Algorithm overview:

Conceptually, the software for the arm can be split into three different phases -- communications, where we get commands and send telemetry, control algorithm, where we decide how we want to interpret the command based on things such as feedback from sensors and etc, and finally actually causing the motor/servo/whatever to move.

The second and third jobs are the main focus in this document. They are a problem in that they entirely depend on what the arm is specifically doing -- IE moving the arm based on simple speed, or wanting to tell it to go to a certain position -- and what type of motor device the arm is actually using. All of these things can be changed and modified, which means the software for them needs to be able to adapt as well.

To save development time and allow multiple programmers to work at once AND allow quick integration of different motors servos etc etc, we shall develop software for these two phases using a framework that enforces modularity and allows for plug-and-play style development. We call this our control framework, the architecture we’ll use.

The idea is thus: for each type of algorithm -- move based on speed without feedback, move based on speed with feedback, etc -- and for each type of motor controller device the armboard shall talk to -- a discrete (made out of non-IC components) H bridge, a motor controller, an IC H bridge, etc -- all these different things can be wrapped up as separate modules, that is to say classes. Imagine this: Each algorithm class knows what input it takes -- position or speed, the two that can be currently sent by base station -- and what it should output to the motors -- also speed or position, some motor-controlling devices take one or the other. If it knows those two things, then it can take in an input value, decide what it wants the motor to do, and then returns either a speed or position value to be pushed onto the motor. Meanwhile, each motor-controlling-device-representing class shall contain the information for actually making the desired movement happen. It shall know what input it takes -- speed or position -- and that’s all it needs to know. From there, it’s free to convert the input into the actual physical action to make the motor move.

By lacing these classes together with a commonly accepted input -- speed from -1000 to 1000, or position from 0 to 10000 are the default values for now -- then these classes NEVER must know what each other is actually doing. You can make a new type of motor-controlling-device class for each new one we buy/each new one the mechanical team needs us to use, and just immediately plug it into the framework with minimal effort. Just have the new device class take in either the speed or position values -- the device wanting just one or the other -- and then the one programming that class doesn’t have to worry about where it’s coming from or how the algorithm decided on that value. So, we write the class for that device and just throw it into the framework, the existing algorithms will spit out a value the new device class knows how to use and everything just works together immediately.

Software implementation of the algorithm, overview:

The architecture is composed of four classes, generally speaking.

1)An input-output-algorithm class. Abstract superclass is called IOAlgorithm. It’s responsible for figuring out how to convert between the main program’s input, and an output demanded by the physical motor controller that will actually move the motor, as well as any control loops. It expects an input of a specific type -- either speed or position -- and outputs either speed or position. Internally, it does whatever it needs to based on its control loop logic. For example, for something like IK controls where the main program wants the motor to go to a certain position, we’ll have a class derived from the overall IOAlgorithm abstract class that inputs a position, reads from an encoder what position the controlled part of the arm is actually at, figure out how far this arm joint needs to move to get to the desired position, figures out from that how fast the motor controlling the joint needs to go, and then finally returns a value representing that speed. This class makes use of the next class, feedback device, to get its feedback information if it has any, and thus contains an internal pointer to it.

2)A feedback device class. Abstract superclass is called FeedbackDevice. It contains the information needed to actually get feedback from a device. For example, there should be a class made to represent an encoder sensor if we use one; the class will contain functions for reading signals from the device, and return either speed or position (different types of feedback can be implemented later, if used). It shouldn’t be the job of the IOAlgorithm class to worry about how the feedback is actually communicated, just what it is, which is why this class exists. The class should be constructed by the main program and as such the constructor should be public, and the constructor should contain whatever information is needed to get the feedback such as what pins need to be read from, if any.

3)A motor-controlling device class. Abstract superclass is called OutputDevice. Basically, this represents whatever external device we are actually using to move the motor (yes we’ll always be using some kind of external device to do it, you can’t move a motor just with a microcontroller. Exception might be a servo, we’ll just create a derived class called ‘OutputServo’ or something). The class is responsible for knowing how to communicate to the external device in order to get the motor actually moving, and to get it to move based on the value the class is sent. The two types of input these class can currently expect to take are either speed or position. The class should be constructed by the main program and as such the constructor should be public, and the constructor should contain whatever information is needed to get the feedback such as what pins need to be read from, if any.

4) An external interface representing this arm joint overall – JointInterface -- for the main program to interact with; this joint class was designed so that it’s all the main program needs to call to cause an output to be sent to any device the main program wants to move. The joint interface acts as something of an overseer of the other two or three classes, internally. It makes sure that the algorithm and controller device work together. There are three derived JointInterface classes that are used to represent different joints. There is one joint interface class used to represent a joint that’s controlled by a single motor device, one that represents a joint that’s controlled by two motor devices working together, and one that represents a joint that’s controlled by two motor devices moving in opposite directions. The joint classes have at least two constructors. Firstly, it takes in the motor controlling device class being used, and it takes in an enum value representing what kind of input the main program is going to give it, currently either speed or position. In this constructor’s case, the joint interface then takes these factors, and pics out what algorithm should be used – these algorithms will all be open loop, so they can be simple and constructed internally to the joint interface. The other constructor meanwhile deals with the more complex algorithms, where closed loop is implemented. Constructor B takes in input type, the IOAlgorithm class representing the algorithm to be used on this joint, the output devices being used on this joint, and finally what feedback devices are being used to inform the closed loop control (if there is no feedback device being used, then it’s not closed loop, and this constructor isn’t the one that should be called). Then when the user calls the runOutputControl method internal to this class, it passes the value to the algorithm class, and gets back the modified value to be passed to the output device class. From there, if the joint is for a single motor joint, it just passes the value into the output device class for it to use. If it’s a ‘two motors working in tandem’ joint class, it passes the value into both output device classes. If it’s a ‘two motors working in opposite directions’ joint class, then it passes the value into one output device class and inverts it for the other output device class.

Example) let’s say the user calls up the interface class and demands that joint A moves. Let’s say Joint A is a single motor controlled joint, controlled by Motor A. Let’s say this motor uses a discrete H bridge to control it, which moves a motor based on speed. Let’s say the microcontroller would not get feedback for the motor. We’d select the single motor joint class. The joint interface class would take these three factors -- controller type, input type, feedback -- and chooses to call the algorithm built for converting the user’s demanded speed to the controller’s input speed without feedback. This algorithm probably won’t do much, since it’s already in speed form. Then it calls the motor controller class for a custom H bridge and tells it to move at the calculated speed. The motor controller class takes the speed, converts it into a pwm signal since that’s what’s used to control a custom H bridge, and outputs that pwm signal on the pin that’s connected to the custom H bridge, either its forward or reverse pin based on whether or not the speed indicated to go forward or backward.

Another example: If the user calls up the device interface and demands that joint A moves to a certain position, and that motor A still has a discrete H bridge controlling it and that the microcontroller will get feedback from the motor using encoder type B. The user constructs the joint interface by passing it position input type, whatever IOAlgorithm is used to do position-to-speed closed loop control, and the feedback device being used. The joint interface then calls that algorithm, passes it the feedback device, and waits for it to return a value to pass to the discrete H bridge. Once the algorithm calculates desired motor speed based on the motor’s current position vs its destination position, it returns this speed. The interface calls the H bridge motor controller class and gives it that speed. The motor controller class then converts the speed into a pwm signal and sends it off on the pin where the H bridge is attached, either forward or backward.

In all these cases, the joint interface’s run output control function should return an enum representing the results of the interface’s attempts to carry out the user’s command. The enum represents states such as the user inputting invalid inputs, if the joint interface was improperly constructed, if the output was successfully ran and the joint is currently attempting to get to its desired end state such as moving to a specified position or accelerating to a specified speed (and as such the control loop should be put into a while loop or something until it’s no longer in transit), or that the output was ran successfully and the joint has reached its desired end state so there’s no need to stay in the control loop for that motion.

Adding new modules to the framework, YOU the programmer.

Remember: IO algorithm classes take in either speed or position with optional feedback device, and output either speed or position. OutputDevice classes take either speed or position, and are constructed publically by the main program with all the hardware info needed to send values to the phyiscal output device. Feedback devices output either speed or position, and are publically constructed by the main program with all the hardware info needed to receive values from the physical feedback device.

When you make a new module for any of these -- if you want to implement a new feedback device in software, a new motor controller thingy, or just make a new algorithm -- have it inherit from the appropriate superclass, IOAlgorithm OutputDevice or FeedbackDevice. These superclasses contain a virtual method you need to override, IOAlgorithm’s int RunAlgorithm(const int, &ret\_ControlComplete), OutputDevice’s void move(const int), and FeedbackDevice’s int getFeedback(); Most contain enums internally for what type of in/out they have, make sure to set that in their constructors if so. Open loop algorithms need to specify that JointInterface is a friend class of theirs. Finally, if you’ve made a new open loop IOAlgorithm (one that doesn’t use a feedback device) then you have to edit Joint Interface’s selector method – the methods that choose an algorithm to use -- so that it knows that it can select your new class, and make sure that it will only select it if your algorithm is appropriate based on what the passed output device wants and what the main program’s input type is. When in doubt, make an inquiry. Also keep in mind accessor levels – the main program should not be able to call most of these class’s methods besides interface, nor the constructors for open loop IOAlgorithms. We want to enforce the architecture’s proper usage.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Classes using other classes diagram. Top classes use bottom classes. Not actual class names, just descriptors** | | | | | | |
| **Class type** | Example implementations | | | | | |
| **Joint Interface** | Single Motor Joint, Tilt Joint, or Rotate Joint | | | | | |
| **Input-to-output Algorithm** | No feedback to microcontroller. Inputs speed, outputs speed | Feedback pos to microcontroller. Inputs position, outputs speed | | | No feedback to microcontroller. Inputs position, outputs position | |
| **Output controller -- physical device interface** | Custom H bridge for brushed DC motor. Takes in speed | Custom H bridge for brushed DC motor. Takes in speed | Motor controller with speed control for brushed DC motor. Takes in speed | Motor controller with speed control for brushless DC motor. Takes in speed | Motor controller with positional control for brushed DC motor. Takes in position. | Motor controller with positional control for brushless DC motor. Takes in position |