Modular Firmware Outline

Oscar Bezi (bezi@cmu.edu)

Contents

1 Introduction

This is the layout for the Carnegie Mellon University Biorobotics Lab's generic C++ firmware libraries. Using the lab's SEASNAKE firmware libraries as a guide, we have implemented a firmware library that uses a flexible build system and modular components to provision other hardware systems with the high-level functionality (such as velocity control or spring-constant estimation). The primary driving factors in our design are modularity, reliability, and ease of adding new hardware platforms. With this in mind, we implemented a **message-based architecture**, described in this document. This is a discussion behind the motivation of the current design architecture, details of its implementation, and a guide to implementing it to a new hardware platform. This document should be considered a work in progress, as it changes with the firmware (see the Changelog, section 5, to see when this edition was last updated).

2 Message-Based Systems

2.1 What is a Message-Based System?

The concept of a message-based system is much deeper and has more far-reaching ramifications than described in this document. To guarantee real-time sustainability and deterministic behaviour, we constrain the pattern to only implement what is needed in this firmware system. More in-depth implementations, such as LCM (also used by the lab) have many more high-level features (most of which are unneccesary to our goals).

A message-based system uses a message board to decouple consumers from providers. Every component has a reference to a central message board, and can read and write to it. Whenever a sensor reads a value, it publishes the latest value to the field allocated to it. Then, a higher level behaviour can read from these fields, perform calculations or report the value, without knowing any implementation details about the original sensor. This is applied to all inter-component interactions.

2.2 Advantages of Messsage-Based Systems

This is advantageous because all communications are now implementation agnostic. What this means is that we can now completely separate low-level driver implementations from higher level behaviours. This allows the firmware to be compiled to many different hardware platforms by swapping the lowest level of driver, which are only responsible for reading from the message board and writing to output pins or fields, making them simpler to write and debug.

2.3 Disadvantages of Message-Based Systems

The primary disadvantage of this system the lack of thread-safety. It is important to note that we do have interrupt routines, which could lead to issues since this system is not locked. Therefore, as a style policy we enforce that these should not write to the board. Instead, they should write to an intermediate register, which then writes to the sensor database during the standard control loop. The driving example for this policy is the situation in which we receive data from a sensor at a faster rate than the control loop. If the sensor writes to the database, it will overwrite previous values, when we more likely would like to run some sort of filter. Otherwise there is no benefit over simply polling once per control loop.

Other disadvantages include increased debugging difficulty. Since there is no call stack, we cannot see in what order elements are writing to the board and what values were previously in the board. There are also complex interactions with potentially many channels. To mitigate this, every provider has a separate channel, which is why address checking is implemented (see the implementation notes).

Lastly, the importance of the (user-defined) order in which we call the `update()` functions of the components is disadvantageous. Ideally, this would happen automatically. This is further discussed in the implementation notes in section 3.3.

3 Our Implementation

3.1 The Message Board Interface

There are three public methods:

- `void set(void* obj, FieldID id, float value);`
- `inline bool has(FieldID id);`
- `inline float get(FieldID id);`

Our message board is implemented in `common/modules/MessageBoard.h/.cpp`. Internally, it uses two arrays of the same size. The first is a permission vector, keeping track of the objects allowed

to write to the board's fields. The second one is a value vector, which actually contains the values, which are of type `float`. There is an `enum` called `FieldID` which has every field in the database. To add a field, simply add a new element to the `enum`. The last element is called `FieldID.numOfFields`, and by the properties of the enum has the size of the database. The arrays are statically allocated to be of that size, and therefore adding fields requires a recompilation. On initialisation of the `MessageBoard`, the every element of the permission vector (of type `void*`) is set to `NULL`. Every element of the value vector is set to 0.

Only one object is allowed to write to a given field in the board. This allows us to make some guarantees to make debugging more straightforward. This is enforced by the `set` function. Whenever we wish to write to the board, we need to pass in a reference to the class we're working with. The first time `set` is called for a particular field, it sets the equivalent element in the permission vector to be the address of the calling object (the first parameter to `set`), registering the field to that object. Then, on subsequent calls, if the first parameter does not match, an error is thrown and the value is not set.

The `has` function returns true if the value has been set at least once, indicating that a publisher exists. This allows us to ensure that a subsequent `get` operation does not return a garbage value. It does so by checking if the field's registered object equals `NULL`. This also allows for a generic module feedback implementation using Google proto buffers.

The `get` function returns the value in the field. It is an inline function that reads from the array, so it does no safety checking (that is what `has()` is for).

3.2 Object Design

Each component in the system extends the `MessageBoardUser` class. Therefore, it must provide a public (possibly empty) `update()` function, and it has a pointer to the global message board (`mb_`), passed in in the constructor. Furthermore, it has implemented the protected `boardSet()`, `boardGet()`, and `boardHas()` functions, which are just wrappers around the database functions. Note that `boardSet()` only takes two parameters, as it passes a reference to `&this` as the first parameter to `mb_.set()`.

3.3 Control Loop

The control loop in `main.cpp` should take care to call the `update()` functions in the correct order. To this end, all components are split into four categories:

- `Sensor`: A hardware dependent input. Reads values from hardware and writes to the sensor database. Order independent. Examples include thermistors (temperature sensor) and encoders (position sensors).
- `VirtualSensor`: A hardware agnostic input. Virtual sensors are read-only systems that use sensor data and data from other virtual sensors to emulate a sensor. Examples of virtual sensors include velocity and torque sensors that use encoder data and a dt to return a specific value.
- `Module`: Hardware agnostic, high level behaviours that take inputs, run calculations, and write to the message board to command the system state for the next iteration of the loop. Examples include a motor controller or spring constant estimation.

- `Actuator`: A hardware dependent output. Read from the message board to set behaviour for this iteration of the control loop. Examples include a motor. These are order agnostic as well.

The loop should run `update()` of every component. The order is up to the programmer, but it is important to make sure they are in a logical order. A checker script could be written using dependency resolution algorithms (see Appendix A). This is future work.

4 Programmer's Guide

This has been a review of the motivation and design considerations behind the current firmware architecture. If you don't read any of this and want to write firmware components, these are the CliffNotes.

- There is a global message board that all systems read and write to.
- Only one system can write to a given field in the board. Anything can read it.
- Rather that interconnecting with object inheritance, interconnect by reading and writing to the board.
- To add to the board protocol, add an element to the `enum` in `modules/MessageBoard.cpp`. Give it a descriptive name that's likely to be unique, since every single subsequent revision of the firmware will have that element in it.
- Your object will have an `update()` function that runs once per control loop. This is the only way to interact, and this should be your only public method unless there is a good reason for more (for example, the LED controller should have an interface for us to manipulate the lights from the control loop as well as from an error handler).
- This is not specific to the our code, but make sure to adhere to style guidelines and to Keep It Simple, Stupid.

5 Changelog

- Fixed some typos, Oscar Bezi (bezi@cmu.edu), 24 June 2014
- Edited with feedback from Florian Enner, Oscar Bezi (bezi@cmu.edu), 12 June 2014
- First draft, Oscar Bezi (bezi@cmu.edu), 11 June 2014