FrostAV Development

Stage 1: Base

Updated November 6, 2019

Contents

1 System Abstraction

1.1 Abstract System Operation Description

The Frost Autonomous Vehicle consists of a vehicle and a system for controlling the vehicle. The vehicle provides an interface mainly to move it and to steer it. Bridge Controllers handle the peripherals that make up the vehicle interface. Bridge Controllers cannot operate unless they have instructions. These instructions come from a wired-bus. Different types of modules can send instructions to the wired-bus. The most important modules are the Networker and each individual Sensor System. Sensor Systems provide feedback for the Bridge Controllers to act upon. Furthermore, the Networker transfers information between the wired-bus and the wireless-bus. The wireless-bus allows for communication with devices that are not on the vehicle itself. Overall, the abstract system provides a map for how information is received and directed towards the vehicle's interaction with the environment.

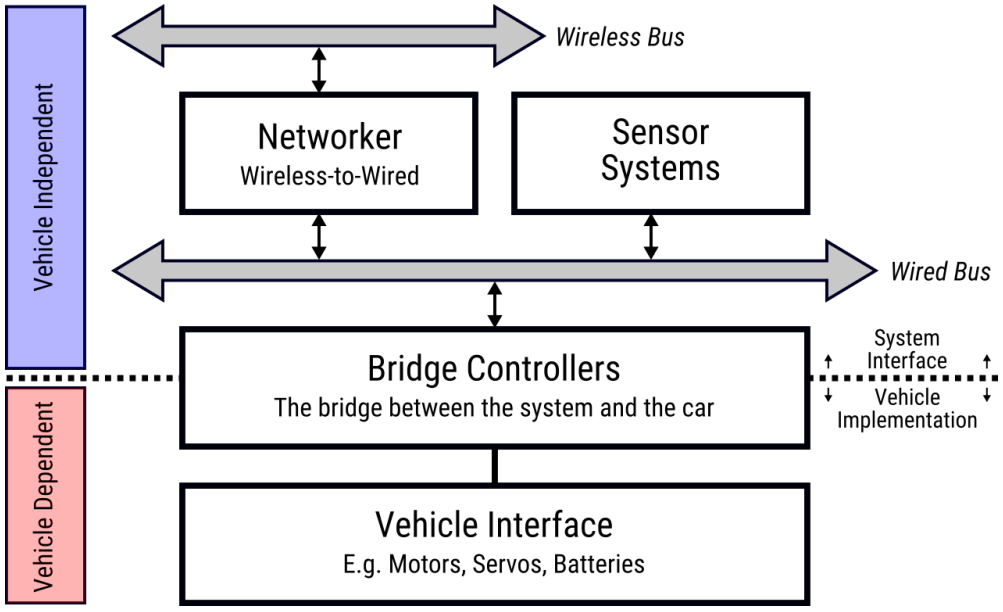


Figure 1: Fundamental module abstraction. Consists of a wired-bus for inter-communication between modules on the Frost vehicle, and a wireless-bus for communication with a server or an ssh client. Bridge Controllers act as the coupling between the vehicle and the modules that do not depend on the vehicle interface.

1.2 Bridge Controllers

"Bridge" refers to a software design pattern from the Gang of Four, which intends to "decouple an abstraction from its implementation so that the two can vary independently." As such, the Bridge Controllers will have software that implements the Bridge design pattern, so that implementation code (for a specific vehicle) is decoupled from the system interface.

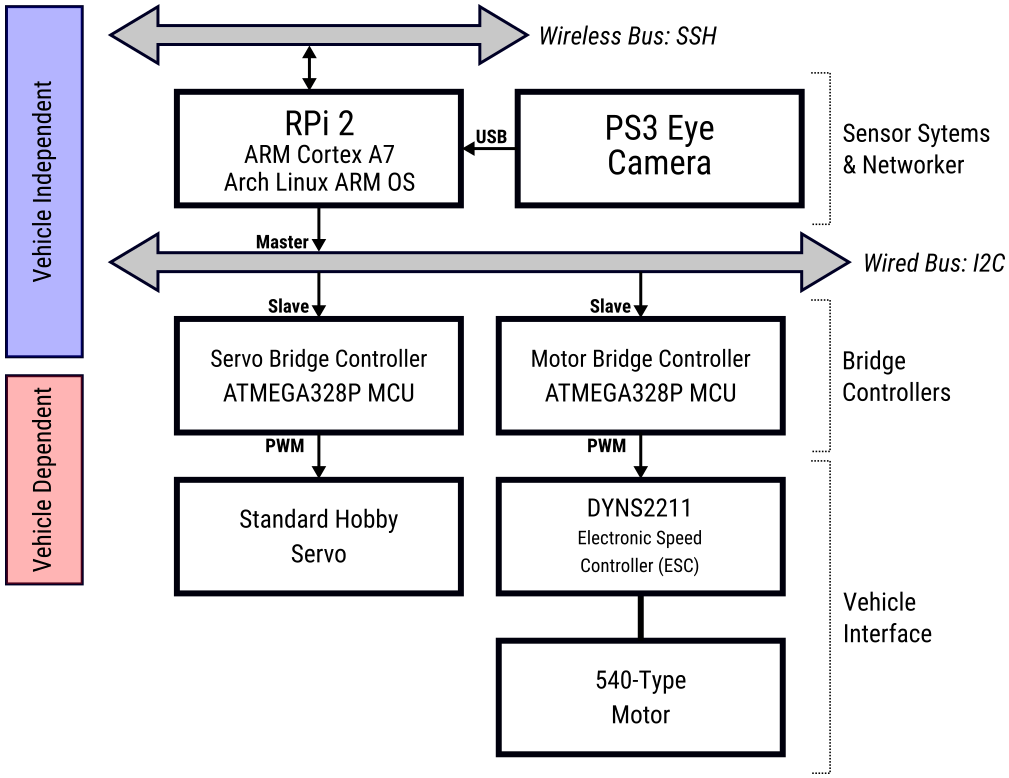
1.3 Networker

The Networker is reponsible for transfering information between the wireless-bus and the wired-bus.

1.4 Sensor Systems

Various Sensor Systems may be added to the wired-bus. An individual Sensor System should receive information from a sensor, process the information, and send the result to the wired-bus.

2 System Plan



2.1 System Operation Description

The PS3 Eye Camera provides visual information about the environment. This information will be processed with the OpenCV library on a Raspberry Pi 2 with an Arch ARM operating system. Processed camera information will then be piped through I^2C to the Bridge Controllers. The Servo Bridge Controller has a PWM output which directly drives the steering servo. The Motor Bridge Controller has a PWM output that is handled by the electronic speed controller (ESC), which drives the motor.

Overall, this system operates in a feedback loop commonly illustrated similar to the figure below.

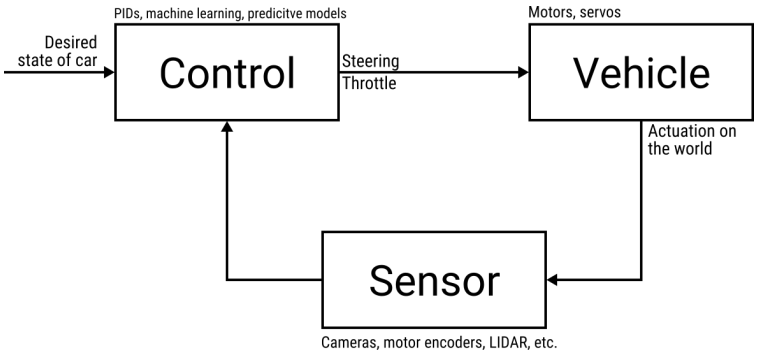


Figure 2: A simple feedback loop for an autonomous car.

2.2 Feedback Loop Flow of Information

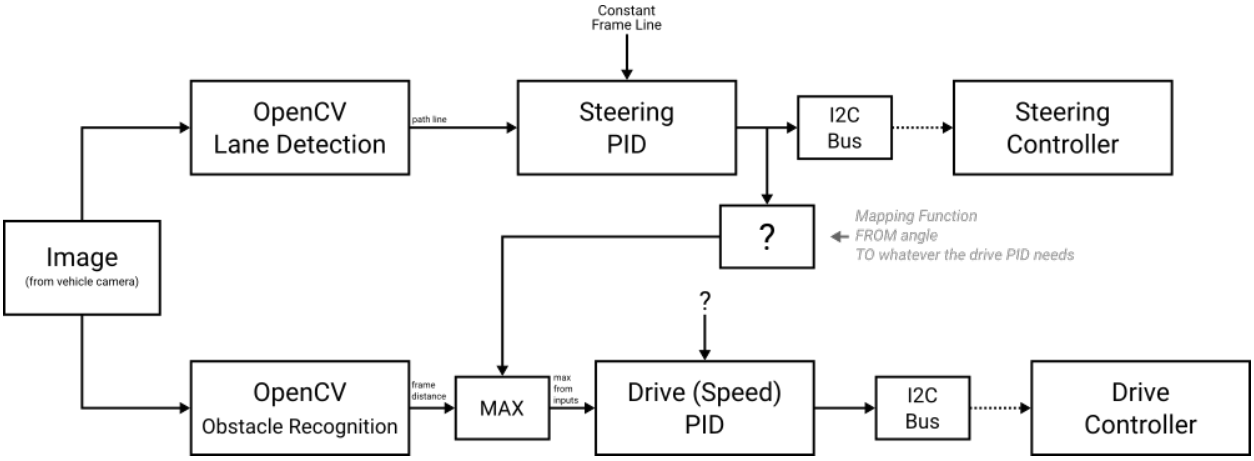


Figure 3: A concrete illustration of the feedback loop in figure 2

2.3 System Assumptions

The assumption made in this development stage is that the motor speed and servo steering angle can be controlled individually, such that motor speed and steering angle do not require direct knowledge of each other. Hence, the arrows from the wired-bus to the Servo and Motor Bridge Controllers are unidirectional. This assumption is meant to simplify the I^2C wired-bus configuration - since bi-directional communication on this bus would require arbitration.

2.4 Bus Communication

JSON will be used as the format for data for both the wireless and wired buses. JSON provides easy to understand data structure packets that can be serialized and sent as a stream. Using JSON sacrifices speed for readability and extendability of the system.

2.4.1 Wireless-Bus

In this stage the wireless-bus should allow for an SSH client to connect to it. As such, JSON packets that would normally be sent to the wireless-bus, will instead be stored locally in the system, but will be callable from an SSH client.

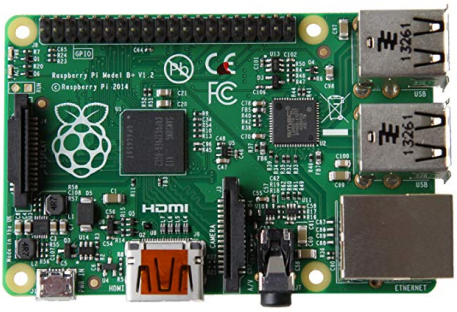
2.4.2 Wired-Bus

For the wired-bus, I^2C will be used. I^2C addressing will be done manually for each slave connected to the bus. Data will be sent from master to slaves via the JSON format.

2.5 Block Function Description

2.5.1 Raspberry Pi 2 (RPi2)

The RPi2 has an Arch Linux ARM operating system. It is responsible for processing camera information from the PS3 Eye. It is also responsible for being the Networker, which transfers information between the wired and wireless buses.



2.5.2 Steering Servo & Motor Bridge Controllers (ATMEGA328P)

Processes commands from the wired-bus to control the vehicle's steering servo and motor.

2.5.3 PS3 Eye Camera

Provides images over USB, for environmental perception.

2.5.4 Electronic Speed Controller (DYNS2211)

Takes in a PWM signal and outputs an amplified and directional PWM wave to the vehicle's motor.

3 Controlling the Steering Servo with a PID

3.1 Concept

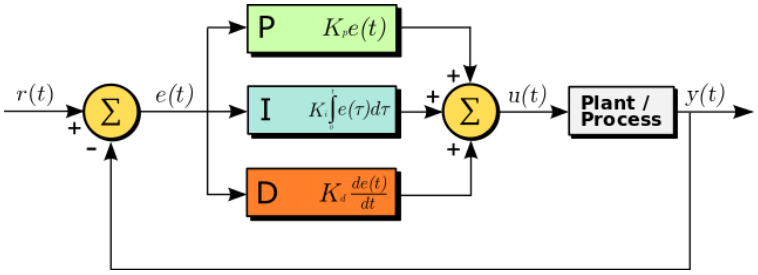


Figure 4: A block diagram of a PID controller in a feedback loop. r(t) is the desired process value or setpoint (SP), and y(t) is the measured process value (PV). *Arturo Urquizo, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0*

A PID controller can be used to correct the steering of an autonomous vehicle. Given a vehicle with the task of following a line or path, the vehicle must be able to detect the path as well as where it is in relation to the path. The difference can be taken from these two pieces of information to form the cross-track error. The cross-track error will be sent as the input to the PID — in figure 4, the cross-track error is denoted $e(t)$.

The PID has three control terms that are summed together to get a control variable output. Proportional control allows the vehicle to steer harder when it is further from the path. Derivative control induces a resistance to the pull from the proportional control; this brings the car smoothly back to the path in a timely manner while reducing the chance that the vehicle will overshoot the

path due to the proportional control. Finally, the integral control corrects for external effects such as wind and terrain.

Each of these controls rely on a properly tuned PID. To tune a PID, each control term has a gain which must be adjusted until the system is stable. Such details will not be covered in this document.

3.2 Parts

Part	Purpose
Arduino Uno (AVR attmega328p)	Steering Vehicle Interface Controller that interfaces between the Pi and the vehicle servo.
Standard Hobby Servo	The same type of servo that will be to steer the vehicle.

3.3 Experimental Setup

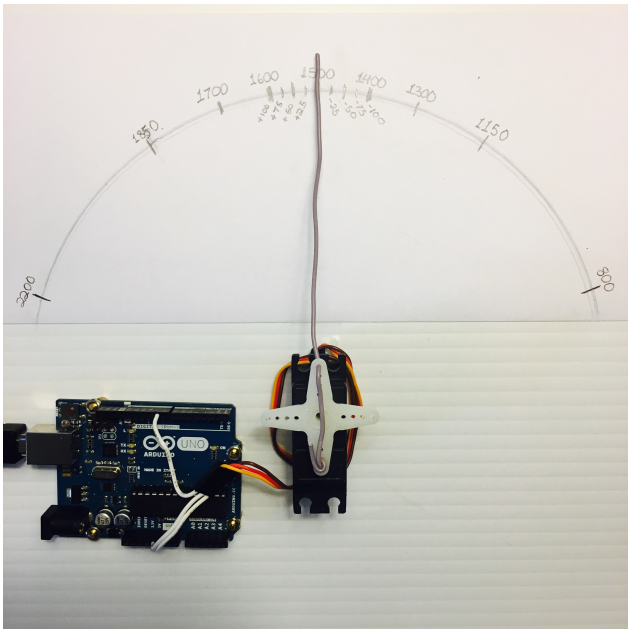


Figure 5: The Arduino provides microsecond PWM values from 800 to 2200μs to the servo. The servo reacts to an error value via a software PID controller. Since there is nothing in this setup that generates a true error value, error values must either be simulated or retrieved from a feedback sensor (such as a camera). The values along the circumference are microsecond values that correspond to the servo position; they help verify the accuracy of the PID.

3.4 Software Design

All code in this section was compiled using the avr-g++ tool, and using C++17. Test code was compiled with g++ as opposed to avr-g++.

3.4.1 USART to Command the Servo from a Terminal Emulator

In order to talk to the servo, we need to set up the USART on the AVR microcontroller. We will only show the code for a basic USART setup. Since the USART is only being used for experimental purposes only, we will not explain the code. For reference of how the code was used in our experiment, see the `main.c` file in the code appendix section 5.1.1.

`usart.hpp`:

```
#ifndef USART_HPP
#define USART_HPP

#include <avr/io.h>
#include <stdint.h>

namespace usart {
    void setup(uint32_t clockFrequency, uint16_t baud) {
        uint8_t ubrr = clockFrequency/16/baud - 1;
        UBRR0H = ubrr >> 8;
        UBRR0L = ubrr;

        //Enable Transmitter & Receiver
        UCSR0B = 1 << RXEN0 | 1 << TXEN0;
        //Frame Format: 8 data, 2 stop
        UCSR0C = 1 << USBS0 | 3 << UCSZ00;

    }

    void print(char c) {
        while (!(UCSR0A & 1<<UDRE0));
        UDR0 = c;
    }

    void print(char* string) {
        while (*string) print(*string++);
    }

    char getChar() {
        while (!(UCSR0A & 1<<RXC0));
        return UDR0;
    }
}

#endif
```

3.4.2 Servo Control

To control the position of the servo, the attmega328p has a 16-bit timer. Once we set up the timer, we can provide it a pulse duration between the accepted values of the servo (typically 1-2ms).

The code for the servo is at the register level. Ideally, one should use a hardware abstraction layer (HAL) to avoid writing production code at a register level. In our `main.cpp` file, we have the following to set up our servo.

```
#include <avr/io.h>
```

```
static constexpr uint32_t hertzToCycles(uint16_t hertz) {
    return clockFrequency/prescaler/hertz;
}

static void setupServoPwm() {
        DDRB |= 1 << PINB1; //Set pin 9 on arduino to output

        TCCR1A |=
        1 << WGM11 | //PWM Mode 14 (1/3)
        1 << COM1A0 | //Inverting Mode (1/2)
        1 << COM1A1; //Inverting Mode (2/2)

        TCCR1B |=
        1 << WGM12 | //PWM Mode 14 (2/3)
        1 << WGM13 | //PWM Mode 14 (3/3)
        1 << CS11; //Prescaler: 8

    //50Hz PWM to cycles for servo
        ICR1 = hertzToCycles(pwmFrequency)-1;
}
```

The servo uses the MCU's 16-bit Timer/Counter1 with PWM. It is connected to pin 9 on the Arduino Uno (port B, pin 1).

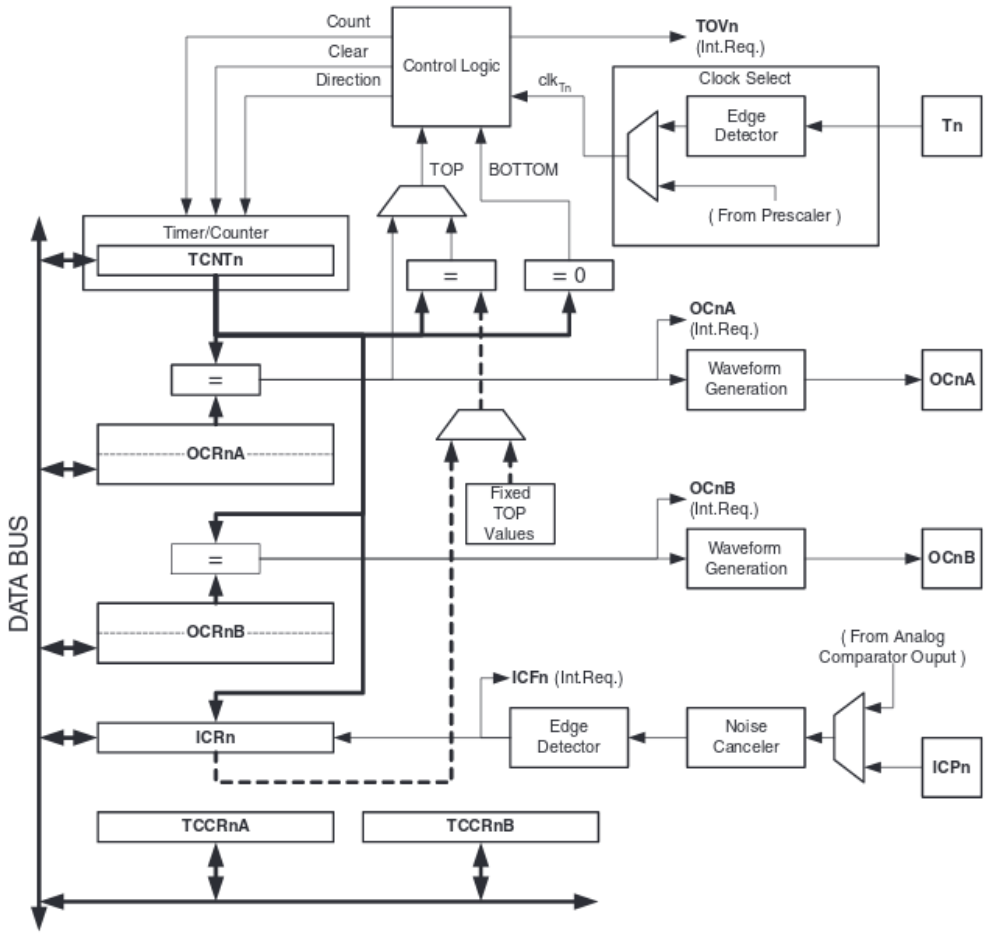


Figure 6: ICRn stores the number of cycles needed for a 50Hz PWM signal for the servo. OCRnA stores the number of cycles until the counter reaches the TOP. OCRnA is used to change the pulse width of the PWM signal.

We also need to specify that we want to use Fast PWM Mode (resets counter when it reaches the TOP) and set ICR1 as the TOP. This corresponds to `mode 14`.

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Figure 7: Shows the `WGM` flags needed for mode 14.

We do this on the register level by setting the `WGM` flags from figure 7 in the `TCCRnA` and `TCCRnB` registers (as shown in the code above).

Since the Arduino Uno uses a $16MHz$ clock, we will need a prescaler to divide the frequency such

that,

$$\frac{16MHz}{50Hz \cdot \texttt{prescaler}} < 2^{16} - 1.$$

$50MHz$ is the needed frequency for the PWM signal for the servo and $2^{16} - 1$ is the maximum value that will fit in the `OCRnA` register (the size of an `int` on the attmega328p). If we set the prescaler to 8, the inequality above is true. We can do that by setting the `TCCRnB` register using the figure below. Setting this register for a prescaler of 8, looks like `TCCR1B |= 1 << CS11` (as shown in the code above).

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Figure 8: We are using a prescaler of 8, which corresponds to `CS1 = 0b010`.

Finally, we can set the timer to inverted mode, meaning the pulse will occur at the end of the PWM period, as opposed to the beginning of the period. This is trivial for our application, but we still must choose either inverting or non-inverting. To set the timer to inverted we set the corresponding flags in the `TCCRnA` register, as `TCCR1A |= 1 << COM1A0 | 1 << COM1A1` (as shown in the code above).

COM1A1/COM1B1	COM1A0/COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 14 or 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode)

Figure 9: We make sure to set the `COM1A1` and `COM1A0` flags in the `TCCRnA` register.

3.4.3 Clamping

One common problem is providing a servo with a PWM value outside the valid range of the servo. For example, our servo has a maximum acceptable pulse width of $2200\mu s$. If we provide our servo with a pulse width $2300\mu s$, it could damage the servo. We will discuss a simple method for clamping the pulse widths provided to the servo.

The `Clamp` class looks as follows:

```
#ifndef CLAMP_HPP
#define CLAMP_HPP

#include <stdint.h>

struct Bounds {
    int16_t lower;
    int16_t upper;
};

class Clamp {
    Bounds bounds;

public:
    constexpr static Clamp makeFromBounds(Bounds bounds) {
        return Clamp(bounds);
    }

    constexpr Clamp(Bounds bounds): bounds{bounds} {}

    int16_t clamp(int16_t value) {
        return (value < bounds.lower) ? bounds.lower :
            (value > bounds.upper) ? bounds.upper:
            value;
    }

    Clamp() {}
};

#endif
```

The class takes in a `Bounds` and uses the `clamp` member function to output a value between the upper and lower bounds. Since we are using this code on a microcontroller, we want an object of type `Clamp` to be initialized at compile-time. To do this we declare the constructor, and the static factory method, as `constexpr`. To ensure compile-time initialization, we must pass a constant expression or rvalue to either the static factory method or the constructor.

Here is a unit test suite, using CxxTest, for the `Clamp` class:

```
#include <cxxtest/TestSuite.h>
#include "Clamp.hpp"

class TestClamp: public CxxTest::TestSuite {
    Clamp clamp;
    Bounds bounds;

public:
    void setUp() {
        bounds = {
            .lower = 10,
            .upper = 20
        };

        clamp = Clamp::makeFromBounds(bounds);
```

```
    }

    void test_inputAboveUpper_clampsToUpper() {
        int16_t expected = bounds.upper;
        int16_t actual = clamp.clamp(25);
        TS_ASSERT_EQUALS(actual, expected);
    }

    void test_inputBelowLower_clampsToLower() {
        int16_t expected = bounds.lower;
        int16_t actual = clamp.clamp(5);
        TS_ASSERT_EQUALS(actual, expected);
    }

    void test_inputBetweenBounds_noClamp() {
        int16_t expected = 15;
        int16_t actual = clamp.clamp(15);
        TS_ASSERT_EQUALS(actual, expected);
    }
};
```

Each `test_` function in the test suite demonstrates the functionality of the `Clamp` class. It also includes the way we recommend instantiating the `Clamp` class. That is, by using the static factory method as opposed to the constructor.

```
clamp = Clamp::makeFromBounds({
        .lower = 10,
        .upper = 20 });
```

This is purely for readability, so the reader can understand that a `Clamp` object requires a `Bounds`.

Again, after instantiating the `Clamp` class, you can use it as,

```
int16_t clampedValue = clamp.clamp(5); // Returns 10 since the lower bound is 10.
```

3.4.4 PID

As discussed in section 3.1, a PID will provide reactive control to the servo from error feedback. In our system, the camera detects lanes. The further the car is from being centered between those lanes, the greater the error that is input into our steering PID. We've encapsulated the algorithm for a PID in a class.

```
#ifndef PID_HPP
#define PID_HPP

#include <stdint.h>

struct Pid {
    struct Component {
        int16_t proportional;
        int16_t integral;
        int16_t derivative;
    };
```

```
private:
    Component gain;
    Component error;
    int16_t scale;

public:
    int16_t updateError(int16_t error);

    constexpr static Pid makeFromGain(Component gain) {
        return Pid(gain);
    }

    constexpr explicit Pid(Component gain):
        gain{gain}, error{}, scale{1} {}


    constexpr static Pid makeFromScaledGain(int16_t scale, Component gain) {
        return Pid(scale, gain);
    }

    constexpr Pid(int16_t scale, Component gain):
        gain{gain}, error{}, scale{scale} {}

    Pid() {}
};

#endif
```

The PID takes in a structure of gains (proportional, integral, and derivative). Then, when `updateError` is called, it will return the control variable output of the PID.

Since we may have gains that are decimal values, and assuming we don't want to do floating point arithmetic on a microcontroller, we've included a scale variable which divides the final output of the PID. This means if we provide a gain value of `1` (for one of the control gains) and a scale of `10`, then we equivalently have a gain value of `1/10`. To get this behavior, we use the `makeFromScaleGain` to instantiate the `Pid` class, instead of `makeFromGain`.

The definition of the `updateError` function is

```
int16_t Pid::updateError(int16_t newError) {
    error.integral += newError;
    error.derivative = newError - error.proportional;
    error.proportional = newError;

    return (gain.proportional*error.proportional +
            gain.integral*error.integral +
            gain.derivative*error.derivative) / scale ;
}
```

Notice the result is divided by scale, as discussed above. Furthermore, we've provided a unit test that demonstrates how to use this class.

```
#include <cxxtest/TestSuite.h>
#include "Pid.hpp"

class TestPid: public CxxTest::TestSuite {
```

```
public:
    void test_errorInput_returnsCorrectedOutput() {
        Pid pid = Pid::makeFromGain({
                .proportional = 1,
                .integral = 2,
                .derivative = 3 });

        int16_t actual = pid.updateError(2);
        int16_t expected = 12;

        TS_ASSERT_EQUALS(actual, expected);
    }

    void test_scaledGain() {
        int16_t scale = 100;

        Pid pid = Pid::makeFromScaledGain(scale, {
                .proportional = 100,
                .integral = 200,
                .derivative = 300 });

        int16_t actual = pid.updateError(2);
        int16_t expected = 12;

        TS_ASSERT_EQUALS(actual, expected);
    }
};
```

Ideally, the user would call `updateError` iteratively though.

3.4.5 Terminal-Emulator Results

The `main.cpp` code in the code appendix section 5.1.1 depcits the general flow of the program. Ultimately, the Arduino asks for an initial position for the servo (in microseconds), and then the servo will go to that position. After a short delay, the PID will begin updating its control variable output in a for loop. That control variable output is directly used to position the servo. For debugging purposes, the Arduino also sends the PID control variable output over USART, so we can see the values the PID is giving.

The figure below illustrates an example where I send the servo to $800\mu s$, and then the PID corrects the servo's position to $1500\mu s$.

```
>> 800
GOT: 800
1640
1332
1561
1455
1521
1487
1506
1497
1501
1500
1500
1500
1500
1500
1500
>>
```

Figure 10: Note that this PID is not tuned at all, and that the error we are using is not based on a real feedback error from a sensor. This demo is purely to show that the system is prepared for real feedback error from the OpenCV system.

4 Future

4.1 I2C Arbitration

It is expected that modules communicating on the wired bus will need to communicate with each-other. In this case, there would be multiple masters on a single i2c bus. To implement this, arbitration would be necessary so that masters on the bus do not interrupt each other.

5 Appendix: Code

5.1 Steering PID

5.1.1 main.cpp

```cpp
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include "Pid.hpp"
#include "Clamp.hpp"
#include "String.hpp"
#include "usart.hpp"

constexpr uint8_t prescaler = 8;
constexpr uint32_t clockFrequency = F_CPU;
constexpr uint8_t pwmFrequency = 50;
constexpr uint32_t baud = 9600;

static constexpr uint32_t microsToCycles(uint16_t micros) {
    constexpr uint32_t unitConversion = 1E6;
    return (clockFrequency/unitConversion/prescaler) * micros;
```

```
}

static constexpr uint32_t hertzToCycles(uint16_t hertz) {
    return clockFrequency/prescaler/hertz;
}

static void setupServoPwm() {
    DDRB |= 1 << PINB1; //Set pin 9 on arduino to output

    TCCR1A |=
        1 << WGM11 | //PWM Mode 14 (1/3)
        1 << COM1A0 | //Inverting Mode (1/2)
        1 << COM1A1; //Inverting Mode (2/2)

    TCCR1B |=
        1 << WGM12 | //PWM Mode 14 (2/3)
        1 << WGM13 | //PWM Mode 14 (3/3)
        1 << CS11; //Prescaler: 8

    //50Hz PWM to cycles for servo
    ICR1 = hertzToCycles(pwmFrequency)-1;
}

int main() {
    usart::setup(clockFrequency, baud);
    setupServoPwm();

    Clamp steeringClamp = Clamp::makeFromBounds({
            .lower = 800,
            .upper = 2200 });
    Pid steeringPid = Pid::makeFromScaledGain(10, {
            .proportional = 10,
            .integral = 0,
            .derivative = 2 });

    String<10> message;
    char currentChar;
    int16_t idealServoMicros = 1500;
    while(1) {
        currentChar = usart::getChar();

        if (currentChar != '\n') message.append(currentChar);
        else {
            usart::print("GOT: ");
            usart::print(message);
            usart::print('\n');

            int16_t initialServoMicros = atoi(message);
            int16_t servoMicros = steeringClamp.clamp(initialServoMicros);
            OCR1A = ICR1 - microsToCycles(initialServoMicros);
            _delay_ms(1000);

            int16_t error = idealServoMicros - servoMicros;
            for (uint32_t i = 0; i < 15; ++i) {
                servoMicros = steeringPid.updateError(error) + servoMicros;

                String<10> buffer;
                itoa(servoMicros, buffer, 10);
```

```
                usart::print(buffer); usart::print('\n');

                servoMicros = steeringClamp.clamp(servoMicros);
                OCR1A = ICR1 - microsToCycles(servoMicros);
                _delay_ms(100);

                //TODO replace with actual feedback error
                error = idealServoMicros - servoMicros;
            }

            message.clear();
            usart::print(">> ");
        }
    }
}
```

5.1.2 usart.hpp

```
#ifndef USART_HPP
#define USART_HPP

#include <avr/io.h>
#include <stdint.h>

namespace usart {
    void setup(uint32_t clockFrequency, uint16_t baud) {
        uint8_t ubrr = clockFrequency/16/baud - 1;
        UBRR0H = ubrr >> 8;
        UBRR0L = ubrr;

        //Enable Transmitter & Receiver
        UCSR0B = 1 << RXEN0 | 1 << TXEN0;
        //Frame Format: 8 data, 2 stop
        UCSR0C = 1 << USBS0 | 3 << UCSZ00;

    }

    void print(char c) {
        while (!(UCSR0A & 1<<UDRE0));
        UDR0 = c;
    }

    void print(char* string) {
        while (*string) print(*string++);
    }

    char getChar() {
        while (!(UCSR0A & 1<<RXC0));
        return UDR0;
    }
}

#endif
```

5.1.3 Pid.hpp

```
#ifndef PID_HPP
#define PID_HPP

#include <stdint.h>

struct Pid {
    struct Component {
        int16_t proportional;
        int16_t integral;
        int16_t derivative;
    };

private:
    Component gain;
    Component error;
    int16_t scale;

public:
    int16_t updateError(int16_t error);

    constexpr static Pid makeFromGain(Component gain) {
        return Pid(gain);
    }

    constexpr explicit Pid(Component gain):
        gain{gain}, error{}, scale{1} {}


    constexpr static Pid makeFromScaledGain(int16_t scale, Component gain) {
        return Pid(scale, gain);
    }

    constexpr Pid(int16_t scale, Component gain):
        gain{gain}, error{}, scale{scale} {}

    Pid() {}
};

#endif
```

5.1.4 Pid.cpp

```
#include "Pid.hpp"

int16_t Pid::updateError(int16_t newError) {
    error.integral += newError;
    error.derivative = newError - error.proportional;
    error.proportional = newError;

    return (gain.proportional*error.proportional +
            gain.integral*error.integral +
            gain.derivative*error.derivative) / scale ;
}
```

5.1.5 Clamp.hpp

```
#ifndef CLAMP_HPP
#define CLAMP_HPP

#include <stdint.h>

struct Bounds {
    int16_t lower;
    int16_t upper;
};

class Clamp {
    Bounds bounds;

public:
    constexpr static Clamp makeFromBounds(Bounds bounds) {
        return Clamp(bounds);
    }

    constexpr Clamp(Bounds bounds): bounds{bounds} {}

    int16_t clamp(int16_t value) {
        return (value < bounds.lower) ? bounds.lower :
            (value > bounds.upper) ? bounds.upper:
            value;
    }

    Clamp() {}
};

#endif
```

5.1.6 String.hpp

```
#ifndef STRING_HPP
#define STRING_HPP

#include <stdint.h>

template<int16_t capacity>
class String {
    int16_t size;
    char string[capacity];

public:
    void append(char c) {
        if (size < capacity) string[size++] = c;
    }

    String(): size{0}, string{0} {}

    void clear() {
        while (size != 0) string[--size] = 0;
    }
```

```
    operator char*() { return string; }
    operator const char*() { return string; }
};

#endif
```

5.1.7 Makefile

```
BIN=sweep

CPP = $(BIN).cpp $(wildcard *.cpp)
OBJ = $(CPP:%.cpp=%.o)
DEP = $(OBJ:%.o=%.d)

CROSS_COMPILE=avr-
CXX=$(CROSS_COMPILE)g++
OBJCOPY=$(CROSS_COMPILE)objcopy
CXXFLAGS=-Os -Wall -DF_CPU=$(F_CPU) -mmcu=atmega328p -std=gnu++17

F_CPU=16000000UL
PORT=/dev/ttyACM0
DEVICE=ATMEGA328P
PROGRAMMER=arduino
FLASH_BAUD=115200
COM_BAUD=9600

TESTS = TestPid.hpp TestClamp.hpp
TEST_SOURCES= Pid.cpp
TEST_RUNNER = runner



$(BIN).hex: $(BIN).elf
    ${OBJCOPY} -O ihex -R .eeprom $< $@

$(BIN).elf: $(OBJ)
    $(CXX) $(CXXFLAGS) -o $@ $^

-include $(DEP)

%.o: %.cpp
     $(CXX) $(CXXFLAGS) -MMD -c $< -o $@

flash: $(BIN).hex
    avrdude -c $(PROGRAMMER) -P $(PORT) -p $(DEVICE) -b $(FLASH_BAUD) -U flash:w:$<

test: $(TESTS)
    cxxtestgen --error-printer -o $(TEST_RUNNER).cpp $(TESTS)
    g++ -o $(TEST_RUNNER) -I$CXXTEST $(TEST_RUNNER).cpp $(TEST_SOURCES)
    ./$(TEST_RUNNER)

com:
    picocom -b $(COM_BAUD) $(PORT) -p 2


clean:
```

```
    rm -f ${BIN}.elf ${BIN}.hex $(OBJ) $(TEST_RUNNER)* $(DEP)

.PHONY: clean
```