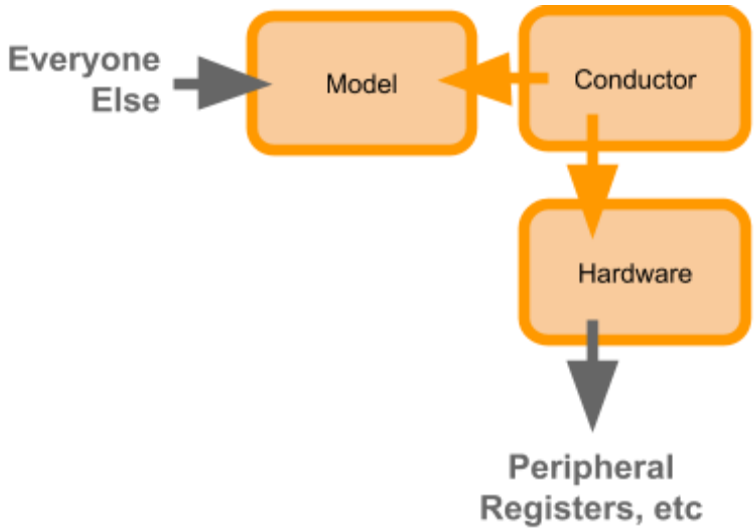
Some Helpful Embedded Design Patterns

There are many widely used design patterns in the world of software engineering. A little searching will bring up hundreds of articles, books, and videos on the topic. If this sort of thing interests you, we definitely support your decision to learn more! For now, though, here is a really quick overview of some patterns that we find especially helpful in the embedded space:

Model Conductor Hardware Pattern

Model Conductor Hardware is a pattern for creating asynchronous hardware interfaces. It promotes good separation between your actual hardware interface and the public interface. Most often that means device drivers that need to run in the background. It could be that they need to be called at a particular periodic rate. It could be because they need to handle on-demand events like incoming data from a communication port. In any case, the goal is to have the public interface (the model) non-blocking, allowing the hardware interface to be processed as needed in the background.



You can create this pattern in Ceedling by specifying MCH

Good For

- Periodic Drivers
- Asynchronous Drivers
- Polling Interfaces

Don't Forget

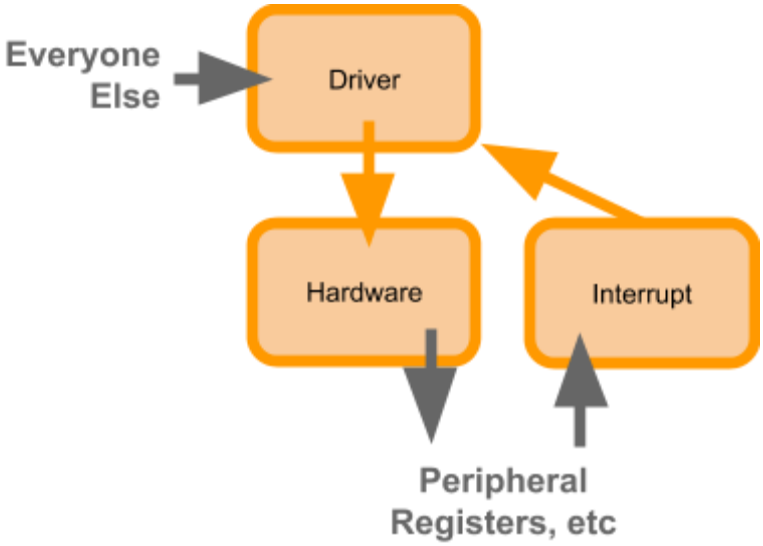
- The public world talks to the Model only
- The Conductor calls the Model and Hardware, not the other way around
- The Conductor is mostly about passing information between the other two
- Hardware can mean registers, a hardware library, or whatever makes sense.

Tips

- Conductor Exec can be called periodically from a main loop OR can be its own loop if using and RTOS

Driver (Interrupt) Hardware Pattern

Driver (Interrupt) Hardware is another pattern for working with hardware. It also promotes a healthy separation between the hardware interface and the public interface. In this case, though, the public interface (the Driver) makes calls directly to the Hardware interface. The Interrupt, which is an optional part of this pattern, makes direct calls back to the Driver as needed. Because of the directness of their relationship, this pattern is good for situations where calls will either return immediately OR it's okay to block on hardware events.



You can create this pattern in Ceedling by specifying DH or DIH, respectively

Good For

- Blocking Drivers
- Immediate Drivers

Don't Forget

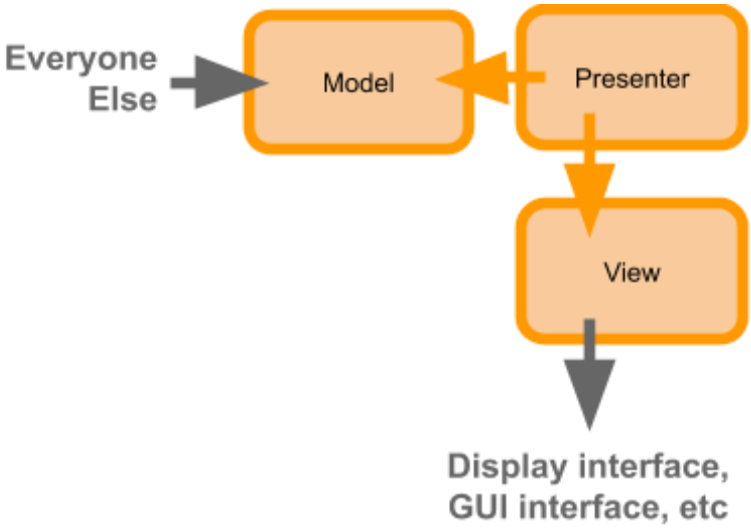
- The public world talks to the Driver only
- The Driver calls the Hardware, not the other way around
- All stored data should happen in the Driver
- Hardware can mean registers, a hardware library, or whatever makes sense.

Tips

- Use a callback for the Interrupt's call to Driver so that a change of hardware will not require a change to Driver

Model View Presenter

Model View Presenter was the original inspiration for MCH. It comes from the world of GUI's, which is worth mentioning as more and more embedded devices have some sort of GUI (You interface might be as complex as a touchscreen and a high resolution display, or it might be as simple as a button and an LED). All data and configuration is stored in the Model, which is what interacts with the rest of the world. The presenter takes data from the Model, sorts and formats it appropriately, and tells the View what to show. It also collects inputs from the View and updates the Model as needed. While we don't use it in this class, there is a lot of information online about MVP, if you'd like to learn more.



You can create this pattern in Ceedling by specifying MVP

Good For

- Embedded GUI's

Don't Forget

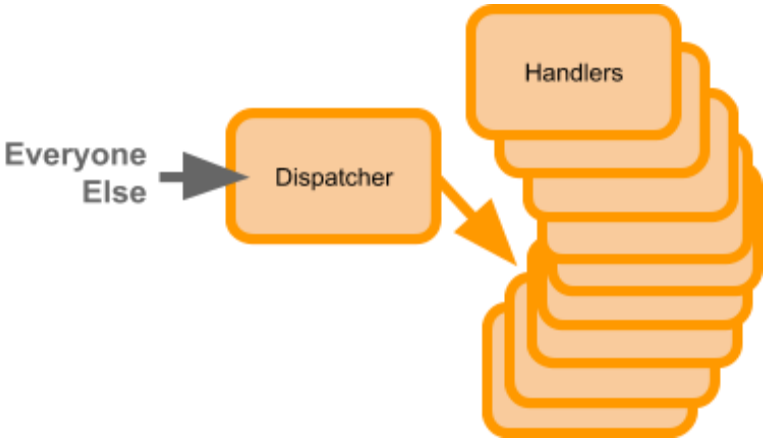
- The public world talks to the Model only
- The Model stores all necessary data and configuration.
- The Presenter calls the Model and the View, not the other way around.
- The View is the interface to screens, led's, buttons,etc.

Tips

- Many frameworks support the Presenter making links between Model and View

Dispatch Pattern

The Dispatch Pattern is useful for triggering a single function handler from a collection of handlers, based on some sort of incoming ID. That ID could be numerical, a letter, a string, or just about anything else. The ID is passed into the Dispatcher module, which then uses it to determine which Handler to run. It runs it to completion and then returns the results. There are asynchronous Dispatchers also, but that usually is done by combining this pattern with MCH.



The actual dispatch mechanism used internally has MANY options. A few have been listed below. The primary concerns are usually number of handlers, sparsity of ID's, memory constraints, and algorithm speed. Search for Search Algorithms for help on the topic.

Good For

- Command Handlers
- Event Handlers
- Exclusive Operations

Don't Forget

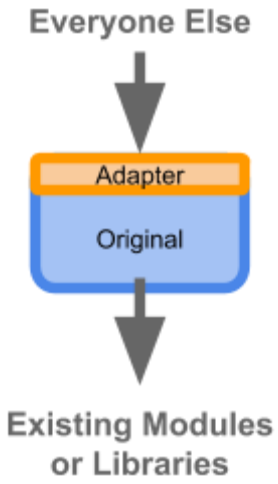
- Dispatcher itself shouldn't DO anything other than fire off the Handler
- Dispatcher can be scaled to support more or different handlers by changing the Dispatcher itself, without any need to change the caller or Handlers.

Tips

- Choosing the best internal algorithm is a complex topic. Start with the simplest one that will work and adjust as needed.

Adapter Pattern

The Adapter Pattern, sometimes called a wrapper, is useful for creating a thin layer around a module to make it act differently to the rest of the world. For example, you might create an LED driver, which basically is just calling particular digital outputs, but you've created a useful abstraction that can then be called throughout your program. It's also useful for making one module act like another. If a program was written to support a serial port, but eventually the serial port was replaced by a USB interface, an Adapter could be put in place to make the USB interface act like a serial port to the rest of the application. It's about changing one API to look like another.



Good For

- Abstracting Hardware Interfaces for your Application
- Using one Module in Place of Another.

Don't Forget

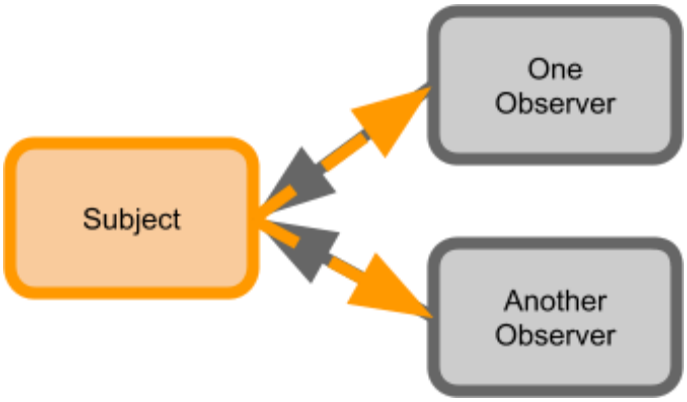
- You don't need to modify the original module… it's the point of the Adapter to make it ACT differently.
- You can use inlines or macros when things line up well.

Tips

- Your goal is to keep the Adapter fairly thin, but that doesn't mean it's all single functions calling another function. It's about behavior!

Observer Pattern

The Observer Pattern is most often referred to as “an event system” in the embedded world. In this pattern, an object (often called the Subject) allows other modules to register to be notified of changes in information. When the information changes, any registered modules are notified of the change as soon as possible, allowing them to act on the new knowledge.



Good For

- Synchronizing information between modules with minimal coupling.
- Acting on state changes in other modules.
- On To Many Relationships

Don’t Forget

- Other than registration, there is very little connection between the two modules.

Tips

- Callback functions are a good way to handle this in C.
- If you’re using an RTOS, semaphores or queues are handy too.