit, duh
 */
void pit_enable(pit_t p_timer)
{
    PIT_TCTRL(p_timer) |= 1;
}

/*
 * disables pit
 */
void pit_disable(pit_t p_timer)
{
    PIT_TCTRL(p_timer) &= ~(1);
}

/*
 * initializes pit with microsecond period
 */
void pit_init(pit_t p_timer, priority_t p_priority, int p_us_period, callback_t p_callback)
{
    //PIT clock gating
    SIM_SCGC6 |= 1<<23;

    //enable PIT
    PIT_MCR = 0;

    int_init(INT_PIT, p_priority);

    pit_set_callback(p_timer, p_callback);

    PIT_LDVAL(p_timer) = 24 * p_us_period;
    PIT_TCTRL(p_timer) |= 0b10;
}

/*
 * sets pit callback
 */
void pit_set_callback(pit_t p_timer, callback_t p_callback)
{
    if(p_timer == pit_0)
    {
        pit0_callback = p_callback;
    }
}

```

```
        if(p_timer == pit_1)
        {
            pit1_callback = p_callback;
        }
    }

/*
 * interrupt handler
 */
void PIT_IRQHandler()
{
    if(PIT_TFLG0 != 0)
    {
        PIT_TFLG0 = 1;
        if(pit0_callback)
        {
            pit0_callback();
        }
    }

    if(PIT_TFLG1 != 0)
    {
        PIT_TFLG1 = 1;
        if(pit1_callback)
        {
            pit1_callback();
        }
    }
}
```


uc_spi.h

```
/*
 * uc_spi.h
 *
 * Created on: June 15, 2014
 * Author: Severin
 */

#ifndef UC_SPI_H_
#define UC_SPI_H_

#include "derivative.h"
#include "gpio.h"

typedef enum{
    spi_0,
    spi_1
} spi_t;

void uc_spi_init(spi_t);
void uc_spi_send(spi_t, char);

#endif /* UC_SPI_H_ */
```

uc_spi.c

```

/*
 * uc_spi.c
 *
 * Created on: June 15, 2014
 * Author: Severin
 */

#include "uc_spi.h"

/*
 * initializes spi. (untested for spi1)
 */
void uc_spi_init(spi_t p_spi)
{
    switch(p_spi)
    {
        case spi_0:
            SIM_SCGC4 |= 1 << 22;
            SPI0_BR = 2 << 4;
            SPI0_BR |= 8;

            SPI0_C1 = SPI_C1_MSTR_MASK;

            gpio_port_init(port_E, pin_17, alt_2, output);
            gpio_port_init(port_E, pin_18, alt_2, output);

            SPI0_C1 |= 1 << 6;
            break;

        case spi_1:
            SIM_SCGC4 |= 1 << 23;
            SPI1_BR = 2 << 4;
            SPI1_BR |= 8;

            SPI1_C1 = SPI_C1_MSTR_MASK | SPI_C1_SS0E_MASK;

            gpio_port_init(port_D, pin_5, alt_2, output);
            gpio_port_init(port_D, pin_6, alt_2, output);
            gpio_port_init(port_D, pin_7, alt_2, output);
            gpio_port_init(port_D, pin_4, alt_1, output);
            gpio_set_pin_state(port_D, pin_4, 0);
            SPI1_C1 |= 1 << 6;
            break;

        default:
            break;
    }
}

/*
 * sends a byte (untested for spi1)
 */
void uc_spi_send(spi_t p_spi, char p_char)
{
    int i;
    switch(p_spi)
    {
        case spi_0:

```

```
        //waiting for spi to be available
        while((0b100000 & SPI0_S) == 0)
        {}
        //send char
        SPI0_DL = p_char;

        //busy wait loop. 2000 seems to be about right.
        //they should really have some kind of transmit complete flag.

        for(i = 0; i<2100; i++)
        {
            asm("nop");
        }
        break;

    case spi_1:
        gpio_set_pin_state(port_D, pin_4, 1);
        while(!(0b100000 & SPI1_S))
        {}
        SPI1_DL = p_char;

        for(i =0; i<2100; i++)
        {
            asm("nop");
        }
        gpio_set_pin_state(port_D, pin_4, 0);
        break;

    default:
        break;
}
}
```

uc_sw.h

```
/*
 * uc_sw.h
 *
 * Created on: Aug 28, 2014
 * Author: Severin
 */

#ifndef UC_SW_H_
#define UC_SW_H_

#include "derivative.h"
#include "int.h"

typedef enum{
    switch_1,
    switch_3
} switch_t;

void uc_sw_init(switch_t);
void uc_sw_init_int(switch_t, callback_t);
int uc_sw_poll_switch1();
int uc_sw_poll_switch3();

#endif
```

UC_SW.C

```
/*
 * uc_sw.c
 *
 * Created on: Aug 28, 2014
 * Author: Severin
 */

#include "uc_sw.h"
#include "gpio.h"

/*
 * configures switch for polling
 */
void uc_sw_init(switch_t p_switch)
{
    if(p_switch == switch_1)
    {
        //setup GPIO for relevant pin
        gpio_port_init(port_C, pin_3, alt_1, input);

        //enable pullup res
        PORTC_PCR3 |= 1 << 1;
    }

    if(p_switch == switch_3)
    {
        gpio_port_init(port_C, pin_12, alt_1, input);
        PORTC_PCR12 |= 1 << 1;
    }
}

/*
 * configures switch as interrupt
 */
void uc_sw_init_int(switch_t p_switch, callback_t p_callback)
{
    if(p_switch == switch_1)
    {
        //setup GPIO for relevant pin
        gpio_port_init(port_C, pin_3, alt_1, input);

        //enable pullup res
        PORTC_PCR3 |= 1 << 1;
        gpio_enable_interrupt(port_C, pin_3, trig_int_negedge, p_callback);
    }

    if(p_switch == switch_3)
    {
        gpio_port_init(port_C, pin_12, alt_1, input);
        PORTC_PCR12 |= 1 << 1;

        gpio_enable_interrupt(port_C, pin_12, trig_int_negedge, p_callback);
    }
}

/*
```

```
    * returns state of switch1
    */
int uc_sw_poll_switch1()
{
    return gpio_get_pin_state(port_C, pin_3);
}

/*
 * returns stae of switch3
 */
int uc_sw_poll_switch3()
{
    return gpio_get_pin_state(port_C, pin_12);
}
```

uc_tpm.h

```
/*
 * uc_tpm.h
 *
 * Created on: June 15, 2014
 * Author: Severin
 */

#ifndef UC_TPM_H_
#define UC_TPM_H_

#include "derivative.h"
#include "gpio.h"
#include "int.h"

typedef enum{
    tpm_chan_0,
    tpm_chan_1,
    tpm_chan_2,
    tpm_chan_3,
    tpm_chan_4,
    tpm_chan_5
} tpm_chan_t;

void uc_tpm_init();
void uc_tpm_set_callback(tpm_chan_t, callback_t);
void uc_tpm_set_compare_val(tpm_chan_t, int);
void uc_tpm_unmask_int();
void uc_tpm_mask_int();
unsigned int uc_tpm_time_left(tpm_chan_t);
void uc_tpm_pulse_asap(tpm_chan_t p_tpm_chan);
void uc_tpm_set_neg_compare_value(tpm_chan_t p_tpm_chan, unsigned neg_value);
void uc_tpm_sleep(tpm_chan_t p_tpm_chan);

#endif /* UC_TPM_H_ */
```

uc_tpm.c

```
/*
 * uc_tpm.c
 *
 * Created on: June 15, 2014
 * Author: Severin
 */

#include "uc_tpm.h"

#define TPM_MOD_VAL 0xFFFE
#define TPM_MOD_VAL_OFF 0xFFFF
#define MIN_TIME 50

static callback_t tpm_callback[6] = {0,0,0,0,0,0};

/*
 * initializes tpm
 * hardcoded for motor/output compare function
 */
void uc_tpm_init()
{
    //motors are driven off D2-D4 outputs with TPM function (alt4)
    gpio_port_init(port_D, pin_2, alt_4, 1);
    gpio_port_init(port_D, pin_3, alt_4, 1);
    gpio_port_init(port_D, pin_4, alt_4, 1);
    gpio_port_init(port_D, pin_5, alt_4, 1);

    //internal clock selected for FLL and MCGIRCLK active
    MCG_C1 |= MCG_C1_IRCLKEN_MASK | MCG_C1_IREFS_MASK;
    //choose fast internal reference clock
    MCG_C2 |= MCG_C2_IRCS_MASK;

    //enable clock to tpm
    SIM_SCGC6 |= SIM_SCGC6_TPM0_MASK;
    //choose MCGIRCLK as TPM clk source
    SIM_SOPT2 |= SIM_SOPT2_TPMSRC(3);

    //CMOD set increment on each incoming clock pulse
    //divide incoming clock by prescaler. 0b000 = 1, 0b111 = 128
    TPM0_SC = TPM_SC_CM0D(1) | TPM_SC_PS(6);

    // set to 0xFFFE to leave non-triggering value at 0xFFFF
    TPM0_MOD = TPM_MOD_VAL;

    //setup interrupt
    int_init(INT_TPM0, priority_1);

    int n;
    for(n = 1; n < 6; n++)
    {
        //configd for output compare
        TPM0_CnSC(n) = TPM_CnSC_MSB_MASK | TPM_CnSC_MSA_MASK |
            TPM_CnSC_ELSA_MASK;

        //initialized to "safe" off value at 0xFFFF
        //TPM0_CnV(0) = TPM_MOD_VAL_OFF;
        //temporary
        TPM0_CnV(n) = 1;
    }
}
```



```

        //ACTIVATE INTERRUPT. 3. 2. 1. G000000000000000000000000
        TPM0_CnSC(n) |= TPM_CnSC_CHIE_MASK;
    }
}

/*
 * sets the callback pointer
 */
void uc_tpm_set_callback(tpm_chan_t p_tpm_chan, callback_t p_callback)
{
    tpm_callback[p_tpm_chan] = p_callback;
}

/*
 * sets the new compare value
 */
void uc_tpm_set_compare_val(tpm_chan_t p_tpm_chan, int p_value)
{
    //set compare value to "OFF" state
    if(p_value == TPM_MOD_VAL_OFF)
    {
        TPM0_CnV(p_tpm_chan) = TPM_MOD_VAL_OFF;
    }
    else
    {
        //if counter is currently in OFF state, wakeup channel
        if(TPM0_CnV(p_tpm_chan) == TPM_MOD_VAL_OFF)
        {
            TPM0_CnV(p_tpm_chan) = ((TPM0_CNT+2)%TPM_MOD_VAL_OFF);
        }
        else
        {
            TPM0_CnV(p_tpm_chan) =
            ((TPM0_CnV(p_tpm_chan)+p_value)%TPM_MOD_VAL_OFF);
        }
    }
}

/*
 * Interrupt handler
 */
void TPM0_IRQHandler()
{
    //grab counter value to lock counter value when interrupt occurred
    int n;
    // loop through all used tpm channels
    for(n = 5; n > 0; n--)
        //check if interrupt occurred in that channel
        if((TPM0_CnSC(n) & TPM_CnSC_CHF_MASK) == TPM_CnSC_CHF_MASK)
        {
            TPM0_CnSC(n) |= TPM_CnSC_CHF_MASK;

            //callback time!
            if(tpm_callback[n])
            {
                tpm_callback[n]();
            }
        }
}

```

```
}

/*
 * masks tpm interrupt
 */
void uc_tpm_mask_int()
{
    int_mask(INT_TPM0);
}

/*
 * unmask tpm interrupt
 */
void uc_tpm_unmask_int()
{
    int_unmask(INT_TPM0);
}

/*
 * returns the time left until the next pulse occurs
 */
unsigned int uc_tpm_time_left(tpm_chan_t p_tpm_chan)
{
    //get counter and written value
    unsigned int counter = TPM0_CNT;
    unsigned int c_val = TPM0_CnV(p_tpm_chan);

    if(c_val >= counter)
    {
        return c_val-counter;
    }

    else
    {
        unsigned int temp = TPM_MOD_VAL - counter;
        return c_val + temp;
    }
}

/*
 * sets a channel to pulse as soon as possible
 */
void uc_tpm_pulse_asap(tpm_chan_t p_tpm_chan)
{
    TPM0_CnV(p_tpm_chan) = (TPM0_CNT + 3)%TPM_MOD_VAL_OFF;
}

/*
 * sets a channel to the off value, puts it to sleep
 */
void uc_tpm_sleep(tpm_chan_t p_tpm_chan)
{
    TPM0_CnV(p_tpm_chan) = TPM_MOD_VAL_OFF;
}

/*
 * instead of adding to the current compare value, this goes backwards
 * so it subtracts from the current value and the pulse will occur earlier
 */
void uc_tpm_set_neg_compare_value(tpm_chan_t p_tpm_chan, unsigned neg_value)
```

```
{
    //passing 0 if neg value is greater
    if(neg_value > TPM0_CnV(p_tpm_chan))
    {
        neg_value = TPM0_CnV(p_tpm_chan) - neg_value;
        TPM0_CnV(p_tpm_chan) =  TPM_MOD_VAL - neg_value;
    }
    else
    {
        TPM0_CnV(p_tpm_chan) =  TPM0_CnV(p_tpm_chan) - neg_value;
    }
}
```

uc_uart.h

```
/*
 * uc_uart.h
 *
 * Created on: June 21, 2014
 * Author: Severin
 */

#ifndef UC_UART_H_
#define UC_UART_H_

#include "derivative.h"
#include "int.h"

void uc_uart_init();
unsigned char uc_uart_get_data();
void uc_uart_set_callback(callback_t);
void uc_uart_mask_int();
void uc_uart_unmask_int();

#endif /* UC_UART_H_ */
```

uc_uart.c

```

/*
 * uc_uart.c
 *
 * Created on: June 21, 2014
 * Author: Severin
 */

#include "uc_uart.h"

static callback_t uart0_callback = 0;

/*
 * inits UART0
 */
void uc_uart_init()
{
    //clock setup stuff
    MCG_C1 |= 0b11 << 1;
    MCG_C2 |= 1;

    SIM_SCGC4 |= 1 << 10;

    //baud rate. MCGIRCLK=2000000, OSR default = 15.
    //MCGIRCLK/((OSR+1) (13)) = 9600
    UART0_BDL = (unsigned char) 13;
    //clear relevant bits in BDH
    UART0_BDH &= 0xE0;
    UART0_BDH |= (unsigned char)((~0xE0)&(13 >> 8));

    //enable receiver
    UART0_C2 |= 1 << 2;

    //setting receiver interrupt enable
    UART0_C2 |= 1 << 5;

    //configuring interrupt.
    int_init(INT_UART0, priority_2);

    SIM_SOPT2 |= SIM_SOPT2_UART0SRC(3);
}

/*
 * sets interrupt callback for UART0
 */
void uc_uart_set_callback(callback_t p_callback)
{
    uart0_callback = p_callback;
}

/*
 * reads and returns incoming data for UART0
 */
unsigned char uc_uart_get_data()
{
    return UART0_D;
}

/*
 * interrupt handler

```

```
*/
void UART0_IRQHandler()
{
    if(uart0_callback)
    {
        uart0_callback();
    }
    else
    {
        //data lost
        uc_uart_get_data();
    }
}

/*
 * disables uart interrupt
 */
void uc_uart_mask_int()
{
    int_mask(INT_UART0);
}

/*
 * enables uart interrupt
 */
void uc_uart_unmask_int()
{
    int_unmask(INT_UART0);
}
```