Singleton Pattern

The Singleton pattern ensures that only one instance of an object can exist. In C, the API of such an object typically consists of methods that do not take an instance pointer (no self pointer as first parameter).

This pattern is often misused in C programs, as there are often hundreds of small variables declared as static, both within functions and outside of them. Some of these variables are even declared as global - which is even worse from code maintenance perspective. This is a major problem, so in this lesson, I will explain how the Singleton pattern should and should not be used. This will provide you with a clear understanding of the correct ways to use it and also give you guidance on how not to use it.

I will also demonstrate how to create a thread-safe Singleton in C, as well as a system-initialized Singleton that uses Zephyr pre-init code and is initialized before your firmware application starts.

Definition

The singleton pattern is a widely-used design technique that ensures that a specific class has only one instance at any given time. It also provides a global access point to this instance, making it easily accessible throughout your application.

It is important to note that there are situations where alternative design patterns are more appropriate.

The singleton pattern is defined by a few key characteristics:

- **Control over instantiation and use**: singleton pattern prevents multiple instances from existing in the system at the same time. Sometimes this is also extended to enforce single user at a time as well.
- **Private construction**: in C this basically means that construction is automatic and is done privately inside the module source code - and only once.
- **Stateless interface**: provides a method for getting the instance or implements only stateless methods with a call order agnostic api (methods do not take an instance pointer).

Use cases

The singleton pattern is commonly used for global subsystems and key firmware services such as the networking stack, virtual file system, and system hardware hierarchy.

- **Logging**: when you use the `LOG_INF` or `LOG_DBG` macros. Instead of passing an object into these macros, they reference a logger that is created using the `LOG_MODULE_REGISTER` macro at the top of your C file. The same logger singleton instance can be shared across multiple files by using the `LOG_MODULE_DECLARE` macro. By doing this, you are effectively sharing and referencing the global logger instance that was created with the `LOG_MODULE_REGISTER` macro.
- **Configuration**: the Zephyr settings subsystem acts as a singleton, allowing other subsystems to register endpoints that will be called upon loading and saving settings.

Callbacks are a common feature of singletons, as the singleton object acts as a "manager" for many other objects that are registered with it.

- **Firmware Subsystems**: such as power management, device initialization, and networking stacks.
- **Thread scheduler**: this is another example of a singleton. There is only one thread scheduler across the application and no need for multiple scheduler. Threads are "registered" with this scheduler and the scheduler then manages the threads. We let the system instantiate the scheduler.

All of these use cases share a common feature, in that the singleton serves as a central hub for registration by many other parts of the code.

We can generally compare the singleton pattern to a "service". Just like a single web server is able to accept web requests and route them to different subdomains and api endpoints, the singleton accepts "requests" from all application components and routes them where they need to go (or handles them internally). It is a shared resource where we are sure that only one instance is needed.

When we develop software in C, we use the singleton as a "manager" of resources that are "registered" with it (or created by it). Sometimes, our singleton can also act as a factory and be responsible for creating and disposing of resources. Other times, it simply acts as a hub for other parts of the system.

To identify additional use cases for the singleton, you can ask yourself "what parts of my application must be guaranteed to have only one instance?". These are the areas that can be improved by using the singleton pattern.

Benefits of the singleton pattern

The primary purpose of using the singleton pattern is to establish a global point of access. This ensures that there is only one instance of the object in existence, and all actions intended to be performed on that object will be executed on this singular instance.

Other benefits of using the singleton pattern include:

- **Single instance requirement**: the pattern ensures that there is only one instance of the object being created.
- **Simplifies code**: no requirement to pass the context along code execution path.
- **Improving performance**: Additional instances do not need to be created, initialized and destroyed. This comes at the expense of having to lock the resource while it is in use and serializing access to it.
- **Managing shared resources**: Your application components may require a single global point of access with which other subsystems can register their callbacks. Acts like a coordinator.

Reasons to avoid using singleton

The singleton pattern is often used automatically in C programming because many programmers do not apply object-oriented design principles to their C code - so all of their variables effectively become singletons from the start. In most programming situations, the singleton pattern is not the appropriate solution.

Examples of situations where the singleton pattern is not suitable include:

- **UART driver**: you can have multiple UARTs and using a singleton for this reduces flexibility. Instead, use multiple instances with a factory and abstract interface patterns to enable generic access to these instances.
- **Network protocol driver**: you must expect that there will be multiple devices using the same protocol. Therefore, the protocol implementation should be able to be instantiated multiple times and the states of these instances should be separate.
- **Abstract interfaces**: when you need to switch between multiple different implementations behind the same interface. A singleton actively prevents you from being able to implement such flexibility.
- **Concurrent operation**: when you have a shared singleton, it will always be a bottleneck if any operation within it takes time. This may not be a problem when part of your requirement is processing data in sequence, but it can be a bottleneck when many subsystems try to access the singleton from multiple threads.

You should not use the singleton pattern in any situation where there is a good chance that your object will be used by multiple threads and in multiple places with different states. This generally means: most cases.

Instead, reserve singleton usage for subsystems, global services, and aggregators of objects (when you have a single global facade and multiple registered implementations with callbacks).

Implementation

The most prominent feature of the singleton pattern is that it implements a single instance access to an object.

What do we mean by this?

A single point of access means that to access and use the object we do not need to have a reference to its instance. The instance data is instead contained in a static local instance inside the implementation C file and is not passed along the code path as we do with all other objects in a typical application.

Conventional object implementation has methods that take "self" as their first parameter:

instantiated objects

```
void object_method(struct object *self){
    ...
}
```

In contrast, when using the singleton pattern all member functions of a singleton class operate on a single instance and consequently we can simplify the interface by removing the "self" pointer and by not having to pass the instance reference around in other code that uses the singleton (you may however still need to pass an instance if you have a requirement to also have a single user at a time)

singleton objects

```
static struct object _singleton;

void object_method(void){
    ...
}
```

Singleton initialization

The first problem that arises is: who will be responsible for the initialization of the singleton instance?

In C\\, a common practice is to initialize the singleton when it is first accessed:

```
class SomeClass {
    private static SomeClass *_singleton = 0;
    public static SomeClass *getSingleton() {
        if(!_singleton){
            _singleton = ...
        }
        return _singleton;
    }
}
```

This approach initializes the singleton on demand and we have no control of when it is first initialized. In addition to that, this implementation suffers from lack of thread safety. We simply can not guarantee that two threads will not create the singleton twice because both of them have taken turns to check that _singleton was null and proceeded to create it.

The approach that I prefer to use in firmware development is by using startup code to call an init function which in turn initializes the singleton:

```
static struct object _singleton;
static void _init_singleton(void){
    ... initialize instance
}
SYS_INIT(_init_singleton, APPLICATION, priority)
```

This approach utilizes initialization tables, which are special sections in the executable file. These tables are parsed and executed by the system initialization code as the application is starting.

This way, I can control when my singleton is initialized and I do not have to worry about calling the initialization function from my application code, which could be prone to errors (and omitted initialization if application is changed).

The initialization table works like a singleton itself and the **`SYS_INIT`** macro, by placing an entry into that table, guarantees that the initialization function will always be called by the startup code and always before the main application.

This implementation approach is very useful for system-wide services because it allows us to start the service when the application starts and to do so in a robust way without introducing uncertainty about when it is actually started.

We don’t have this flexibility with static variable C++ constructors. The **`SYS_INIT`** macro is a feature we get from the Zephyr kernel.

Safe "on demand" creation

If you want to implement a singleton for an existing object using the initialization on demand approach then here is a way to do it.

We simply add a global function to the interface of the object:

```
struct object *object_get_singleton(void);
```

And we implement it in a thread safe way. Also for sake of reusability, I have implemented this concept using a macro that allows to easily duplicate it across multiple objects.

```
#define DEFINE_SINGLETON_TYPE(type) \
    static struct type *_##type##_self; \
    static struct k_spinlock _##type##_lock; \
    struct type *type##_get_singleton(void) \
    { \
        static struct type _singleton = { 0 }; \
        k_spinlock_key_t key = k_spin_lock(&lock); \
        if (!_##type##_self) { \
            _##type##_self = &_singleton; \
            ##type##_init(_##type##_self); \
        } \
        k_spin_unlock(&_##type##_lock, key); \
        return _##type##_self; \
    } \
```

The spinlock in this case ensures that threading (and all interrupts) are disabled when the singleton is being initialized. This avoids the possibility of it being initialized twice.

All we have to do then to use this macro to instantiate the implementation:

```
DEFINE_SINGLETON_TYPE(my_object)
```

And we can then use it:

```
void main(void){
    struct my_object *obj = my_object_get_singleton();
    my_object_do_something(obj);
}
```

Note that initialization of the singleton should be very basic at this point and simply do the minimum amount of work required to place it into operational state.

If you have access to the **`SYS_INIT`** approach then it is better if you use that approach instead. The **SYS_INIT** approach uses instanceless methods and completely hides the singleton instance from the user - removing the ambiguity that arises when looking at a variable and trying to deduce whether this variable is a singleton or not.

Interface considerations

Avoid this:

```
my_singleton_object_lock(); (1)
my_singleton_object_do_something();
my_singleton_object_do_something_else();
my_singleton_object_unlock();
```

[1]: this is stateful api usage pattern

In your application, any global interface, particularly singletons, should be stateless. When you create a function without a "self" pointer, consider it an indication that the function is stateless.

You should consider the lack of "self" pointer to be a cue to the fact that whatever function is being called must be stateless. On the other hand, a self pointer is an indication that a stateful operation may occur, as the self pointer represents the "state" of the object.

You should use this concept in your firmware even if other third party code may not follow this convention.

Additionally, the presence of a self pointer indicates that whoever is calling the function is also the owner of the "self" pointer at the time of the call.

Without a self pointer, it is impossible to designate the owner, and any part of the program can access and use the API without restriction. So you should follow the convention to require a "self" pointer to the state of the object as much as possible (it makes it clear who owns the object and consequently also leads to cleaner architecture).

Sometimes it is necessary that only one user owns the singleton at any given time. This is especially important if the API of your object is stateful (ie call order is important). We can implement this using a macro as well:

```
#define DEFINE_SINGLETON_ACQUIRE_RELEASE(type) \
    static struct type *_##type##_in_use; \
    struct type *type##_acquire_singleton(void) \
    { \
        k_spinlock_key_t key = k_spin_lock(&_##type##lock); \
        if (!_##type##_in_use) { \
            _##type##_in_use = &_##type##_singleton; \
            k_spin_unlock(&_##type##_lock, key); \
            return _##type##_in_use; \
        } \
        k_spin_unlock(&_##type##_lock, key); \
        return NULL; \
    } \
    void type##_release_singleton(struct type **self) \
    { \
        k_spinlock_key_t key = k_spin_lock(&_##type##_lock); \
        if ((self != NULL) && (*self == _##type##_in_use)) { \
            _##type##_in_use = NULL; \
            *self = NULL; \
        } \
        k_spin_unlock(&_##type##_lock, key); \
    }
```

The above macro expects **`DEFINE_SINGLETON`** to already be called before this macro to instantiate the singleton and lock. You can modify it however you want for your own usage pattern.

The usage of this singleton then becomes:

```
void main(void){
    struct object *self = object_acquire_singleton();
    if(self){
        object_do_something(self);
        object_do_something_else(self);
        object_release_singleton(&self);
    }
}
```

This API ensures that if the singleton has already been acquired, it cannot be acquired again. As a result, only one part of the application can use it at any given time.

We have implemented the release function to take a pointer to a pointer of self, as we want to ensure that the released pointer is set to NULL after we have released the singleton.

Unfortunately, the C language does not have a strict memory ownership convention that is enforced by the compiler. This means that we have to manage ownership through adherence to customized conventions by ourselves.

Best practices

- **Automatic initialization**: Initialize the instance at system startup. use initialisation priority to determine exactly when.
- **Stateless public API**: use my_object_operation(params) instead of my_object_operation(self, params) for singletons to avoid giving a pointer to this object to user.
- **Avoid singletons as much as possible**: avoid using singletons as much as possible and use conventional object oriented design in most of your code. Only use singleton pattern when it is an absolute requirement that there is only one instance of the object in your application at any time.
- **Use lock**: Use a lock when initializing the instance to avoid race conditions when multiple threads try to create or access the singleton object at the same time (not always necessary)
- **Hide the instance**: Use a static variable to hold the instance to ensure that the instance is only visible to the object implementation, and is not directly accessible through any

other means than the singleton interface itself.

Common pitfalls

- **Overuse**: Unthinkingly using the singleton pattern for everything will severely damage the flexibility of your architecture.
- **Stateful interface**: Since singletons are shared across many services that are part of your application, their API should not rely on call ordering.
- **Cutting corners**: Thinking that "now we only need one instance, maybe later we can shift to using more instances". The singleton pattern explicitly states that the only time you should use it is when there is no possibility at all that there will ever be more than one instance of the thing you are creating.

If you overuse the singleton pattern, you may encounter significant issues with the flexibility of your design. This is particularly true if you are implementing your API without applying the object pattern and passing the "self" pointer as first parameter to each function.

Refactoring existing singleton code to use object pattern is often one of the fastest ways to quickly improve your architecture.

To avoid this problem of flexibility and quality, you should only use the singleton pattern for services that you are certain will only exist as a single instance of the object you are implementing. If there is any possibility at all that you may want to add another instance in the future, you can either implement an instanced singleton (where member methods take a self pointer - thus making it easy to later refactor existing code if singleton is removed) or avoid the singleton pattern completely and simply implement the object pattern.

Another pitfall to avoid is the situation where you API expects a certain call order sequence:

do not do this:

```
my_object_lock();
my_object_do_something();
my_object_unlock();
```

Because your singleton is shared across the whole application (if it isn’t then you shouldn’t be using the singleton pattern for whatever you are implementing), you have no way of ensuring that your stateful interface is not violated by other components - which can easily lead to race conditions and hard to track bugs.

All of your singleton member methods should instead be stateless:

lock inside the function always:

```
my_object_do_something(); // locks and unlocks as needed
```

All the functions in the singleton interface must be callable in any order, from any thread and still always produce expected results.

Alternatives to singleton pattern

The primary case where you should consider alternatives to the singleton pattern is where multiple instances of the object must exist at the same time. The singleton pattern is primarily intended to prevent the existence of multiple instances.

Some of the alternative design patterns that you can consider instead are:

- **Object Pattern**: where the instance is created by user and then passed to the singleton. This is also a variation of dependency injection pattern.
- **Factory Pattern**: where you hide the details of creation of the instance from the user thus simplifying the user code.
- **Prototype Pattern**: where you copy an existing object using a virtual constructor (a variation of the virtual API pattern)
- **Endpoint API Pattern**: where you make a call to a global function and provide a path such as "path/to/endpoint" and the function then resolves the path to a previously registered callback. This is a form of request routing. This can also be referred to as "Service Locator Pattern".

Final remarks

In this module, we have examined the singleton design pattern, including when it is appropriate to use it and when it is not.

The quality of your code depends on your ability to adhere to patterns while coding. If you continually take shortcuts and excuse bad practices with the mentality of "I’ll fix it later," you will continue to accumulate technical debt and make the same mistakes repeatedly.

Now that you have a better understanding of singletons, you should be able to clearly identify when and how to use them correctly.

There is no longer any excuse for misusing or abusing this pattern in your code.

Quiz

- What is the main purpose of the singleton pattern?
- When should you avoid using the singleton pattern?
- How is the singleton pattern implemented in C?
- Give a few examples of common use cases for the singleton pattern in firmware development.
- What are best practices when implementing the singleton pattern?
- What are common pitfalls?
- Can you see some cases in your existing code where you can improve structure using the singleton pattern?
- Can you see some places where singleton pattern is being misused at the expense of code flexibility?