# SOLID Design for Embedded C

Embedded Systems Conference, San Jose, CA, May 2012
Class ESC-231
By James W. Grenning

The most common design pattern I've seen in C code is the function-call data-structure free-for-all. Not all C code is a mess, but it sure seems that there is a lot of it. It's not easy, but C code does not have to be a mess. Good designs are modular, and the modules have high cohesion and loose coupling. We've heard those terms for years, but what do they mean?

To build good designs, we have to change the usual way of design evaluation from Not Invented Here (NIH) to using SOLID design principles. The SOLID design principles give us some specific things to look for in a design to develop modules with high cohesion and loose coupling. The five design principles, described in Bob Martin's book (Agile Software Development, Principles, Patterns, and Practices [Mar02]), spell the word SOLID.

| | |
|---|---|
| S | Single Responsibility Principle |
| O | Open Closed Principle |
| L | Liskov Substitution Principle |
| I | Interface Segregation Principle |
| D | Dependency Inversion Principle |

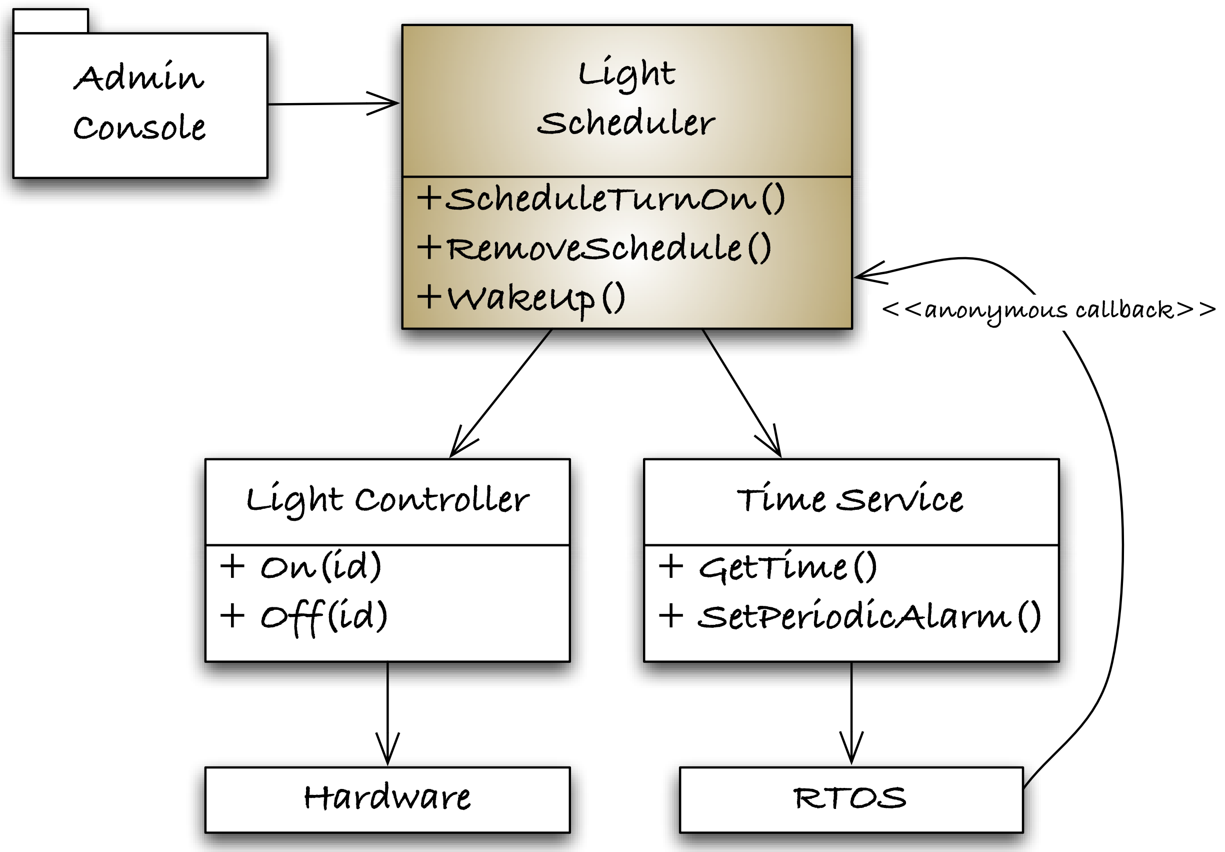
Let's look at the SOLID design principles, which are tried-and-true principles that help build better designs. They come from the Object Oriented world, but there is no reason we cannot apply them and get benefit from them when programming in C. We'll look at examples of code using these principles.

As it turns out, making code that is unit testable leads to better designs. Testable code has to be modular and loosely coupled. In my book, Test-Driven Development for Embedded C [Gre11], I go into how Test-Driven Development can help to steer a design, but in this paper, we'll mainly look at some of the ways to structure C code to build designs that can pass the test of time. Let's start by looking at the principles and them some design models in C to implement them.

## Single Responsibility Principle

The Single Responsibility Principle (SRP) [Mar02] states that a module should have a single responsibility. It should do one thing. It should have a single reason to change. Applying SRP leads to modules with good cohesion, modules that are made up of functions and data with a unified purpose—in a nutshell, modules that do a job and do it well.

The module that accesses numerous data structures and globals, doing all the work, is not following SRP. This diagram, which is part of a design of a home automation system illustrates SRP, separating different concerns:



The admin console subsystem can tell the LightScheduler module to turn on or off a light at a scheduled time. It has the responsibility of managing the light schedule. The LightController interacts with the hardware that can turn on or off some light by its ID. The TimeService has the responsibility of providing the time and periodically waking its client through a callback mechanism.

When modules are well-named with well-named functions, responsibilities should be clear. There should be little need for complex explanations. The modules, along with their tests, tell their story.

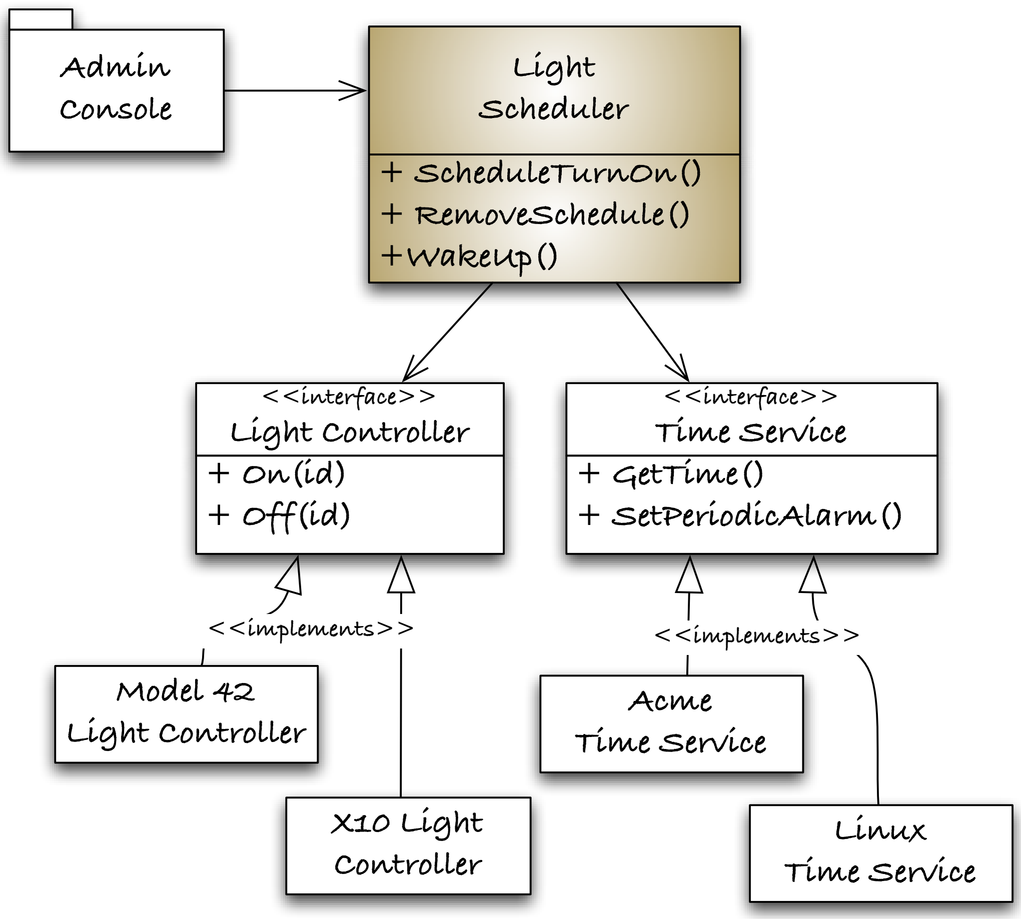
You can apply SRP to functions. Well-focused responsibilities help you recognize where changes should be made as requirements evolve. When this principle is not followed, you get those 1,000-line functions participating in global function and data orgies.

## Open Closed Principle

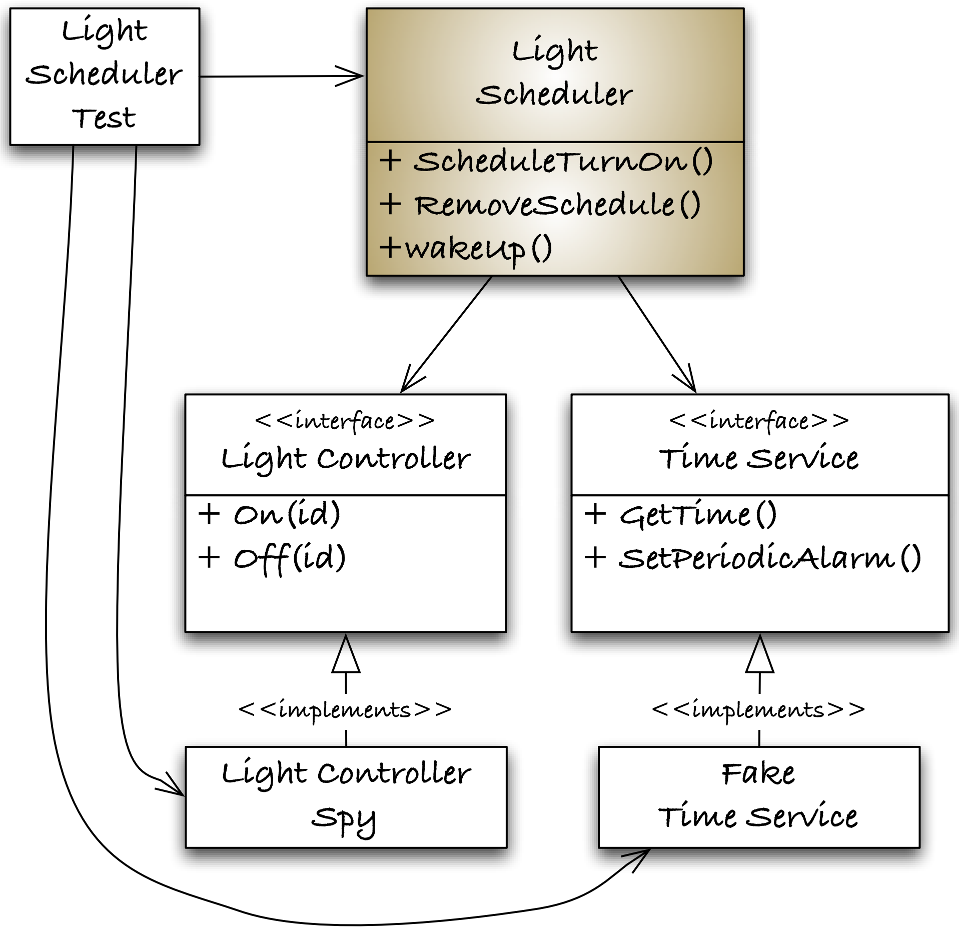
The Open Closed Principle (OCP), described by Bertrand Meyer in Object-Oriented Software Construction [Mey97] and interpreted by Bob Martin, says that a module should be “open for extension but closed for modification.”

Let me explain OCP by metaphor: a USB port can be extended (you can plug any compliant USB devices into the port) but does not need to be modified to accept a new device. So, a computer that has a USB port is open for extension but closed for modification for compliant USB devices.

When some aspect of a design follows the OCP, it can be extended by adding new code, rather than modifying existing code. We can say that the LightScheduler (from the previous example) is open for extension for new kinds of LightControllers. Why? If the interface is obeyed, the calling code (the client) does not care about the type of the called code (the server). OCP supports substitution of service providers in such a way that no change is needed to the client to accommodate a new server.  This diagram illustrates that LightScheduler can work, unmodified, with Model 42 and X10 LightControllers as well as Acme and Linux versions of TimeService.



A design that follow OCP and SRP is more testable.  During test we can create a build that provides a test stub like this:



This means we can't let hardware or OS implementation knowledge make its way into the LightScheduler.  This design allows substitutability of depended upon modules.  In C, the header file is the interface and the C file is the implementation.  We can use the linker to substitute in different version of LightControllers and TimeServices. But there is more to substitutability than having the same interface.

## Liskov Substitution Principle

The Liskov Substitution Principle (LSP) was defined by Barbara Liskov in her paper Data Abstraction and Hierarchy [Lis88]. Paraphrasing her work, LSP says that client modules should not care which kind of server modules they are working with. Modules with the same interface should be substitutable without any special knowledge in the calling code.

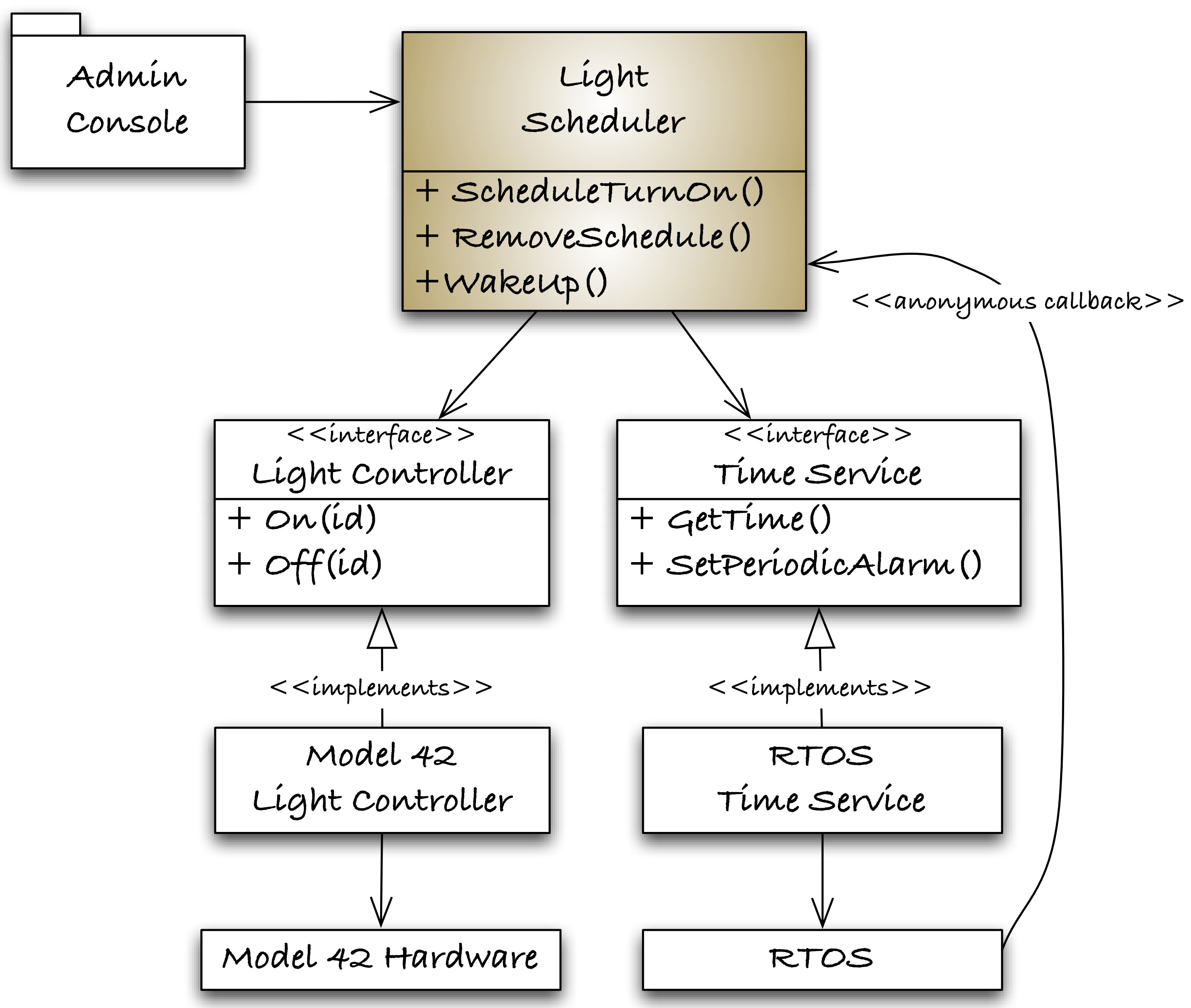
As long as the LightScheduler does not have to behave differently when interacting with the test stub, LightControllerSpy, the design adheres to LSP.

The Liskov Substitution Principle may sound a lot like the Open Closed Principle. That's because OCP and LSP are two sides of the same coin. But there is more to LSP than just having an interface that links or a compatible function pointer type. The meaning of the calls must be the same. The expectations of both the client and the server must be met.

Nothing additional is required from the LightScheduler when it interacts with a LightControllerSpy or a production LightController. No additional preconditions must be established, and no postconditions are weakened.  The LightControllerSpy and LightController are not only syntactically substitutable but are semantically substitutable from the LightScheduler perspective.

## Interface Segregation Principle

The Interface Segregation Principle (ISP) [Mar02] suggests that client modules should not depend on fat interfaces. Interfaces should be tailored to the client's needs. For example, the TimeService, has a very focused interface. It only reveals the operations needed by the application.



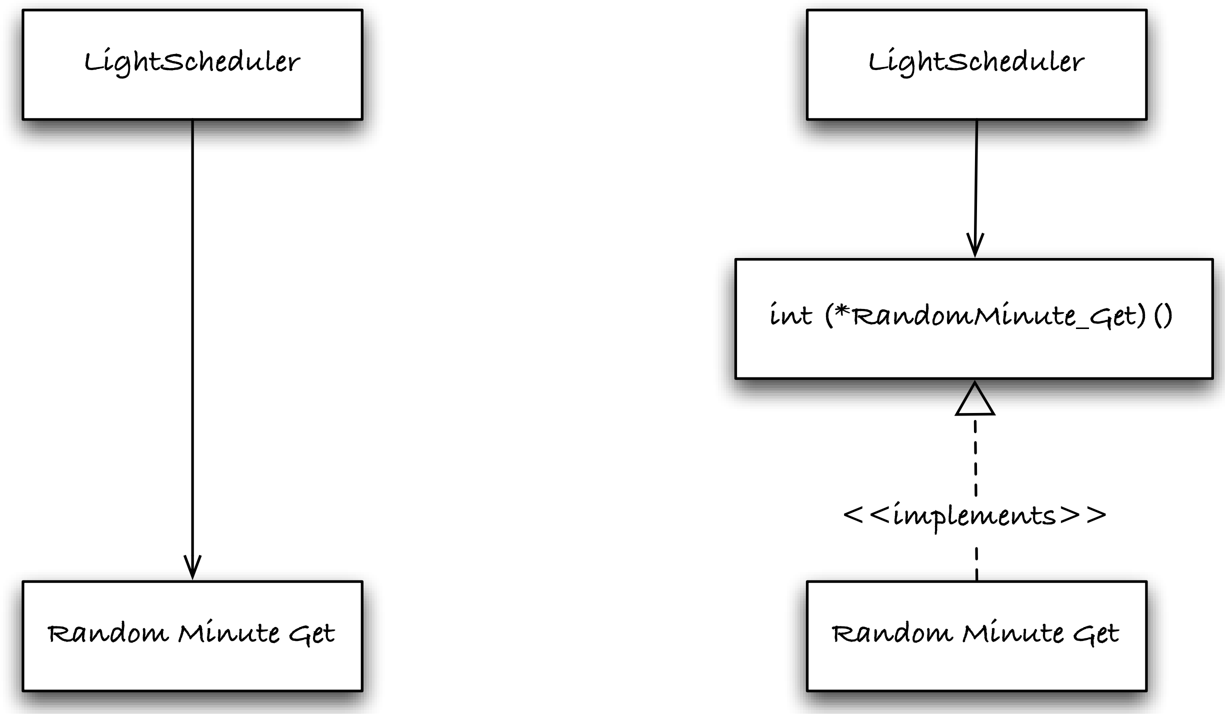
There may be many more time-related functions in the target operating system. The target OS tries to be everything for every application, while the TimeService is focused on the needs of this system. By tailoring interfaces, we limit dependencies, make code more easily ported, and make it easier to test the code that uses the interface.

## Dependency Inversion Principle

In the Dependency Inversion Principle (DIP) [Mar02], Bob Martin tells us that high-level modules should not depend on low-level modules. Both should depend on abstractions. He also says that abstractions should not depend on the details. Details should depend on abstractions. We can break dependencies with abstractions and interfaces.

In C, DIP is typified when we use a function pointer to break an unwanted direct dependency. On the left in this diagram, the LightScheduler depends directly on

RandomMinute_Get. The arrow points to the dependency. The high level depends directly on the details.



The right side of the figure shows an inverted dependency. Here the high level depends on an abstraction, which is an interface in the form of a function pointer. The details also depend on the abstraction; RandomMinute_Get( ) implements the interface.

Operating systems use the same mechanism to keep the OS code from depending directly on your code. A callback function is a form of dependency inversion.

Dependency inversion in C does not have to involve function pointers. In C, it's almost more a state of mind. When we we look at the Single Instance Module pattern later in the paper, we will hide data structures inside the C file while only revealing the name of the structure. There we are applying DIP.

We use DIP when:
• implementation details hide behind an interface,
• the interface does not reveal the implementation details,
• a client calls a server through a function pointer,
• a server calls a client back through a function pointer

## SOLID C Design Models

The SOLID principles can give some guidance for avoiding the data structure function call free-for-all all too prevalent in C programming. Now we'll look at more techniques

for applying these ideas in C. All the examples illustrate SRP and DIP, so I won't call those out.
Each model is more complex than the previous model. Each solves some specific design problem at the cost of some added complexity. You can decide whether the complexity is worth it as we work through some examples. Here are the four models we'll look at:

| Model | Purpose |
|---|---|
| Single-instance Abstract Data Type | Encapsulates a module's internal state when only a single instance of the module is needed |
| Multiple-instance Abstract Data Type | Encapsulates a module's internal state and allows multiple instances of the module's data |
| Dynamic interface | Allows a module's interface functions to be assigned at runtime |
| Per-type dynamic interface | Allows multiple types of modules with the same interface to have unique interface functions |

Each model is more complex than the previous model. I suggest that you choose a model that is the simplest that works for your needs. As things change, and you employ SOLID in your designs (and add automated tests) you will find your code is much softer and more flexible.

**Single-instance module**
For single-instance modules, the header file defines everything needed to interact with the module. The LightController header would only contain these function prototypes.

```
void LightController_Create(void);
void LightController_Destroy(void);
void LightController_TurnOn(int id);
void LightController_TurnOff(int id);
```

Anything that can be hidden should be hidden. The data structures that the scheduler needs to do its job are hidden as file scope variables in the .c file. The scheduler's data structures are not needed in the header because no other modules should care. This makes it impossible for other modules to depend on the structure and assures its integrity is the scheduler's responsibility. If enums or #defines needed to interact with the module they would go into the header file (but they are not in this case).

**Multiple-instance module**
Sometimes an application needs several instances of a module that contains different data and state. For example, an application might need several first-in first-out data

structures. A CircularBuffer is an example of a multiple-instance module. Each instance of CircularBuffer may have its own unique capacity and current contents. Here is what the interface to the CircularBuffer looks like:

```
typedef struct _CircularBuffer * CircularBuffer;

CircularBuffer CircularBuffer_Create(int capacity);
void CircularBuffer_Destroy(CircularBuffer);
BOOL CircularBuffer_IsEmpty(CircularBuffer);
BOOL CircularBuffer_IsFull(CircularBuffer);
BOOL CircularBuffer_Put(CircularBuffer, int);
int CircularBuffer_Get(CircularBuffer);
int CircularBuffer_Capacity(CircularBuffer);
void CircularBuffer_Print(CircularBuffer);
BOOL CircularBuffer_VerifyIntegrity(CircularBuffer);
```

This is a well-established design model based on Barbara Liskov's abstract data type [Lis74]. The members of the CircularBufferStruct are not revealed in the header file. The typedef statement declares that there is a struct of a given name but hides the members of the struct to users of the interface. This prevents users of the CircularBuffer from directly depending upon the data in the struct. The struct is defined in the .c file, hidden from view.  Not that it is relevant, here is what the structure would look like defined near the top of the .c file.

```
typedef struct _CircularBuffer
{
    int count;
    int index;
    int outdex;
    int capacity;
    int* values;
} _CircularBuffer;
```

## Dynamic interface

In a dynamic interface, we are solving the problem of duplicate conditional logic.  Let's say that your application has numerous light controlling hardware implementations.  It's likely that your code has a data structure for each type of LightDriver.  There's probably an enum or set of #defines like this:

```
typedef enum  LightDriverType
{
    TestLightDriver,
    X10,
    AcmeWireless,
    MemoryMapped
 } LightDriverType;
```

Also there would be a struct that all the specific LightDriver types would include as their first member like this:

```
typedef struct LightDriverStruct
{
    LightDriverType type;
    int id;
} LightDriverStruct;
```

Here’s and example usage for a specific strucure:

```
typedef struct X10LightDriverStruct
{
    LightDriverStruct base;
    X10_HouseCode house;
    int unit;
    char message[MAX_X10_MESSAGE_LENGTH];
} X10LightDriverStruct;
```

All that is fine, until we get to the usage of that data.  Here is how the LightControler would turn on lights.

```
void LightController_TurnOn(int id)
{
     LightDriver driver = lightDrivers[id];

     if (NULL == driver)
         return;

     switch (driver->type)
     {
          case X10:
               X10LightDriver_TurnOn(driver);
               break;
          case AcmeWireless:
               AcmeWirelessLightDriver_TurnOn(driver);
               break;
          case MemoryMapped:
               MemMappedLightDriver_TurnOn(driver);
               break;
          case TestLightDriver:
               LightDriverSpy_TurnOn(driver);
               break;
          default:
          /* now what? */
               break;
     }
}
```

You can see with this approach that there will be a very similar function for turning off a light, or destroying the driver. Later when more light operations are needed (like dimming and strobe), more switch statements will be needed. This duplication is bad and makes a mess of the code and an opportunity for errors.

How the dynamic interface helps sovle this problem is by allowing the driver functions to be set at runtime. Instead of direct function calls, the driver functions are called through function pointers. The interface looks like this:

```
void LightDriver_Create(void);
void (*LightDriver_Destroy)(void);
void (*LightDriver_TurnOn)(int id);
void (*Lightriver_TurnOff)(int id);
```

During initialization or configuring the pointers could be set, eliminating the need for the duplicate switch statements. There would be a single switch statement that sets up the pointers.

Having function pointers is very convenient for test purposes also. A test stub version of the driver functions can be dropped into the function pointers, allowing the test code to monitor the light operations. Unlike the LightControllerSpy_TurnOn function in the switch statement, now there would be no dependency on the test code in the production code. That dependency is inverted.

A single set of function pointers works fine for when the same functions are used for each driver type. Although, when there can be multiple supported drivers concurrently, a different solution is needed.

**Per-type dynamic interface**
When we have to support multiple drivers concurrently, then we need the per-type dynamic interface. We'll need a structure that holds a set of function pointers like this:

```
typedef struct LightDriverInterfaceStruct
{
    void (*TurnOn)(LightDriver);
    void (*TurnOff)(LightDriver);
    void (*Destroy)(LightDriver);
} LightDriverInterfaceStruct;
```

Put the struct in a file called LightDriverPrivate.h. It is needed by all the different kinds of LightDrivers, but not the users of the LightDriver. Code that is not a LightDriver implementation should not include that file. There is no stopping them, but still, that's what they should do.

To test the light driver I'd write a test like this:

```
TEST(LightDriverSpy, On)
{
     LightDriver lightDriverSpy = LightDriverSpy_Create(1);
     LightDriver_TurnOn(lightDriverSpy);
     LONGS_EQUAL(LIGHT_ON,  LightDriverSpy_GetState(1));
}
```

Notice that the spy is created for light ID number 1. This would initialize the driver and the function pointers so that when LightDriver_TurnOn is called, the spy's turn on function is called. The spy remembers that it was called and you can tell by getting the state it has saved for light number 1.

Here is the LightDriver data structure that supports the per-type dynamic interface:

```
typedef struct LightDriverStruct
{
    LightDriverInterface vtable;
    const char * type;
    int id;
} LightDriverStruct;
```

LightDriverInterface was defined on the previous page. The name vtable is borrowed from C++.  Virtual functions in C++ work similarly to this.  The vtable is initialized like this and stored in a file scope variable.

```
static LightDriverInterfaceStruct interface =
{
    turnOn, turnOff, destroy
};

LightDriver LightDriverSpy_Create(int id)
{
    LightDriverSpy self = calloc(1, sizeof(LightDriverSpyStruct));
    self->base.vtable = &interface;
    self->base.type = "Spy";
    self->base.id = id;
    return (LightDriver)self;
}
```

The three functions (turnOn, turnOff, and destroy) are also file scope functions.  The spy versions are implemented as follows:

```
static int states[MAX_LIGHTS];
static int lastId;
static int lastState;

static void update(int id, int state)
{
     states[id] = state;
     lastId = id;
     lastState = state;
}

static void turnOn(LightDriver base)
{
     LightDriverSpy self = (LightDriverSpy)base;
     update(self->base.id, LIGHT_ON);
}

static void turnOff(LightDriver base)
{
     LightDriverSpy self = (LightDriverSpy)base;
     update(self->base.id, LIGHT_OFF);
}

static void destroy(LightDriver base)
{
     free(base);
}
```

Finally, the LightDriver_TurnOn function looks like this:

```
void LightDriver_TurnOn(LightDriver self)
{
     self->vtable->TurnOn(self);
}
```

It's kind of hard to look at, and error prone, so it is good it is hidden behind the scenes. Also there is no need to duplicate that in clients of the driver.

The really safe way to initialize the LightDriverInterfaceStruct, if your compiler supports it is:

```
static LightDriverInterfaceStruct interface =
{
     .Destroy = turnOn,
     .TurnOn = turnOff,
     .TurnOff = destroy
};
```

ANSI compilers don't support the named field initialization. You need a C99 compiler for that. As long as we are looking at the safest way to do these things, here is the safest way to dispatch through a vtable:

```
void LightDriver_TurnOn(LightDriver self)
{
     if (self && self->vtable && self->vtable->TurnOn)
          self->vtable->TurnOn(self);
}
```

When combining named field initialization with the above, you could add a new function pointer to the LightDriverInterfaceStruct and it would be initialized to the null pointer value for all initializers that don't mention it. So if we added the Strobe function, and it's not supported by all implementations, there is no work to do. Calling LightDriver_StrobeOn would have no effect because its function pointer is null.

## What Model to Use?

I mentioned this earlier, use the simplest model that works for your current needs. Keep your code modular and when you must evolve it, it won't be so hard. You can see more complete evolution of this example in my book *Test Driven Development for Embedded C*.

## How Much Design Is Enough?

At the start of a new development effort, there is considerable uncertainty. There are unknowns in hardware, software, product goals, and requirements. How can we get started with all this uncertainty? Isn't it better to wait? If you wait, there really is no end to the waiting, because certainty will never come. So, it is better to get started sooner even though some things will get changed later.

I am not suggesting that you don't think ahead. It is impossible to not think ahead, but you can choose what you will act on now vs. what you will act on later. There is a thin line between thinking ahead and analysis paralysis. When you start piling guesses on top of guesses, consider that you've gone too far ahead, and it's time to try the ideas in code.

When there is uncertainty in the hardware/software boundary, you can start from the inside by solving the application problem, working your way to where application code can articulate what it wants from the hardware. Create an interface that provides exactly the services the application needs from the hardware. The LightScheduler/LightController relationship is an example of this. The LightController became part of our hardware abstraction layer.

A nice side effect of the application driving the interface is that hardware implementation details are less likely to pollute the application's core. The LightScheduler knows nothing about X10 or any of the other drivers, and that's a good thing.

We saw in the LightController to LightDriver evolution that as requirements became more clear, the design had to evolve. That's no failure; that's good news that we've learned more. Evolving requirements led to changing design. The problem with much of the legacy code out there today is that as requirements evolved, designs were not improved to more naturally accept the changes.

We can't anticipate all the coming product changes; that is why we have to get good at design evolution. Underlying many of these ideas are the Extreme Programming Rules of Simple Design based on Kent Beck's book Extreme Programming Explained [Bec00]

Let's look at them and see how they help us keep the design good for today's requirements.

**XP Rules of Simple Design**

1. Runs all the tests. The code must do what is needed. Why bother if it does not?
2. Expresses every idea that we need to express. The code should be self-documenting, communicating the intention of the programmer.
3. Says everything once and only once. Duplication must be removed so that as things change, the same idea does not have to be changed in numerous places.
4. Has no superfluous parts. This final rule prevents us from putting in things that are not yet needed.

The rules are evaluated in order. Rule 1 rules them all. If the code does not meet its intended purpose, demonstrated by passing tests, the code is not valuable to anyone. Rules 2 through 3 help with the maintainability of the code, keeping it clean with a design fit for today's requirements. Rules 2 and 4 speak for themselves, but rule 4 is a little more difficult to understand.

The fourth rule tells us to not over-engineer the design. It should be perfect for the currently implemented features. Adding complexity early delays features and integration opportunities. It wastes time when the premature design is wrong. Carrying around unused or unneeded design elements slows progress. Designs always evolve. We need to be good at keeping the design right for the currently supported features.

That fourth rule may be the hardest to follow for people new to TDD. Like I said before, it's OK to think ahead; just be careful what you action. Let the tests pull in the design elements when needed. Having the tests as a safety net makes this a very practical and productive way to work.

## Bibliography

| | |
|---|---|
| [Bec00] | Extreme Programming Explained: Embrace Change |
| [Gre11] | Test-Driven Development for Embedded C |
| [Lis74] | Change. Addison-Wesley, Reading, MA, 2000. |
| [Lis88] | Barbara Liskov. Data abstraction and hierarchy. SIGPLAN Notices, 23(5), May 1988. |
| [Mar02] | Robert C. Martin. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs, NJ, 2002. |
| [Mey97] | Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997. |