ArgosSwerveDocumentation

ArgosSwerve Documentation

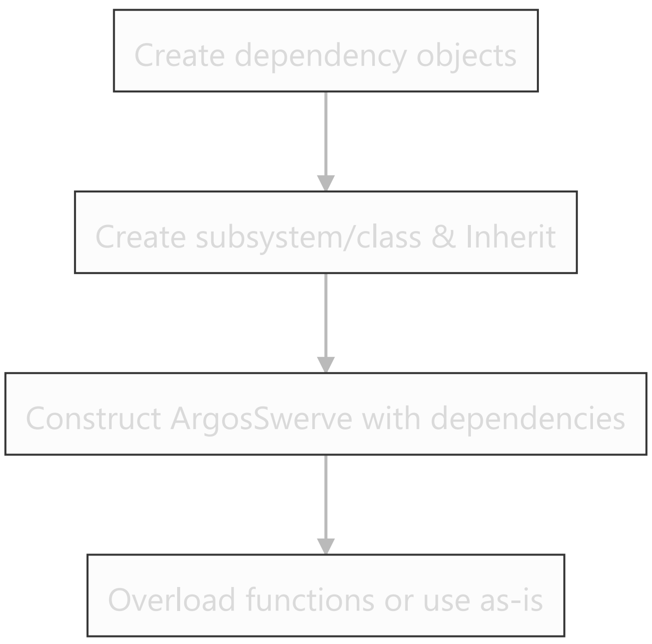
Overview

This documentation describes the `argos_lib::swerve::ArgosSwerve` class, and it's dependencies. The first part of this document is an implementation guide, and the last part is essentially reference/summary.
If you're an intermediate user, go check out the ***Docs for Nerds*** at the bottom of this document, examples of everything mentioned in ***Docs for Nerds*** can be found in the guide.

General Idea:

Using this class is *super easy* given a little background information. `ArgosSwerve` is meant to be inherited into a subsystem, and gives a basic drivetrain to build off of. I highly highly **highly** recommend reading/skimming the `argos_swerve.h` header file, to get and idea of the workings under the hood, and read some descriptions to get a better idea than what may be presented here.
Here is a link if you're unfamiliar with inheritance in C++

You can think of designing with this base class as a sort of *flow* as described:



Create Dependency objects

You'll need some information for the robot to consume. As much as the `ArgosSwerve` class takes care of, making a omni-directional drivetrain with a variable number of wheels requires a bit of info.
Some context is important, each swerve module has it's own config object that `ArgosSwerve` uses to configure itself. You can kind of think of each of these objects as a virtual representation of what exists in the real world, remember, the computer only knows as much about your hardware as you tell it!
So what objects do we need? The constructor of `ArgosSwerve` will tell us. You can find it in `argos_swerve.h`
Constructor:

```cpp
/**
 * @brief Construct a new Argos swerve object
 *
 *
 *
 * @param config Configuration containing all the swerve modules
 * @param imu Pointer to an ArgosIMU
 * @param instance The instance of robot, (Competition? Practice?)
 * @param controlMode The control mode to initialize to
 */
ArgosSwerve(const ArgosSwerveConfig<N>& config, ArgosIMU<IMU>* imu,
            argos_lib::RobotInstance instance,
            SwerveControlMode controlMode = SwerveControlMode::FIELD_CENTRIC)
```

So the object types we need are:

- `ArgosSwerveConfig<N>` -> A Argos Swerve config object with all the necessary configuration required for a swerve drivetrain
- `ArgosIMU*` -> A [pointer] to a `ArgosIMU` IMU. This IMU is used to help the drivetrain understand which way it is facing, very helpful for field-centric driving.
- `RobotInstance` -> An instance of robot, we use this to distinguish between the practice and competition bot.
- `SwerveControlMode` There is no need to change this parameter if you don't need the robot to drive in robot centric by default. For this guide, I'll leave it alone.

So first, we need to get a `ArgosSwerveConfig<N>` object. But how?
Well, we need to *construct* it too. Let's look at the constructor for this class as well, so we can create an object of it to give to `ArgosSwerve` to use:

```cpp
/**
 * @brief Construct a new Argos Swerve Config object containing all the
 * necessary config to run the drivetrain
 *
 * @param turnEncoderResolution The resolution of the encoder on the turn
 * motor
 * @param homesPath The path to the file containing the home values
 * @param maxVelocity The maximum velocity for determing maximum change
 * velocity during optimization
 * @param wheel The first ArgosModuleConfig object
 * @param wheels The rest of the module configuration objects
 */
template <typename... Wheels>
explicit ArgosSwerveConfig(
    const double turnEncoderResolution, const std::string& homesPath,
    const units::velocity::feet_per_second_t maxVelocity,
    const ArgosModuleConfig& wheel, const Wheels&... wheels)
    : m_homesPath{homesPath},
      m_turnConversionFact{360.0 / turnEncoderResolution},
      m_maxVelocity{maxVelocity},
      m_moduleConfigs{wheel, wheels...},
      m_chasisOffsets{wheel.chassisOffset, wheels.chassisOffset...} {}
```

We need to know a bunch of information for this class too, unfortunately, but I promise it's almost over. The only type I'll list here as being needed is the `wheel` and `wheels` parameters, which are of type `ArgosModuleConfig`. The rest can be passed pretty much as literals (I'll show an example in a minute)

- `turnEncoderResolution` should be `4096`, but ask around to make sure
- `homesPath` probably `/homes/swerveHomes`, this is the location on the RoboRio where the homes are saved to a file. Ask the mentor who set up the RoboRio to see where to put homes.
- `maxVelocity` The maximum velocity of the drivetrain (This is just there for the optimizer, and as of 8/6/22 doesn't actually limit the drivetrains velocity)

> ♨ **Important**
>
> For whomever is administrating the roborio, make sure the user who runs the executable for the robot program has rights to access the specified directory

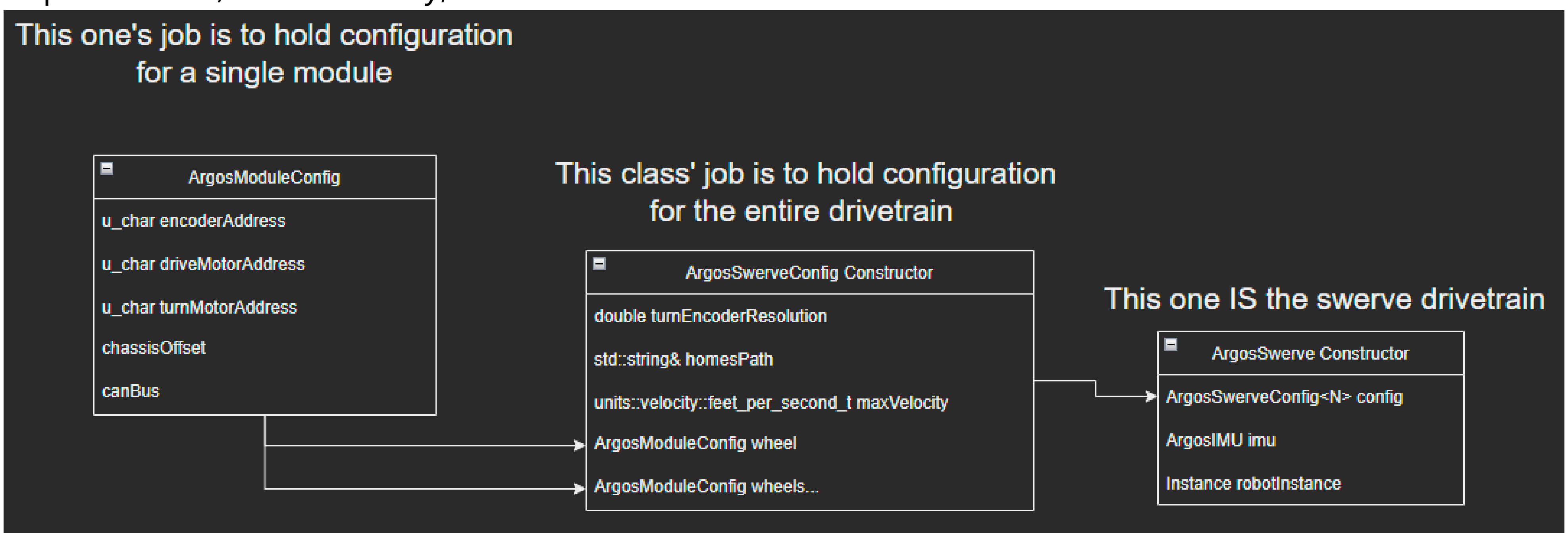
So, with that being said, let's look at the constructor for `ArgosModuleConfig`:

```cpp
/**
 * @brief A struct that contains all configuration options specific to modules
 * to pass to ArgosSwerveConfig class
 *
 */
struct ArgosModuleConfig {
  const u_char encoderAddress;
  const u_char driveMotorAddress;
  const u_char turnMotorAddress;
  const frc::Translation2d chassisOffset;
  const std::string_view canBus;  ///< CAN line on the CANivore to use
  constexpr ArgosModuleConfig(char encoderAddress, char driveMotorAddress,
                              char turnMotorAddress,
                              const frc::Translation2d& chassisOffset,
                              const std::string_view canBus)
      : encoderAddress{encoderAddress},
        driveMotorAddress{driveMotorAddress},
        turnMotorAddress{turnMotorAddress},
        chassisOffset{chassisOffset},
        canBus{canBus} {};
};
```

And these are the last of the `ArgosSwerve` classes for `ArgosSwerveConfig`, (The only other constructor I'd recommend looking at is frc::Translation2d)

- `encoderAddress` -> the address of the turn motor's encoder on the CAN bus
- `driveMotorAddress` -> address of drive motor on the CAN bus
- `turnMotorAddress` same as the last one, but for the turn motor
- `chassisOffset` The offset from the center of the robot in the form of a `frc::Translation2d` object
- `canBus` The can bus that all these objects are located on

This `ArgosModuleConfig` object is what I was talking about earlier, what with "representing real-world objects in code" and stuff. Let me put a little graphic here describing our journey down these dependencies, as admittadly, it can be a little hard to understand at first.



As you can see, `ArgosSwerve` needs to know about (depends on) it's configuration, `ArgosSwerveConfig`. And `ArgosSwerveConfig` depends on it's list of `ArgosModuleConfig`

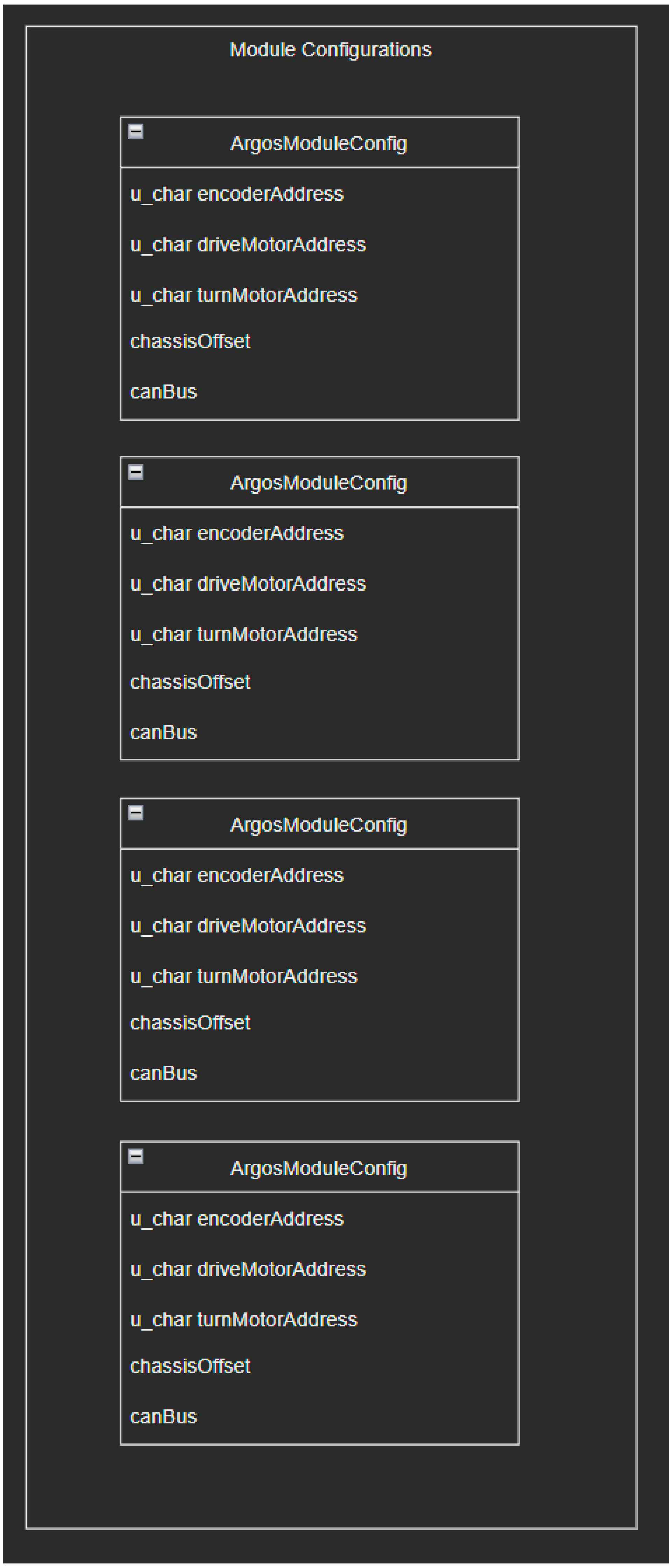
So we need to work our way from left to right of the diagram, constructing the nceccissary information. Finally, let's look at some code:
Find a place to construct and store all your `ArgosModuleConfig` objects. (I recommend in constants somewhere). Here's an example object:

```
const ArgosModuleConfig frModule{
    1,
    2,
    3,
    frc::Translation2d{12_in, -10_in},
    "drive"
};
```

This is the front right module of a robot, it has encoder address 1, drive motor address 2, turn motor address 3, it's 12 inches to the front, and 10 inches to the right from the center of the robot.
`frModule` is the name of our object, of type `ArgosModuleConfig` we now have an in-code representation of a real module.
Please Note:
This is coordinate system the robot runs off of (why the point of the module is (12, -10)):

> **Remember**
>
> Encoders and motors run on different addresses. You can have an encoder with address 1 and a motor with address 1 without a conflict.

This process will have to be repeated for every module, I'll go ahead and say I made 3 more modules, `flModule`, `blModule`, and `brModule` (front left, back left, back right)
Here is a visualization of the information we constructed:



Now it's time to construct an `ArgosSwerveConfig` object. It's recommended to do this inside the subsystem you are implimenting `ArgosSwerve` in, but you don't have to.
Keep in mind that while my names may be different, the general structure of the file I'll be using is pretty standard.
I'll leave it up to you to go back and look at what the constructor for an `ArgosSwerveConfig` module looks like, so I'll jump strait into code:

```
/* my_drive_subsystem.cpp*/
#include <subsystems/my_drive_subsystem.h"

#include <string>

```

```cpp
#include "Constants.h" // Or whatever file where all your module configs are
using argos_lib::swerve::ArgosSwerveConfig;

ArgosSwerveConfig config = ArgosSwerveConfig<4>{
        4096,
        std::string("/homes/swerveHomes"),
        12_fps,
        flModule,
        frModule,
        brModule,
        blModule
}

MyDriveSubsystem::MyDriveSubsystem(){
/* We'll do this later */
}
```

> **Important**
>
> Please note that the order in which you pass the modules to ArgosSwerveConfig **Matters**, it determines in what order they are stored in the internal vector. So, in this case it would be synonomous to `std::vector<ArgosModuleConfig> config{flModule, frModule, brModule, blModule}`

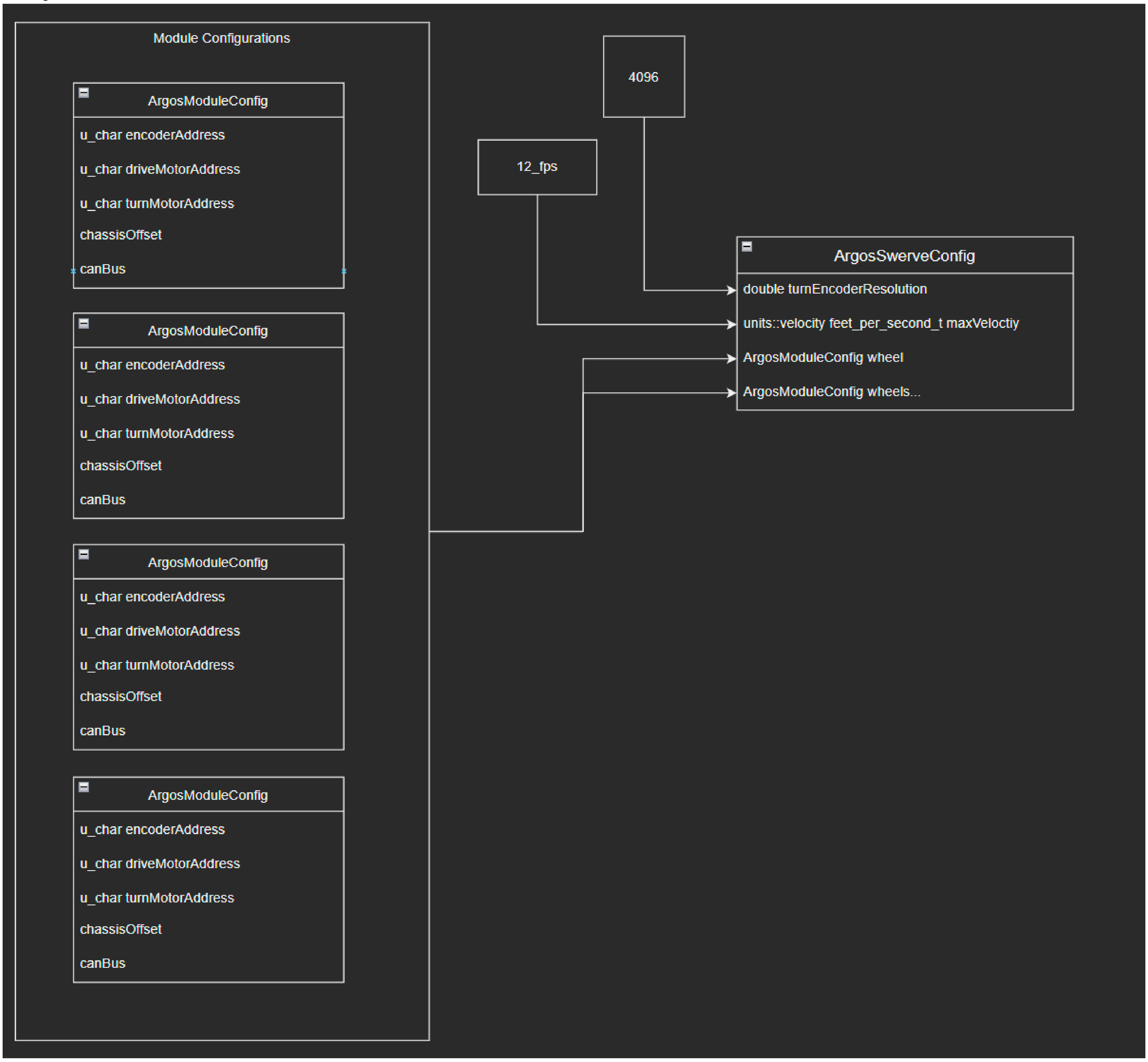
We just made configuration for our Swerve Drivetrain!! Let me explain the little `<4>`, If you're very interested go ahead and look up templates in c++, but on a very high level, it essentially tells `ArgosSwerveConfig` how many modules we are giving to it. You may also notice that `ArgosSwerveConfig` only seems to takes 5 parameters, but we passed 7???, that's because of line 106 here in the constructor:

```cpp
106    template <typename... Wheels>
107    explicit ArgosSwerveConfig(
```

That's a *parameter pack*and is very useful for passing in variable number of parameters in functions.

We now are describing a drivetrain whose encoders have a resolution of `4096`, is storing it's homes in "/homes/swerveHomes" is optimized to go a max of 12_fps, and has 4 modules, `flModuel`, `frModule`, `brModule`, and `blModule`.

We just scratched another task off our list:



We just need one more thing before we can construct the actual drivetrain... The IMU!

There is a class, `ArgosIMU` that contains an IMU that can either be an ADIS16448 IMU or Pigeon 2 IMU for use with the `ArgosSwerve` you'll have to construct one of these to give to the `ArgosSwerve` object we'll be creating. In `argos_imu.h`, you can find the constructor for the IMU:

```
explicit ArgosIMU(T *imu, ArgosAxis upAxis, ArgosAxis forwardAxis)
```

The IMU is of type `T` which is declared by the template parameter at the top of the class here:

```
template <typename T>
class ArgosIMU {        You, 15 hours ago • Initial commit
 public:
  explicit ArgosIMU(T *imu, ArgosAxis upAxis, ArgosAxis forwardAxis)
      : m_imu{imu}, m_upAxis{upAxis}, m_forwardAxis{forwardAxis} {
    Configure();
  }
```

This tells us we have to specify a template parameter containing the type of the IMU, (Similarly to specifying the number of modules in `ArgosSwerveConfig`). The two valid types for the IMU are: `frc::ADIS16448_IMU` and `ctre::phoenix::sensors::Pigeon2`

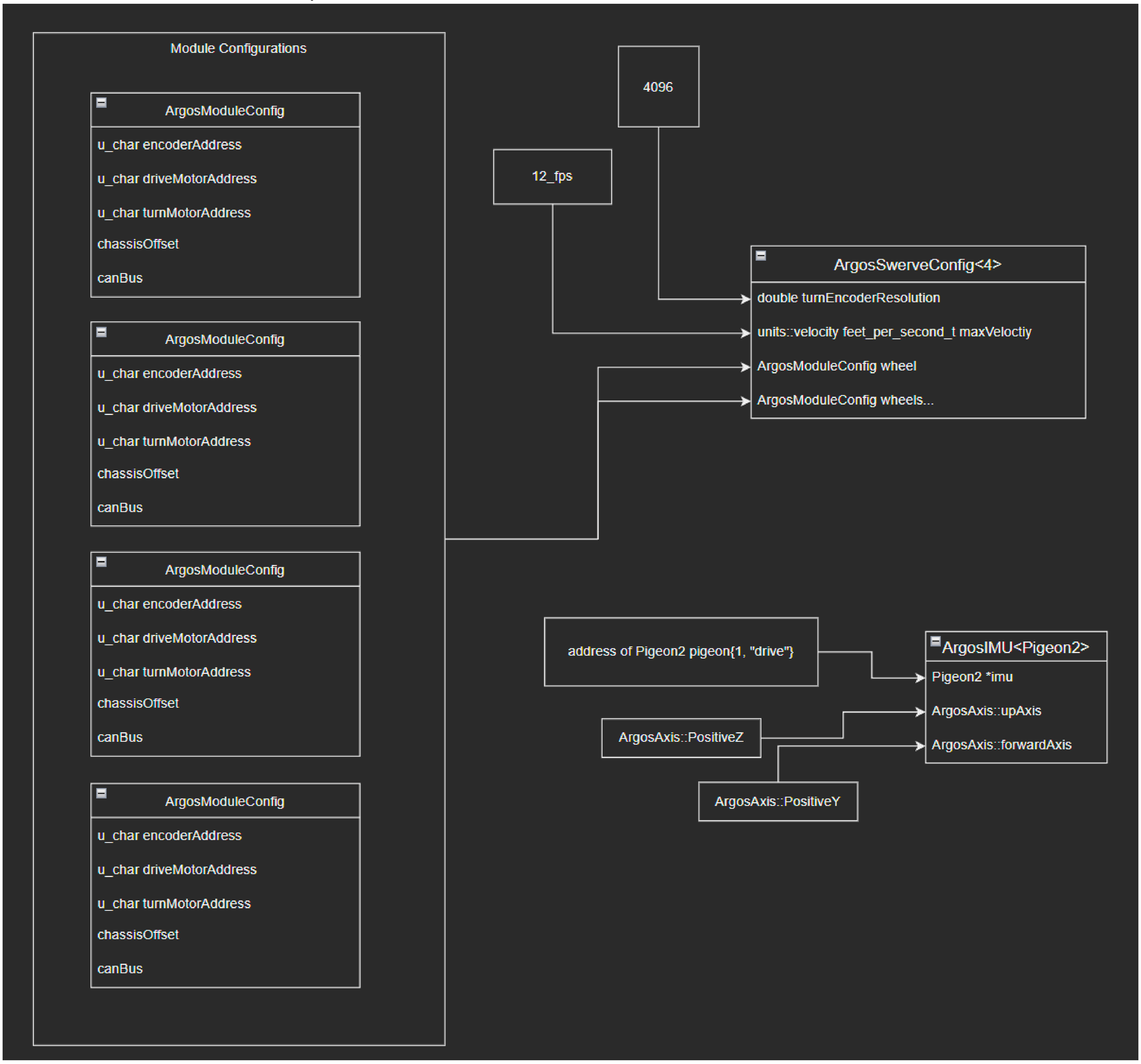
The second parameter is of type `ArgosAxis` which is also defined in this file, a little above the constructor for the `ArgosIMU`:

```
enum ArgosAxis {
  PositiveZ,
  PositiveY,
  PositiveX,
  NegativeZ,
  NegativeY,
  NegativeX
};
```

This is what we supply the the `upAxis` and `forwardAxis`, which is just the axis of the IMU that is pointing in those respective directions.
(In our case, let's assume we are using a pigeon2 IMU, and it's positive z axis is up, and it's y axis is forward)
Let's construct our IMU (I put this in the header file for my subsystem)

```
/* my_drive_subsystem.h*/

#include "argos_lib/general/argos_imu.h"
#include "ctre/phoenix/sensors/Pigeon2.h"

using ctre::phoenix::sensors::Pigeon2;
using argos_lib::swerve::ArgosIMU;
using argos_lib::swerve::ArgosAxis;

class MyDriveSubsystem : public frc2::SubsystemBase{
        public:
                MyDriveSubsystem();
          /**
           * Will be called periodically whenever the CommandScheduler runs.
           */
          void Periodic() override;

          void Disable();

        private:
                Pigeon2 pigeonIMU{1, "drive"};

                ArgosIMU<Pigeon2> m_pigeon{
                        &pigeonIMU, ArgosAxis::PositiveZ, ArgosAxis::PositiveY
                };
};
```

You can visiualize all our dependencies as this so far:



We have now collected all the information necessary to finally construct the drivetrain.
First step is to inherit the `ArgosSwerve` base class from the selected drive subsystem, like so:

```
/*my_drive_subsystem.h*/
#include "argos_lib/general/argos_imu.h"
#include "ctre/phoenix/sensors/Pigeon2.h"

using ctre::phoenix::sensors::Pigeon2;
using argos_lib::swerve::ArgosIMU;
using argos_lib::swerve::ArgosAxis;

class MyDriveSubsystem : protected argos_lib::swerve::ArgosSwerve<4, Pigeon2>
                                           , public frc2::SubsystemBase{
        public:
                MyDriveSubsystem();
          /**
           * Will be called periodically whenever the CommandScheduler runs.
           */
```

```
            void Periodic() override;

            void Disable();

        private:
                Pigeon2 pigeonIMU{1, "drive"};

                ArgosIMU<Pigeon2> m_pigeon{
                        &pigeonIMU, ArgosAxis::PositiveZ, ArgosAxis::PositiveY
                };
};
```

You'll notice the line `protected argos_lib::swerve::ArgosSwerve<4, Pigeon2>` in the header file. this line inherits the members in `argos_lib::swerve::ArgosSwerve` that you need to get swerve drive running. the `<4, Pigeon2>` are the two template parameters, telling `ArgosSwerve` how many modules are on the drivetrain, and what IMU to use.

Moving to the cpp file, the syntax is identical, except you call the constructor and pass the necessary parameters:

```
/*my_drive_subsystem.cpp*/
#include <subsystems/my_drive_subsystem.h"

#include <string>

#include "Constants.h" // Or whatever file where all your module configs are
using argos_lib::swerve::ArgosSwerveConfig;

ArgosSwerveConfig config = ArgosSwerveConfig<4>{
        4096,
        std::string("/homes/swerveHomes"),
        12_fps,
        flModule,
        frModule,
        brModule,
        blModule
}

MyDriveSubsystem::MyDriveSubsystem()
        : argos_lib::swerve::ArgosSwerve<4, Pigeon2>(config, &m_pigeon,
                argos_lib::RobotInstance::Competition,
                argos_lib::swerve::SwerveControlMode::FIELD_CENTRIC){
        // Constructor stuff here
}
```

In doing this, we have inherited from the `ArgosSwerve` class and constructed an instance with the config we made earlier, the address of the IMU of choice (has to be same type as we specified in template), it's a competition instance, and the default control mode if FIELD_CENTRIC control.

You now have access to all these public functions of the `ArgosSwerve` class

- `GetRawModuleStates()`
- `GetFieldCentricAngle()`
- `FieldHome()`
- `SetControlMode()`
- `HomeDrivetrainToFS()`
- `Home()`
- `ResetIMU()`
- `InitDrivetrainHomes()`
- `HomeFieldCentric()`
- `ToSensorUnit()`
- `ToAngle()`
- `Drive()`
- `StopDrivetrain()`
- `ConfigModuleDevice()`
- `ConfigAllModuleDevice()`

The workings / uses of these functions is outside the scope of this documentation, reference the header file for function descriptions. The most useful out of these is probably:

- `Drive()`
- `InitDrivetrainHomes()`
- `FieldHome()`
- `HomeDrivetrainToFS()`
- `StopDrivetrain()`

Two things the end user has to do before calling `Drive()` or any other driving functions is configuring motors. The `ConfigModuleDevice()` and `ConfigAllModuleDevice()` functions try to make this a little easier, allowing you to configure all devices of a type at once on the drivetrain with one.

> **Important**
>
> Secondly, you'll have to call `InitDrivetrainHomes()` with saved homes before doing any driving. Could you call `Drive()` without this? sure. Would it work to well? Probably not.

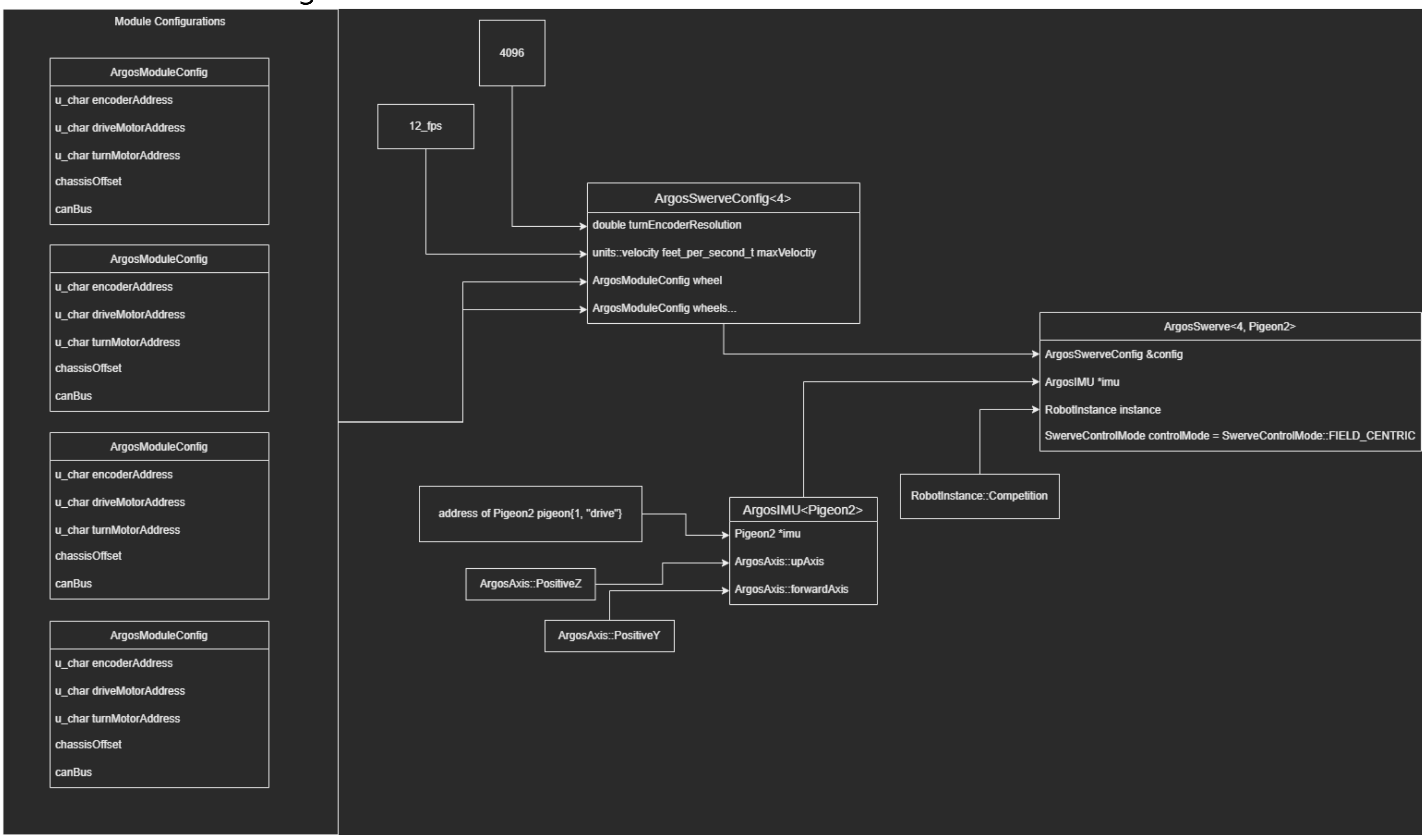
Docs For Nerds

`ArgosSwerve<N, IMU>` (N=Num Modules, IMU = IMU type) is meant to be treated as a base class to the drive subsystem controlling a swerve drivetrain (just like `SubsystemBase`, but more specialized), giving the user immediate access to a swerve drivetrain, and abstracting a lot of boiler-plate. (I used the `MySwerveSubsystem` class as an example above). `ArgosSwerve` depends on information passed to it in the form of a `ArgosSwerveConfig<N>`. `N` being a template param describing the number of modules. The user has to construct an `ArgosSwerveConfig<N>` to pass to `ArgosSwerve<N, IMU>`. `ArgosSwerveConfig` constructor has a parameter pack that allows a variable number of params for

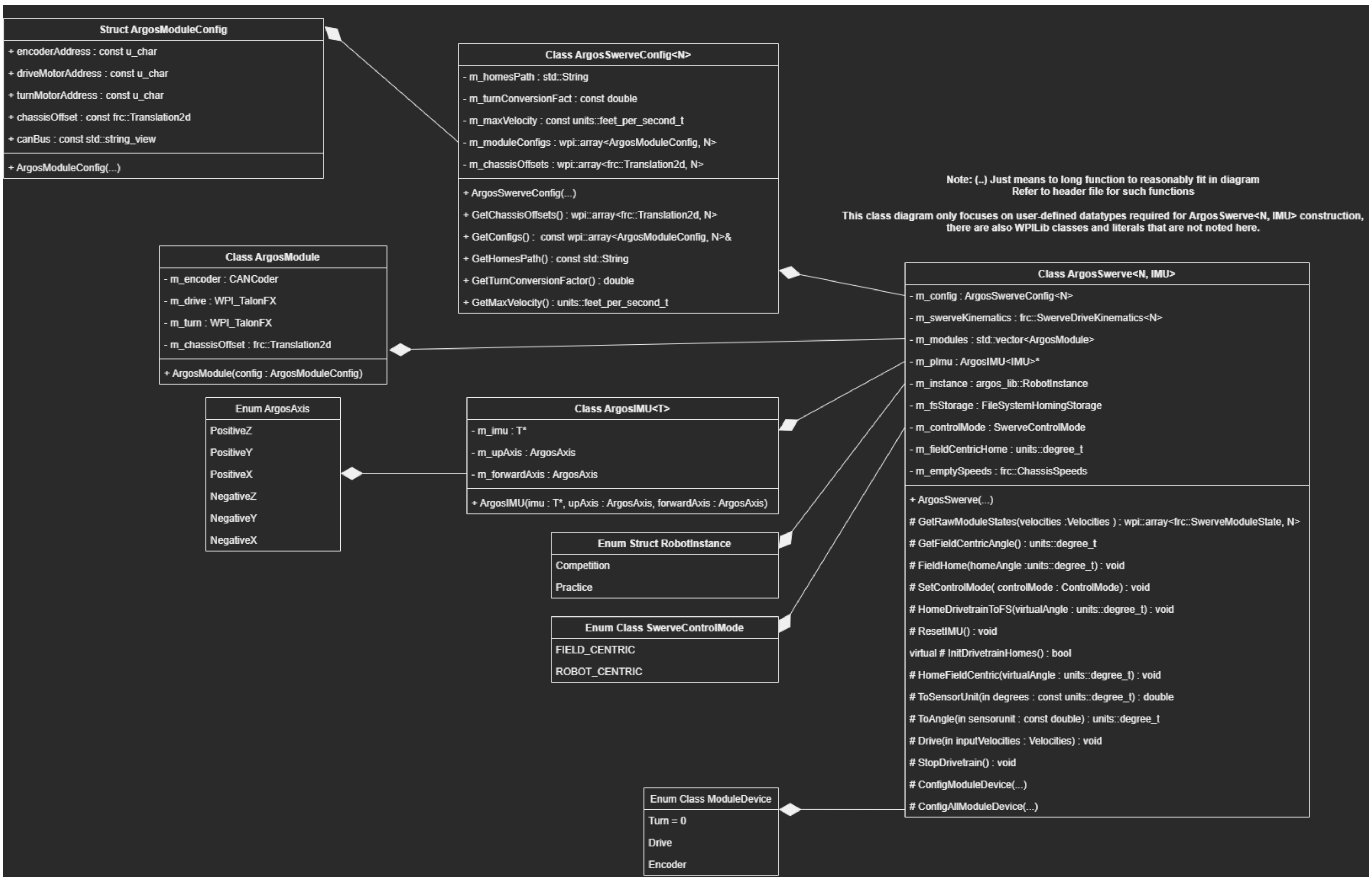
`ArgosModuleConfig`s (number of `ArgosModuleConfig` objects should correspond to `N`) to be passed to the constructor. An `ArgosIMU<IMU>` must also be passed to `ArgosSwerve<N, IMU>`. The rest of the data passed to the constructors of these data structures is mostly self-explanatory. I'll explain the responsibilities of the data structures in `argos_swerve.h`:

- `ArgosSwerve<N, IMU>` Base class for swerve drive subsystems (class)
- `ArgosSwerveConfig<N>` Holds information used by `ArgosSwerve` during the operation/initialization of the drivetrain, such as the struct representations of swerve modules. Other information includes max speed, homes path, and turn encoder resolution (class)
- `ArgosModuleConfig` is a struct that holds the addresses for drive motor, turn motor, and turn encoder along with chassis offset and the can bus. (struct)
- `ArgosModule` Is a struct that contains the motor/encoder objects for a module. (stuct)
- `SwerveControlMode` An `enum` for robot or field centric control (enum class)
- `ModuleDevice` An `enum` for differentiating module devices (drive motor, turn motor, encoder) (enum class)
- `ArgosIMU<IMU>` Not in `argos_swerve.h`, but is a container class for an IMU which is used in the `ArgosSwerve` class. (class)

I very very *highly* encourage you to take a look at the details in the `argos_swerve.h`, and `argos_imu.h` files to get a better idea how things are being passed around. Here's a graphic of a 4 module drivetrain to get a better idea of how construction of a drivetrain works:



And a class diagram...



Construct a `ArgosModuleConfig` class representing every swerve module, pass that and other info into an `ArgosSwerveConfig` object you construct, then throw that into a `ArgosSwerve<N, IMU>` with an `ArgosIMU` and you'll have swerve drive up in no time.

DrivetrainConfig

Configuration of motors and calling of homing/initialization is still up to EU, but hopefully this will be a little easier with this library.
This system comes with 2 member / helper functions to assist with configuration:

- `ConfigModuleDevice<Competition, Practice>(unsigned char moduleIndex, ModuleDevice dev, units::time::millisecond_t timeout = 100_ms)`
- `ConfigAllModuleDevice<Competition, Practice>(ModuleDevice dev, units::time::millisecond_t timeout = 100_ms)`
  The above `ConfigModuleDevice` function takes in the index of the module to be configured (In terms of the internal vector containing the modules, they are in the order you passed them to `ArgosSwerveConfig<N>()`), then the type of device you are configuring (encoder, turn, drive), and then you can pass an optional timeout. The template parameters are just supposed to be a struct containing all these values. (This is identical to `argos_lib::falcon_config::FalconConfig<Competition, Practice>`, which is what both these functions call on the back-end)
  `ConfigAllModuleDevice` does the same thing, but omits the first function param as it applies the configuration to **all** the devices.

ArgosIMU

Argos IMU is a simple wrapper class for an IMU which allows the user to specify either an `ADIS16448_IMU` IMU or a `Pigeon2` IMU object. The functions available are similar to the old functions

used in drivetrain with the `ADIS6448` IMU. Really the only two things to be concearned with in this class is the `ArgosIMU` and the `ArgosAxis`. `ArgosAxis` is used to specify the axis pointing robot up. (used for the `GetAngle` functionality).
Axis are positive counter-clockwise and `ArgosAxis` has negative angles to account for an inverted forward or *up* axis.

ArgosModule
I think it's important for me to mention that the class `ArgosModule` is used only internally to `ArgosSwerve` and is not used by the EU.

All functions and classes are documented in code, and should be accessible in the Doxygen docs too.