Adaptable House: Anti-Sway
Official Control/Software Manual and Documentation

Espinola, Malachi P
Hokenstad, Ethan T
Neff, Callen P
Nguyen, Tri V
Tevy, Vattanary

University of Washington
Department of Mechanical Engineering

Contents

A. Introduction

1. Project Motive

The Adaptable House Capstone, begun by Mary Meyer, envisions an "adaptable house" capable of helping patients with neuralmuscular disease, disorders, and injuries. The goal is for this house to assist people to walk around with a system that supports them, and adapts to their movements. The Anti-Sway division of this capstone was in charge of proving that lateral movement support could be provided to such users. Thus, at the Univeristy of Washington, the Anti-Sway Capstone group developed, tested and verified a small scale system that could prove that it can be done.

2. Manual Coverage

A small and yet significant portion of this system is the embedded software that forms the cornerstone of the system by providing robust control for patients using the system. This document will thus provide a guide through, and for:

1. Software Design of Existing System
2. Implementation Decisions at Project Time
3. Future Integration for Lift Control

3. Manual Limitations

Although this manual aims to serve as a generous guide through the system, it should be noted that the subjects discussed are beyond trivial, and as such, one should be familiar with both control theory and common software development trends (the latter of which changes rapidly), although this manual will attempt to provide some background.

In particular, it is not a reiteration of every single software function ever defined. Such scrutinizing details are well documented within the code. This manual is here chiefly to explain any complicated algorithms, and the high level concepts captured by the software.

B. Requirements

1. Client-Side Requirements

2. Engineering Specifications

C. Background

1. Introduction to Feedback Control

2. Finite State Machines

3. Software Design: Modularity

4. Parallelism and Mutual Exclusion

5. Microcomputers and C Programs

I. Error Codes

II. Memory Constraints

D. Hardware Specifications

1. Microcomputer

I. MyRIO Wiring Connections

2. Mechanical Frame

I. Coordinate Axis Convention

E. Sensor/Actuator Specifications

1. Angle Sensing: Potentiometers

I. Iterations on Angle Sensing

II. Calibration Method and Results

2. Position Sensing: Encoders

3. Force Actuation: Motor

4. Velocity Actuation: Keypad

F. Anti-Sway Control Modes and Schemes

1. Tracking Mode

2. Anti-Sway Mode

3. Force to Voltage Conversion

G. Embedded Software

The following sections exclude discussion of `main.c`, which technically runs the program via its use of a main method. However, it is realtively simplistic, and simply calls on the system to start. The following sections will then discuss it.

1. System Setup/Shutdown: `setup`

This module is probably the simplest in concept. It is in charge of calling all modules' global startup and shutdown functions, which are one time only functions. The modules that depend on this are the MyRIO FPGA (`MyRio.h`), the Sensor/Actuator Module (`io.h`), and the data recording module (`record.h`), as shown in Figure 1:

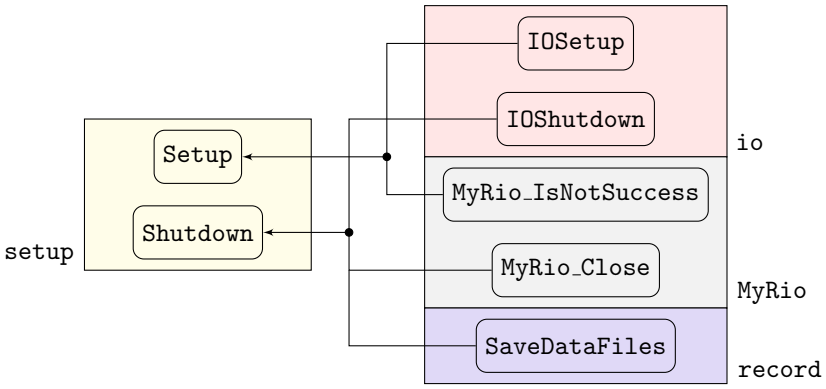


Figure 1: `setup` Module Dependency Diagram

It has two methods (both of which are `extern`):

1. `int Setup()`: Starts the System
2. `int Shutdown()`: Stops the System

which both call the setup and shutdown functionality of the dependent modules, and return the conventional error code to signal if any of the dependent modules fail to initialize/deallocate. In particular, both methods start and stop:

1. The ability to utilize the MyRio's FPGA (via `MyRio`)
2. The ability to interface with Sensors/Actuators (via `io`)
3. (For `Shutdown()` only) Writing all Data to MyRio's Disk (via `Record`)

It is also responsible for initializing the universal error code, which will be discussed later, but is an error signal shared accross threads with specific codes (somewhat related to the C construct of `errno`).

I. Calibration: Setting up `io`

Although it is strange to talk about another module within this context, talking about system initialization through `io` is appropriate here. The `io` module has to know what the reference position and angle are. To do so, it requests that the user set the trolley and the angle to the desired reference, and then sets it, all during the `IOSetup()` method. For reference, the displayed prompt is:

```
Please stablize for calibration.
Press ENTR when ready._
```

More information about this particular method can be found in Section G.3. which will go into the specifics as to how exactly that is done.

2. System Management: `system`

The `system` module is essentially responsible for running the entire software. It obviously does so by deffering basically each "action" to the other modules, so just like the `io` module, it does not really do a lot of heavy lifting. However, as the abstract representation of the system itself, it should be no surprise that its design resembles a Turing Machine (or a View/Controller, if you are familiar with the Model-View-Controller Software Design Pattern). Its interface is a simple function:

- `int SystemExec()`[1]: Executes the System

Now, we shall discuss how the system is structured on a high level. The following Finite State Automata, defined as a Nondeterministic Finite Automata (NFA)[2], shows the state transitions for the System. Let us define the alphabet to be:

$$\boxed{A := \{1, 2, 3, 4, \leftarrow, E\}}$$

where $1 - 4$ are keypad keys, $\leftarrow$ is the keypad delete key, and E is a universal error (with ϵ as the empty string/character). Then:

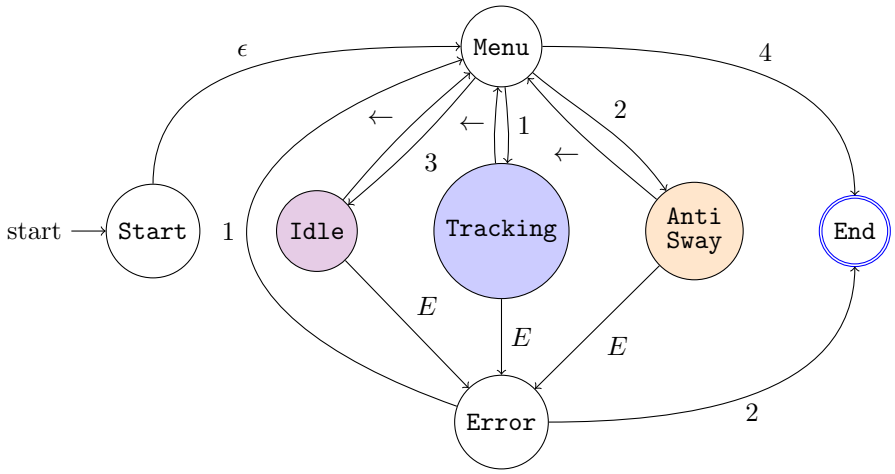


Figure 2: NFA for `system`

Note the correspondence of the 2 anti-sway modes with both the Tracking and Anti-Sway States. The presence of an idle mode is merely to check that sensors are working properly, it is not an actual control mode.

Implementing `system` becomes very straightforward from inspecting Figure 2.. A function is assigned to each state, which are each responsible for indicating the next state to execute as well as executing the appropriate actions for that state, while all `SystemExec()` does is manage these states' execution flow. The dependency diagram for the `system` module is now shown in Figure 3 (next page).

[1]Notice the `Exec` attached to the end of the name, which references the illusion that it is its own program and returns a conventional status code. This function does not actually clone the embedded process, its just a fun naming convention.

[2]This particular version of a Turing Machine is being invoked for simplicity, but also because the input to the `system` is quite literally a senary (base-6) string

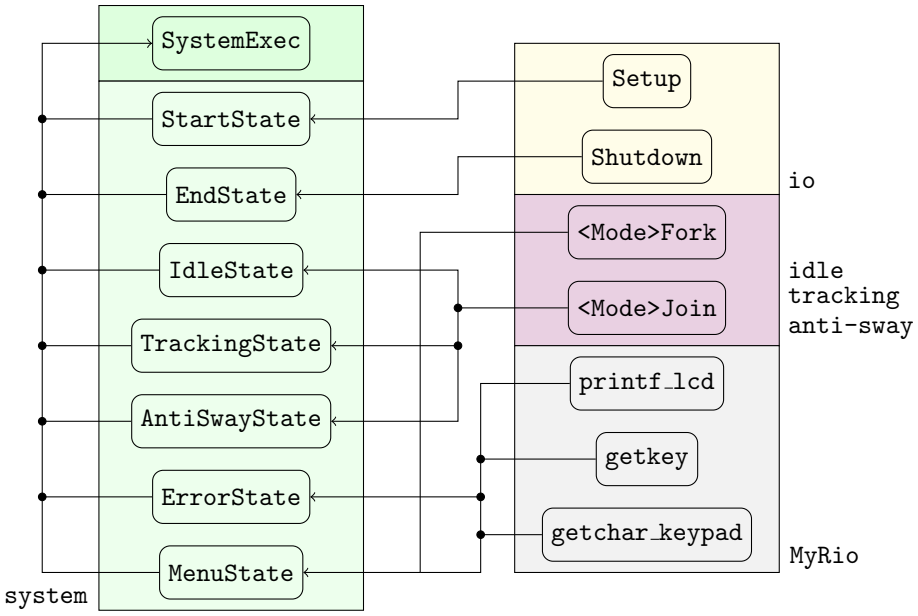


Figure 3: `system` Module Dependency Diagram

The description of such functions follows:

- `static int StartState()`: Dedicated to setting up the System (by setting it up via `Setup()`)
- `static int EndState()`: Dedicated to stopping the System (by shutting it down `Shutdown()`)
- `static int IdleState(), TrackingState(), AntiSwayState()`: Runs the corresponding state, and exits them when ← is pressed or a universal error occurs
- `static int ErrorState()`: Processes a universal error with user input
- `static int MenuState()`: Allows the user to choose any of the Modes, and navigates to it

To acomplish state execution and state transitions, the following data types are employed:

```
enum {
    ANTLSWAY,
    TRACKING,
    IDLE,
    MENU,
    ERROR,
    START,
    END
} state = START;
```

```
static int (* states[])()
    = {AntiSwayState,
       TrackingState,
       IdleState,
       MenuState,
       ErrorState,
       StartState,
       EndState};
```

Thus, any State Function can set the variable `state` using the `enum` constants, and all `SystemExec()` needs to do to continue execution flow is state the line: `states[state]()`.

Future Modification

Notice it is very critical for these fields to align in both the `enum` and the `states` array, and for all functions in the array to have the exact same function signature (save the name of course).

3. Sensor/Actuator Interfacing: `io`

This module is quite complex, although its intention is simple. It is a decorator module over Sensor/Actuator accessor/mutator functions. There are a few reasons for this:

- Ability to convert raw outputs from sensor functions, which are usually in non-usable formats (wrong units).
- Ability to monitor raw inputs into actuator functions, which alone can result in system/physical malfunction.
- Decorator functions provide ease of access and readability to both kinds of devices by reducing the amount of lines written (code reuse).

For each sensors/actuators, a variable of some data type defined by the MyRio I/O interfaces must be employed. Through its lifetime, that data type must have some way to:

1. Initialized (Once)
2. Use
3. Delete (Once)

C-Style Usage

This is a very common design pattern used in C libraries that offer an interface to some data type. More explicitly, the above three functions usually come in the form:

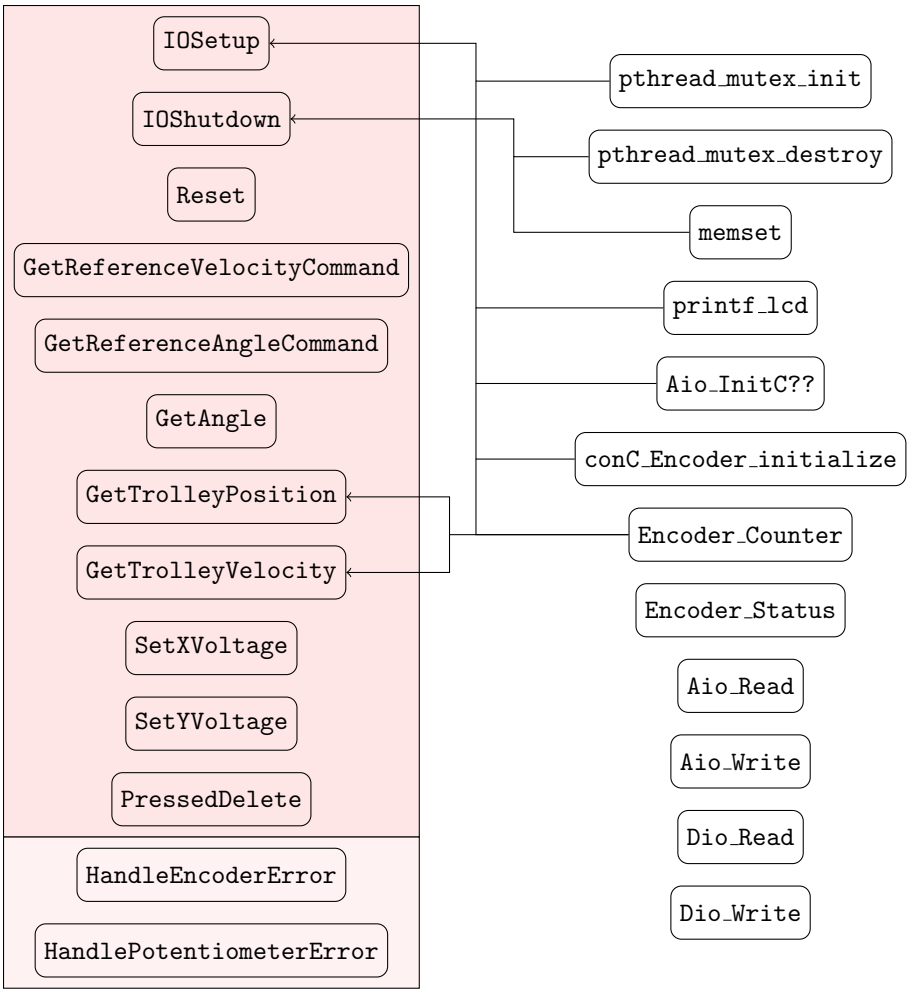
```
// Allocate/Initialize var (of type varType) with some parameters
int AllocVar(..., varType *var);

// Use var with some parameters
int UseVar(varType *var, ...);

// Free/Delete var (usually without parameters)
int FreeVar(varType *var);
```

It resembles object oriented programing with memory managment, but obviously since C isn't object-oriented, it must externally offer functions that do the above 3 on a pointer to the variable of concern. What's more is that this style allows the user to fully control the allocation of the variable. This is important because you'll see this everywhere, including in our code.

Since this is true for all the sensors/actuator we use, you'll find these functions that do such things being used in obvious places. Below is the dependency diagram for the `io` module, which wraps this functionality:



I. Position & Velocity Sensing

II. Smart Limits: Kinematic Limits

III. Keyboard: Multithreaded Resource Managament

4. Discrete Control: `discrete-lib`

I. Control Law Blocks: Control Made Easy

5. Modes: `idle/tracking/anti-sway`

I. Multiaxis Control

6. Thread Management: `thread-lib`

7. Data Requisition: `record`

8. Error Handling: `error`

I. Static Error Codes v.`errno`

H. System Performance

1. Tracking Mode

2. Anti-Sway Mode

I. Discussion

1. Encoder Connections

2. Better Data Acquisition

3. Gradient Descent: Controller Tuning

$$\mathbb{L}(\mathbf{r}, \mathbf{y}) = ||\mathbf{r} - \mathbf{y}||_2^2 = \sum_{t=0}^{T} (r(t) - y(t))^2$$

$$\mathbb{L}(\mathbf{e}) = ||\mathbf{e}||_2^2 = \sum_{t=0}^{T} e(t)^2$$

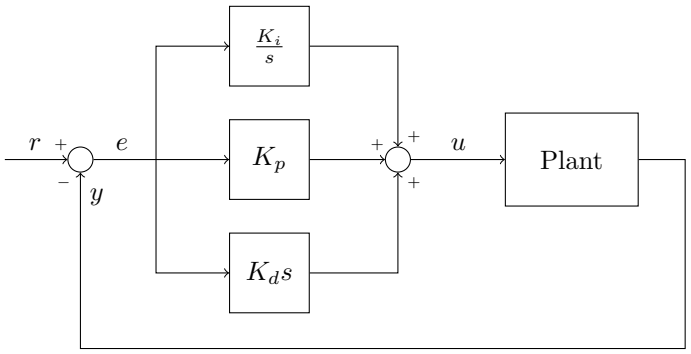
$$\mathbf{K} = \begin{bmatrix} K_i \\ K_p \\ K_d \end{bmatrix}$$

$$\nabla_{\mathbf{K}} \mathbb{L} = \left(\nabla_{\mathbf{y}} \mathbb{L} \nabla_{\mathbf{u}} y + \nabla_{\mathbf{r}} \mathbb{L} \nabla_{\mathbf{u}} \mathbf{r}\right) \nabla_{\mathbf{K}} u$$

$$\nabla_{\mathbf{K}} \mathbb{L} = \sum_{t=0}^{T} \left(2e(t) \frac{\left(\frac{\partial r}{\partial y} - 1\right) \frac{\partial y}{\partial u} + \left(1 - \frac{\partial y}{\partial r}\right) \frac{\partial r}{\partial u}}{1 - K_p \left(\left(\frac{\partial r}{\partial y} - 1\right) \frac{\partial y}{\partial u} + \left(1 - \frac{\partial y}{\partial r}\right) \frac{\partial r}{\partial u}\right)} \begin{bmatrix} \int e(t)d\tau + \frac{\partial}{\partial K_i}\left(K_d \frac{\partial e}{\partial t} + K_i \int e d\tau\right) \\ e(t) + \frac{\partial}{\partial K_p}\left(K_d \frac{\partial e}{\partial t} + K_i \int e d\tau\right) \\ \frac{\partial e}{\partial t} + \frac{\partial}{\partial K_d}\left(K_d \frac{\partial e}{\partial t} + K_i \int e d\tau\right) \end{bmatrix} \right)$$

$$\nabla_{\mathbf{K}} \mathbb{L} = \sum_{t=0}^{T} \left(2e(t) \left(\left(\frac{\partial e}{\partial u} \right)^{-1} - K_p \right)^{-1} \begin{bmatrix} \int e(t)d\tau + \frac{\partial}{\partial K_i}\left(K_d \frac{\partial e}{\partial t} + K_i \int e(t) d\tau\right) \\ e(t) + \frac{\partial}{\partial K_p}\left(K_d \frac{\partial e}{\partial t} + K_i \int e(t) d\tau\right) \\ \frac{\partial e}{\partial t} + \frac{\partial}{\partial K_d}\left(K_d \frac{\partial e}{\partial t} + K_i \int e(t) d\tau\right) \end{bmatrix} \right)$$

$$\mathbf{K}_{n+1} \leftarrow \mathbf{K}_n - \eta \nabla_{\mathbf{K}} \mathbb{L}$$



J. Conclusion

K. Appendix

1. Code Base