# Contents

# WindRider Firmware Guide

This document is a comprehensive guide to installation and development of Wind Rider Firmware. The project is based on STM32F103C8T6 microcontroller and STM's official peripheral library. Download this tutorial as pdf.

## Getting started

Before we dive into the description of the code flow there are some prerequisites we assume you are familiar with and thus will not be explained in this document:

- C++ standard library programming experience.
- STM32 Peripheral library. STM libraries are very well documented inside their header files. Feel free to navigate to link folder and take a closer look.
- Unix command line.
- CMake. Used as a build system. Not very crucial to modify the code.

### *Dependencies*

Tools required to build and debug this firmware.

- arm-none-eabi – Official ARM compiler.
- CMake – Build system.
- An SWD probe – Used for flashing and debugging. Can be either st-link or jlink.

## Build and Flash

### *Build*

Instructions for building the firmware.

#### *Set Toolchain Path*

Specify ARM toolchain location. Set TOOLCHIN_PATH in CMakeLists.txt to point to your toolchain's bin directory

```
SET(TOOLCHIN_PATH "/opt/gcc-arm-none-eabi-9-2019-q4-major/bin")
```

#### *Run CMake*

Navigate into project root directory. Create a build directory, then enter the directory.

```
mkdir build
cd build
```

Run CMake

```
cmake ..
```

CMake should exit with

```
-- Configuring done
-- Generating done
-- Build files have been written to: /…/windrider/build
```

#### *Compile the Project*

From the build folder run make.

```
make -j4
```

Build folder will now contain three firmware files *windrider.bin, windrider.hex, windrider.elf* — these are compiled binary blobs ready to be flashed to the microcontroller.

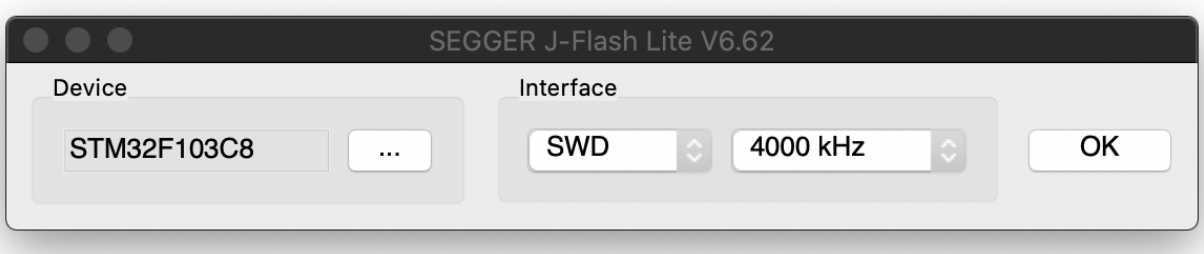
### *Flash*

#### *J-Link*

There are two ways to upload firmware using J-Link - Command Line and GUI.

#### *JFlashLite GUI*

Run JFlashLite with the following settings.



In the next prompt select *windrider.hex* as data file. Click **Erase**, then **Program**.

#### *JLinkExe Command Line*

Navigate to build folder. Connect to target.

```
JLinkExe -device "STM32F103C8" -if SWD -speed 4000 -autoconnect
```

Reser, erase, load, verify.

```
r
erase
loadbin "windrider.bin" 0x08000000
verifybin "windrider.bin" 0x08000000
```

#### *ST-Link*

Navigate to build folder. Run st-flash.

```
st-flash write windrider.bin 0x08000000
```

### *Run*

Connect host computer to WindRider board via USB. Use any modem control software (e.g. minicom) or serial port API (e.g. pyserial).

To print a full list of commands and arguments type ? and hit return. A list of available commands is available here_.

Some examples of valid commands:

```
solenoid 1 5 20       ## Enable impactor at channel 1, on_time = 5ms, off_time = 20ms.
servo 0 100           ## Set servo at channel 0 to 100 deg.
```

## Project Structure

The project tree below outlines the files that were added or modified in the development process. The project is organized according to STM CubeMX code generation tool. Any files that were were imported as STM Libraries and were not modified in the development process are omitted in the following tree.

### *File Tree*

```
|-- .vscode/
|   |-- launch.json             ## VSCode Build and Debug Configuration.
|
|-- CMakeLists.txt              ## CMake build configuration.
|
|-- Drivers/
|   |-- CMSIS/                  ## STM CMSIS drivers.
|   |-- STM32F1xx_HAL_Driver/   ## STM Peripheral drivers.
|
|-- Inc/
|   |-- CommandParser.h         ## Command Parser class declaration.
|   |-- CommandQueue.h          ## Flow control implementation used in USB Queue.
|   |-- FaulhaberComm.h         ## Faulhaber Communication class declaration.
|   |-- HardwareDriver.h        ## Command Handlers namespace declaration.
|   |-- Solenoid.h              ## Solenoid class declaration.
|   |-- UsbComm.h               ## Usb communication helper class declaration.
|   |-- initialization.h        ## Initialization functions declaration.
|   |-- stm32f1xx_it.h          ## Interrupt handlers declaration.
|
|-- Middlewares/                ## Contains STM USB Communication Class Driver.
|
|-- README.md                   ## Git readme.
|-- STM32F103C8Tx_FLASH.ld      ## Loader script.
|
|-- Src/
|   |-- CommandParser.cpp       ## Command Parser class implementation.
|   |-- FaulhaberComm.cpp       ## Faulhaber Communication class implementation.
|   |-- HardwareDriver.cpp      ## Command Handlers namespace implementationxs.
|   |-- Solenoid.cpp            ## Solenoid class implementationxs.
|   |-- UsbComm.cpp             ## Usb communication helper class implementationxs.
|   |-- initialization.cpp      ## Initialization functions implementation.
|   |-- main.cpp                ## Main function.
|   |-- stm32f1xx_it.cpp        ## Interrupt handlers implementation.
|   |-- usbd_cdc_if.cpp         ## Usb interface functions.
|
|-- startup/            ## Startup code.
|
|-- stm32f103c8t6.svd           ## CMSIS Peripheral Address Map (for debugging).
|-- windrider.ioc               ## Template STM CubeMX project configuration.
```

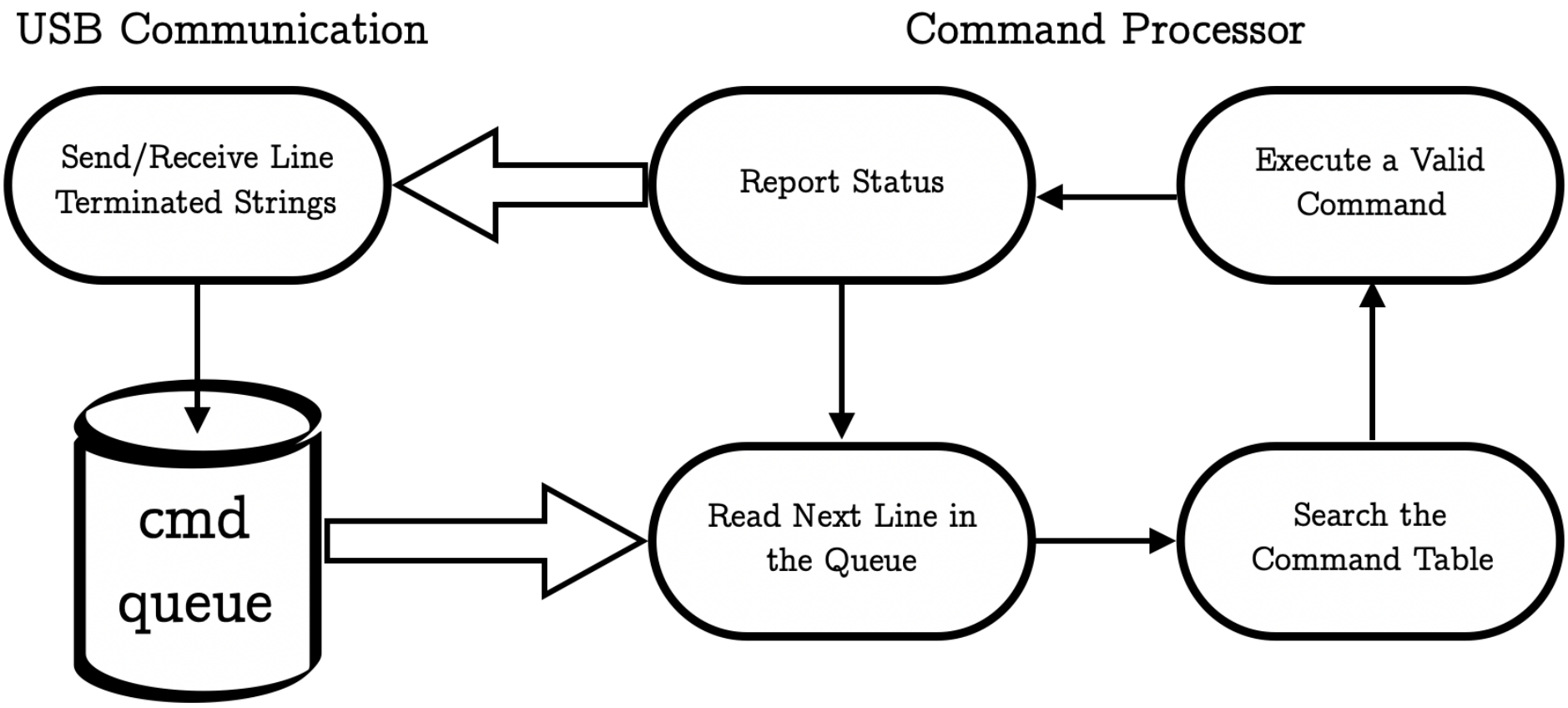
### *Class Tree*

Links to declarations of classes, namespaces, and interfaces. Find brief descriptions of member functions in comments.

- template class CommandQueue
- namespace CommandParser
- class CommandParser::Cmd
- class UsbComm
- class Solenoid
- namespace HardwareDriver

## Code Flow

At the top of the abstraction ladder, the project consists of two processes: USB Driver and Command Parser connected through a Command Queue.




### *The Main Loop*

USB Communication and Command Processor illustrated on the diagram above shake hands in the main loop. Strings stored in `UsbComm::usb_queue` are bing fed to the `CommandParser`, then `CommandParser` makes sense of the input and spits out a response, back to the `UsbComm`. Let's take a closer look to what exactly happens.

[view source]

```
/* Infinite loop */
while(true)
{
  // If there are commands in the usb queue -- execute them.
  if(UsbComm::usb_queue.get_queue_size() != 0){

      const auto response = CommandParser::execute(UsbComm::usb_queue.get_next_cmd());

      // Respond to the command
      UsbComm::usb_send(&response);

      // Remove pending command from the queue
      UsbComm::usb_queue.free_pending_cmd();
 }
```

In the snippet above, received commands are put into a queue elsewhere. Leave the reception details to the mysterious `UsbComm`. Suppose the `UsbComm::usb_queue.get_next_cmd()` returns `"servo 180"`, which tells our microcontroller to set a servo to 180 degrees.

Consider the following line:

```
const auto response = CommandParser::execute("servo 180");
```

`CommandParser::execute` searches the command table for a matching token (in this example token is `"servo"`). In case a match is found, the corresponding command handler is called. A substring of arguments (in this example `"180"`) that follow the token is then passed to the command handler function. Finally, a meaningful status is returned (`"ok"`, `"unknown command"`, `"invalid args"`, etc..).

### *Command Processor*

This section will shine light on what specifically happens inside `CommandParser::execute`, in other words, how do received commands get executed?

#### ***Behind*** `CommandParser::execute`

The following code shows implementation of `CommandParser::execute`. Make yourself comfortable with C++ Algorithms library.

[view source]

Code Flow

```
const std::string execute(const std::string &command){

    // Check if help requested.
    if(command[0] == '?'){
        std::string help_str = help;

        // Return a string with all the tokens and help strings.
        std::for_each(command_table.begin(),
                      command_table.end(),
                      [&help_str](const Cmd &cmd_table_entry){

                          help_str += cmd_table_entry.token;
                          help_str += cmd_table_entry.help;
                          help_str += "\n\r";
                      });
        help_str + "\r";

        return help_str;
    }

    // Find the first space to seperate a token.
    auto separator = command.find_first_of(' ');

    // If there is no space, assume command has no arguments.
    if(separator == std::string::npos)
        separator = command.size()-1;

    // Lookup the given token in the command table.
    // In case found, returns a poiter to the corresponding command container.
    const auto executable = std::find_if(command_table.begin(),
                                         command_table.end(),
                                         [&command, &separator](const Cmd &cmd_table_entry){

                                 return cmd_table_entry.token == command.substr(0, separator);
                             });

    // Check if given token exists in the command table.
    if(executable == command_table.end())
        return unknown_command;

    // Execute the corresponding command from the table.
    executable->execute(command.substr(separator + 1));

    // Return status of the executed command.
    return HardwareDriver::get_status();
}
```

Let’s break this down step-by-step.

1. Check if the first element in the string is `"?"`. In case it is - user requested help. For each command table entry print the token and the corresponding help string.

> ***Warning***
>
> In case help is requested, the function returns the entire content of the command table in a single string, it is not an issue as long as the table is relatively small, but potentially leads to an overflow.

2. We need to break down the received string into a token and arguments. Look for the first occurrence of space character `" "` and save it’s location into the variable `separator`. What if our command does not have arguments? Thus there is no space separator in the given string. In such case, set the separator to represent the last character in the string, which is always `"NULL"`.

   Now, the token can be acquired by calling `command.substr(0, separator)`. The argument string consists of all the characters past the separator, or `command.substr(separator + 1)`

3. Find if the token in the received string exists in the table. In case it does, a call to `const auto executable = std::find_if(...)` returns a pointer to the object in the the table with the matching token. In case such token does not exist, `const auto executable` points to the end of the command table.

4. If the token was not found, exit the function returning `unknown_command` status.

```
// Check if given token exists in the command table.
if(executable == command_table.end())
    return unknown_command;
```

5. If we got to this point, the token exits in the table, let’s execute the corresponding command handler. Remember we saved the pointer to the corresponding entry in the command table. Each entry in this yet mysterious table is an object which encapsulates a token, a help string and a pointer to the handler function. This object has a helper method, which can be called to execute the handler function `object.execute(...)`

```
// Execute the corresponding command from the table.
executable->execute(command.substr(separator + 1));
```

We feed the argument string to the command handler letting it further parse the arguments.

> ***Note***
>
> An attentive reader might notice that passing a raw argument string to the command handler means that each command handler must have a similar routine to further parse the arguments. A **better approach** would be passing a vector of argument-strings while recursively separating the arguments within the `CommandParser::execute` function. Keep in mind, these argument-strings can represent both strings and numbers.

6. Finally, return status. Why `HardwareDriver::get_status()`? All command handlers are part of the `namespace HardwareDriver`, where they share a status string.

***Command Table***

What underlying information do we need to know to execute a given command?

- A **Token**, command's first name.
- **Command Handler**, what should the microcontroller do when the token is called.
- A **Help String**, would be nice to know what the command does.

We can use a helper class to package all the info about each command. Take a look at `class CommandParser::Cmd` declaration.

[view source

```
//! Command container class.
/**
 *  @brief This container is used to place commands into the command lookup table.
 */
class Cmd {

    public:

    //! Default Cmd constructor.
    /**
     * @param token Command token used to call the specified command.
     * @param help  Help string.
     * @param cmd_handler Function to be executed when command with the token is called.
     */
    Cmd(std::string token, std::string help, std::function<void(std::string)> cmd_handler):
        token(token),help(help),cmd_handler(cmd_handler){};

    //! execute method.
    /**
     * @brief Public method to call the command handler.
     * @param arg Argument string passed to the command handler.
     */
    void execute(std::string arg) const{

        cmd_handler(arg);
    };

    const std::string token; // Command token.
    const std::string help; // Help string.

    private:

    // Command Handler function pointer.
    const std::function<void(std::string)> cmd_handler;
};
```

Note how the constructor populates the only three constant members. There is a single method to call the command handler. The mysterious command table is an array of `Cmd` objects.

> ***Warning***
>
> Example below is simplified for illustration, the actual command table can be found here.

```
// Command table. Please keep alphabetical order.
const std::array<Cmd, 2> command_table = {

 Cmd("led", "set led power [0-100]%", HardwareDriver::led),
 Cmd("servo", "set servo angle [0-180]deg", HardwareDriver::servo)

}
```

The table above initializes two entries with tokens `"led"` and `"servo"`

### *USB Communication*

USB Communication Device Class, in short USB CDC or virtual com port is a **Driver** provided by STM to handle all aspects of USB communication the microcontroller is capable of. When connected to a host, the Driver emulates a *tty-like* device on a Unix system (*COM* device on Windows). The nature of the USB protocol and the Driver's code itself are very sophisticated and will not be discussed in this manual. In fact, USB Driver takes up a significant amount of microcontroller's flash, in our case about 30%. The Driver resembles a process which handles initialization, pinging, power management, packet wrapping, redundancy checks, etc.. on the background giving us callback-based interface to RX/TX capabilities, and connect, disconnect events. These callbacks reside in the following source file.

Let's take a look at the **Receive** callback function.

[view source]

```
/**
  * @brief  Data received over USB OUT endpoint are sent over CDC interface
  *         through this function.
  *
  *         @note
  *         This function will block any OUT packet reception on USB endpoint
  *         untill exiting this function. If you exit this function before transfer
  *         is complete on CDC interface (ie. using DMA controller) it will result
  *         in receiving more data while previous ones are still not sent.
  *
  * @param  Buf: Buffer of data to be received
  * @param  Len: Number of data received (in bytes)
  * @retval Result of the operation: USBD_OK if all operations are OK else USBD_FAIL
  */
static int8_t CDC_Receive_FS(uint8_t* Buf, uint32_t *Len) {

   /* USER CODE BEGIN 6 */
   UsbComm::usb_queue.insert_received_elements(reinterpret_cast<char*>(Buf), Len);

   USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
   USBD_CDC_ReceivePacket(&hUsbDeviceFS);
   return (USBD_OK);
   /* USER CODE END 6 */
}
```

The STM's USB Receive callback leaves us with a pointer to a received data buffer and the buffer's length. Also, the comment above the code politely tells us that while we are enjoying our time inside the callback, usb reception is interrupted. Let's take the data and get out!

We could append the data into an array defined elsewhere, when we see `"\r"` execute it away. There are issues we need to consider. What if multiple commands are being received at the same time? or the callback leaves us with unfinished command? or both issues happen at once? In situation such as this we need **Flow Control**. Template class CommandQueue implements a Flow Controlled Queue based on C++ deque library, here are some features:

- Each entry in the queue is a `"\r"` terminated command.
- The insertion method ``insert_received_elements(uint8_t* Buf, uint32_t *Len) must take the same format as arguments passed to the the USB Receive Handler.
- The most recent entry can be unfinished, and `get_queue_size()` accounts for the unfinished entry.
- The oldest terminated command in the queue is accessible by a pointer, not a copy assignment.

We further instantiate a command queue in the USB helper `namespace UsbComm`. Notice the abstraction here, if we decided to use another interface, say SPI instead of USB, we would just need an SPI helper class instead of `UsbComm`, `CommandQueue` template would remain the same. The characters received through the USB callback are placed into `UsbComm::usb_queue`, a queue of strings.

```
UsbComm::usb_queue.insert_received_elements(reinterpret_cast<char*>(Buf), Len);
```

Back to our Main function, `UsbComm` serves as an abstraction layer between STM's Driver and `CommandParser` providing functionality to send and receive data.

[view source]

```
//! class UsbComm
/**
 * @brief A helper class to handle usb cdc communication.
 */
class UsbComm {

    public:
    //! printf implementation
    /**
     * @brief This function sends a string over USB virtual com port.
     * @param str_to_send Pointer to the string to be sent.
     */
    static void usb_send(const std::string *str_to_send);

    // A Queue of commands seperated by '\r'
    static CommandQueue<std::string, '\r'> usb_queue;
};
```

We have covered the abstract firmware architecture and code flow. The next chapter covers `HardwareDriver` and the usage of microcontroller peripherals.

## Command Interface

### *Command Table*

The table of all available commands is shown below.

Command Interface

| Token | Argument 1 | Argument 2 | Argument 3 | Description |
|---|---|---|---|---|
| current | None | None | None | Print suction current. |
| led | #channel [0-3] | [on] or [off] | None | Turn led on/off (uses the most recent currentconfiguration) |
| led | #channel [0-3] | current [0-1500]mA | None | Set led current. |
| move | motor1 rpm | motor2 rpm | None | Set motor speed. |
| servo | #channel [0-1] | Angle [0-180]deg | None | Set servo angle. |
| solenoid | #channel [0-1] | [on] or [off] | None | Enable/Disable impactor. |
| solenoid | #channel [0-1] | [2-5000]ms on_time | [2-5000]ms off_time | Set impactor on/off time intervals. |
| suction | [on] or [off] | None | None | Enable/Disable suction power. |
| uart | [forward] | [“cmd_to_send”] | None | Forward “cmd_to_send” to rs232 |
| uart | [reply] | [on] or [off] | None | Print incoming feedback from rs232 |

Detailed command description is presented below.

### *current*

Prints current supplied to suction motor in mA.

***Example:***

```
current      ## Request suction current. Command has no arguments.
1234         ## Returned: 1.234 A
```

***Peripherals***

Command Handler: `HardwareDriver::suction_current(std::string arg)`

[view source]

ADC1 Measures the voltage level on port GPIOA, pin GPIO_PIN_5.

> ***Warning***
>
> Due to the hardware issues with compatibility of the hall effect sensor with the isolated amplifier, this feature is currently disabled.

### *led*

The led drivers are programmable current sources – adjust the voltage to maintain a set current. This command manipulates the current sources.

***Example:***

```
# Configure led brightness. This does not turn leds on.
led 0 100   ## Set current on channel 0 to 100mA
led 1 200   ## Set current on channel 1 to 200mA
led 2 1000  ## Set current on channel 2 to 1A

# This turns all led channels on at once
led on

# We can turn all channels off at once as well

led off

# Or turn channels on/off selectively

led 0 on    ## Turns on only channel 0.
led 2 off   ## Turns off only channel 2.
```

***Peripherals***

Command Handler: `HardwareDriver::led(std::string arg)`

[view source]

Led drivers require pwm input, the duty cycle is mapped proportionally to the current output of the led drivers. The drivers will automatically adjust the voltage to maintain constant current.

- 0% duty cycle — 0mA current
- 100% duty cycle — 1500mA current

The pwm generation is mapped to the following peripherals.

LED0 – TIM2_CH1 – GPIO_PIN_0 – GPIOA
LED1 – TIM2_CH2 – GPIO_PIN_1 – GPIOA

Timer reload frequency is 50Hz (to match with servo control).

> ***Warning***
>
> TIM2 is shared between led drivers and servos.

### *move*

This command directly loads speeds into Faulhaber motor drivers.

***Example***

```
move 1000 1000       ## Rotate both motors at 1000rpm.
move -1000 1000      ## This will make the robot rotate around itself
```

***Peripherals***

This command uses `class FaulhaberComm` underlying uart driver to communicate to the motors.

### *servo*

This command manipulates servo angles by adjusting duty cycle on the timer outputs. There are currently two channels allocated for 5V towermicro-like servos.

***Example***

```
servo 0      ## Set servo to the 0 deg position
servo 77     ## Set servo to 77 deg position
```

***Peripherals***

Command Handler: *HardwareDriver::servo_angle(std::string arg)*

[view source]

The servos are driven by **pwm** signal's **high time**:

1ms – 0º
2ms – 180º

***Mapping***

SERVO1 – TIM2_CH3 – GPIO_PIN_10 – GPIOB
SERVO2 – TIM2_CH4 – GPIO_PIN_11 – GPIOB

Timer reload frequency is 50Hz.

> ***Warning***
>
> TIM2 is shared between led drivers and servos.

### *solenoid*

This command controls the impactors.

> ***Warning***
>
> Impactor *on/off* timing must be initialized prior to running the impactors. Improperly initialized timings **can burn the impactors**.

Once `solenoid 0 on` command has been issued, the impactor at channel 0 will continuously run utilizing the *on/off* timing pattern. `on_time` corresponds to the time interval in milliseconds when the current is supplied to the impactor. Similarly, `off_time` – current is not supplied.

> ***Note***
>
> You might be curious why impactor timing is limited to 2ms. The reason lies in hardware. Solenoids are connected to the 24V line, thought most of the solenoids run at 12V. To mitigate the difference in voltages, the output voltage of the solenoid control pin is set to a software generated square wave with 50% duty cycle. The timer responsible for toggling the solenoid control pins runs with a resolution of 1ms, thus min period is 2ms.

***Example***

```
solenoid 0 4 20      ## Configure channel 0 impactor on/off timings.
solenoid 0 on        ## Start the impactor at channel 0.
solenoid 0 off       ## Stop the impactor at channel 0.
## Same applies to channel 1.
```

***Peripherals***

Command Handler: `HardwareDriver::solenoid(std::string arg)`

*[view source] <https://github.com/ccny-ros-pkg/WindRider-II_firmware/blob/master/Src/HardwareDriver.cpp#L202>*

Two solenoid transistor switches are driven by gpio pins. TIM3 is used to handle on/off times with 2ms resolution.

***Mapping***

SOLENOID0_Pin – GPIO_PIN_12 – GPIOB
SOLENOID1_Pin – GPIO_PIN_13 – GPIOB

SOLENOID1_Pin is defined as SPARE pin on the circuit schematic.

## *suction*

This command controls the suction motor power in percentage. The on/off option physically connects and disconnects the suction motor circuit. Setting power before enabling the suction circuitry will result in no action.

***Example***

```
suction on   ## Enable suction motor circuitry.
suction 42   ## Set power to 42%.
suction 80   ## Set power to 42%.
suction 0    ## Turn off suction, but keep the circuitry enabled.
suction off  ## Disable suction motor circuitry.
```

***Peripherals***

Command Handler: `HardwareDriver::suction_power(std::string arg)`

[view source]

Suction motor driver is controlled by a 1kHz pwm signal generated on TIM3. A gpio pin is used to enable/disable the motor circuitry.

0 duty_cycle – 0% power
0.5 duty_cycle – 50% power
1 duty_cycle – 100% power

***Mapping***

SUCTION_PWM – TIM3_CH1 – GPIO_PIN_6 – GPIOB
SUCTION_EN_Pin – GPIO_PIN_4 – GPIOA

## *uart*

This command implements usb-to-uart forwarding feature, which allows to forward character strings from the host computer to the uart interface of the microcontroller, in this case connected to rs-232 and further Faulhaber motor drivers.

***Example***

Let’s say we would like to load a custom configuration into the connected Faulhaber motor drivers.

```
uart "1SP1000"       ## string "1SP1000\r" will be forwarded through uart.
```

Faulhaber motor must have replied something, the reply forwarding is disabled by default. Use the following command to enable the forwarding.

```
uart reply on        ## Enables reply forwarding.
```

### *Warning*

After issuing the command above everything received on UART will be forwarded to host through USB until `uart reply off` is issued.

***Peripherals***

Command Handler: `HardwareDriver::uart(std::string arg)`

[view source]

`class FaulhaberComm` implements helper methods to send and receive strings via USART1 peripheral. Please see *FaulhaberComm.cpp/h* for the further description.