Callback Pattern

The callback pattern is a central design pattern that is used when we want to notify multiple other objects of the state changes to the object being monitored.

This pattern is also often referred to as "Observer Pattern", however, in embedded environments we typically also need to think about which context our callback will run in and so we have to cover the callback itself in greater detail.

I have therefore chosen to call it the "Callback" pattern because it contains more features than the observer pattern and nearly all of these features are centered around the callback itself.

In this module we are going to cover the implementation of the callback pattern specifically geared towards embedded software development and we will make sure that it is type safe and clean in its design so that your project can maintain a scalable and decoupled software architecture.

Definition

The Callback design pattern is a behavioral pattern in which an object, called the subject, maintains a list of its dependents, called 'observers', and notifies them automatically through callbacks of any changes to its state, allowing multiple objects to observe the changes to the subject and be notified accordingly.

- **Object being observed**: this can be an opaque object (with hidden implementation) exposing only an interface to register callbacks for various events. Observers register callbacks with this object.
- **Callback receiver**: this is the code that executes the callback. One important consideration is that this function must have a way to retrieve callback owner state because from the perspective of the object being observed, we don’t know about the object that is observing it - we only have the callback.
- **Full separation of concerns**: The object being observed is typically fully separated from the objects observing it. This is an important feature of this pattern because it allows clean separation of concerns.

Use Cases

In many projects callbacks are implemented without due diligence. The result is architecture that is hard to maintain. For example, if you have multiple instances of a data structure and you want to register callbacks for each instance individually then in many cases you need to somehow pass the pointer to the private data of that instance to the callback.

Traditional C programming doesn’t provide a straightforward way to do this. This is why we have the callback pattern.

The callback pattern is also:

- **A mechanism for notification**: the core concept behind this pattern is ability to register and receive notifications.
- **Used to 'Observe' State Of Object**: this can be done in response to event notifications or

directly through parameters of the callback.
- **An idiom for 'One-To-Many' notifications**: it is not uncommon to have unlimited number of observers spread around your code. The pattern helps keep them independent of each other.
- **Used for patterns like the Work Queue**: when used together with a work queue, we can build observer of a work queue item which will execute a callback when a timeout expires. This is very powerful way to avoid unnecessary threads in embedded systems.

One of the most interesting ways in which the callback pattern is used in embedded software development is specifically when we use it in conjunction with work queue items.

This allows us to submit work to the system work queue and ensure that all work queue items run in sequence without interrupting each other. Having this guarantee allows us to keep our code clean and implement asynchronous programming without any complex logic besides the work queue items themselves and a few variables to hold our software state (we will look at this in more detail below).

Benefits

- **Decoupling**: A callback can be represented by a structure making it the only data structure that needs to be shared between the observing code and the object being observed. This is important because we don't want our object being observed to have to depend on every single other object that needs to receive notifications from it.
- **Enables One-To-Many notifications**: since the only shared data structure is our callback structure, we can easily add features to this callback - such as the ability to arrange our callbacks into a list which in turn allows our object to send notifications to multiple receivers in sequence.
- **Code reuse**: Callback listeners (observers) can be reused in multiple subjects, reducing the duplication of code.
- **Scalability**: The callback pattern makes it easy to add new observers without affecting the subject or other observers (without the callback pattern, our subject would have to be directly dependent on API of all observers - which is clearly very bad).
- **Flexibility**: The callback pattern provides a way to change the notification mechanism dynamically, making the system more flexible.
- **Ease of maintenance**: The observer pattern makes it easy to maintain the system, as it reduces the dependencies between objects and makes it easier to modify or extend the system. One of the biggest sources of maintenance drag in many C projects is 'dependencies out of control'!

We implement full decoupling by implementing a callback as a structure. This will allow us to retrieve the memory location of our private state later - making our code fully type safe even though we are using C to express this pattern!

Drawbacks

- **Performance overhead**: Delivering all of the notifications of a change to all listeners can result in a significant performance overhead, especially if there are many listeners.

- **Lockup**: if one observer/listener happens to block execution for whatever reason, all other listeners will not have the chance to run. Preferably, any action done as part of the response to a notification must make sure that it does not sleep (block) the execution in any significant way.
- **Complexity**: The callback pattern can increase the complexity of the system, as it requires implementing and maintaining the observer list and the notification mechanism. Luckily we can reuse ready made data structures for this which are available to us in Zephyr RTOS.
- **Tight coupling risk**: If not implemented carefully, the callback pattern can result in tight coupling between the subject and observers, making it difficult to change or extend the system. This is particularly a risk when the callback is implemented as a direct call instead of a function pointer.
- **Order of notifications**: There is typically no predictable sequence in which observers are notified. This can create problems if the listeners are designed in a way that does not allow them to perform actions independently.

To avoid these problems it is very important to implement the callback as a structure object and to make sure that all other objects in the system follow the 'Object' pattern as much as possible having reasonably independent operations that do not depend as much on state outside of the object and rely mostly on the state of the object itself (basically, for best result, the interface of each object in the system must be reentrant and commutative).

Implementation

The most basic way of using the callback pattern requires us to define our callbacks as a struct:

```
struct button_callback {
    void (*cb)(struct button_callback *cb);
};
```

A struct containing the callbacks can contain one or more callbacks. But the key notion here is that our callbacks are inside a dedicated callback struct. We can call this struct 'ops' - short for 'operations'.

The callback function itself takes pointer to the callback structure. This is is an important feature of using a struct in this case because when we use a struct we can pass the struct pointer to the function itself. Without a struct we can not pass the pointer to the function we are defining into the function itself.

This design gives us two advantages:

- **Advantage #1**: we can easily instantiate the callback itself - which works as a much cleaner alternative to typedef.
- **Advantage #2**: we can use pointers to the callback structure to get address of the higher level context of module a instance which allows us to decouple A and B. This would be impossible if our callback was just a function pointer.

Let’s look at how this is done:

Let’s start off by implementing a button object which will then allow an observer to register callbacks. The callbacks will then be called when button needs to notify the observer of something happening (such as a button press).

We will start off with a single callback and then we will also make our button support a list of callbacks.

File: button.h

```
#include <zephyr/sys/slist.h>
#include <stddef.h>

struct button {
    /** Alternative 1: single callback */
    struct button_callback *cb;
    /** Alternative 2: multiple callbacks */
    sys_slist_t callbacks;
};

// this is a public callback definition
struct button_callback {
    sys_snode_t node;
    void (*cb)(struct button_callback *cb);
};

void button_init(struct button *self);
void button_deinit(struct button *self);
void button_add_callback(struct button *self, struct button_callback *cb);
void button_remove_callback(struct button *self, struct button_callback *cb);
void button_do_something(struct button *self);
```

The button implementation is very simple here. We just provide a way of storing the callback and then calling it upon some operation.

File: button.c

```
#include "button.h"

#include <zephyr/sys/slist.h>
#include <string.h>

void button_init(struct button *self){
    memset(self, 0, sizeof(*self));
}

void button_deinit(struct button *self){
    self->cb = NULL;
}

void button_add_callback(struct button *self, struct button_callback *cb){
    self->cb = cb;
}

void button_remove_callback(struct button *self, struct button_callback *cb){
    self->cb = NULL;
}

void button_do_something(struct button *self){
    if(self->cb)
        self->cb->cb(self->cb); // call the callback
}
```

The observer is there to respond to the events coming from the button. The key point here is that we instantiate the callback struct as part of our object - which then allows us to get the pointer to the instance of the observer from inside the callback handler itself:

Observer source code

```
          2
struct observer {
    /** Button instance */
    struct button *btn;
    /** Button callback */  3
    struct button_callback cb; // our callback
};
                                                           1
static void observer_on_button_cb(struct button_callback *cb){
    // this is how we can now get pointer to instance of B
    // notice that the callback is generic1     2              3
    struct observer *self = CONTAINER_OF(cb, struct observer, cb);
    printk("Observer callback for button %p\n", self->btn);
}

void observer_init(struct observer *self, struct button *a){
    self->btn = a;
    self->cb.cb = observer_on_button_cb;
    button_add_callback(a, &self->cb);
}

void observer_deinit(struct observer *self){
    button_remove_callback(self->btn, &self->cb);
}
```

Now we can use this in our application like this:

File:application.c

```
struct observer ctrl;
struct button btn;

button_init(&btn);
observer_init(&ctrl, &btn);

// this will call observer callback function
button_do_something(&btn);

observer_deinit(&ctrl);
button_deinit(&btn);
```

Notice how elegant this is. We are completely decoupling our button from the observer. The observer of course needs to know about the type of the button, but the button still no longer needs to know about all possible observers. We have thus successfully decoupled our button from one or more observer types that can respond to its events.

If we would like to extend this to support multiple listeners, all we have to do is add a list node to the callback structure:

```
struct button {
    /** List of callbacks */
    sys_slist_t callbacks;
};
struct button_callback {
    /** Callback list node */
    sys_slist_t node;
    void (*cb)(struct button_callback *cb);
};
...
void button_add_callback(struct button *self, struct button_callback *cb){
    sys_slist_append(&self->callbacks, &cb->node);
}

void button_delete_callback(struct button *self, struct button_callback *cb){
    sys_slist_find_and_remove(&self->callbacks, &cb->node);
}
void button_do_something(struct button *self){
    sys_snode_t *node;
    /* Call all the callbacks */
    SYS_SLIST_FOR_EACH_NODE(&self->callbacks, node){
        // once again we use container of here since we know node points to cb->node
        struct button_callback *cb = CONTAINER_OF(node, struct button_callback,
node);
        cb->cb(cb); // call the callback
    }
}
```

One important fact to remember is that your button callback structure must never go out of scope. You must make sure that it never does - otherwise your callback pointer will be garbage. This means that if you allocate the object that holds the callback on stack, you must make sure to implement a way to remove the callback before the object goes out of scope and unlink it.

Best practices

The primary purpose of the callback/observer design pattern is to implement simple publish-subscribe mechanism that can be used extensively throughout the application and without being too heavy on the developer.

Here are a few best practices that I have found work really well when working with this design pattern:

- **Always use struct for the callback**: this gives context and most importantly ensures that we can use it easily together with the object pattern and make our architecture truly object oriented.

- **Always pass context to callback**: do not use "custom" context such as "user data" or some other thing you have invented just for passing data to the callback. Instead place the data either in the callback struct itself or in the enclosing object.
- **Avoid time consuming operations in callback**: It is better to defer operations to a dedicated work queue if they take longer time because a time consuming callback will stall all other callback handlers that are also waiting for the notification.
- **Use `CONTAINER_OF`**: do not try to pass context as a separate 'user data' variable or anything like that. The callback pattern makes these things unnecessary!

Common Pitfalls

As with almost every single task in C, the biggest issue is always "how do we manage our data?". Most of the time developers opt for using global variables to store the data and this of course leads to very convoluted and tangled up C programs.

Thus, here are possible pitfalls that you will run into when implementing this pattern in C:

- **Callback going out of scope**: failing to unregister callbacks before destroying a callback observer is one of the biggest pitfalls of this pattern.
- **You make your data global**: this may look like a nice convenience, but it completely violates the data flow through the code. If you for instance make the callback operate on global data, you have not not only messed up the data flow for one listener - but for all listeners by not passing a pointer to a structure that holds the callback.
- **Inventing 'userdata' passing**: whenever I see some variation of 'void *user_data' in a structure or callback, I know that the developer does not really understand how to properly implement the observer pattern. Remember: you do not need to store pointer to user data at all. You get access to user data using **`CONTAINER_OF`** macro.
- **Life cycle of objects**: if you do not pay close attention to when a stack allocated listener goes out of scope, you may run into very weird problems caused by garbage pointers (due to the fact that the object no longer exists but is still referenced from the list of callbacks!). So pay close attention to where you data resides. Of course, if you use the object pattern, you will have a pretty good order in your data.

Alternatives

- **Event Bus Pattern**: this pattern is slightly different from observer in that we have a global event **`queue`** (observer/callback pattern has NO event queue) that stores events in transit. This pattern is used to implement conventional "publish/subscribe" mechanisms at small scale in C programs. Both observer and the observed become decoupled.
- **MVC (Model-View-Controller) Pattern**: This pattern is a derivation of the callback pattern in that the controller acts as an adapter between the model and the view. The controller uses the model to represent state of the program and registers callbacks with the visual representation to respond to events that are necessary for smooth visual rendering.
- **Command Request Pattern**: This pattern uses a command request as central way to pass an event. It is different from the event pattern in that the request itself can be

queued and the object that queues the event does not need to know about the type of object that receives the event.

Conclusion

This module has familiarized you with the details and uses of the callback/observer design pattern. This should be enough to ensure that from now on you will always be implementing your callbacks the right way.

You should now understand how to use the callback pattern to decouple your "subject" being observed from the code that responds to events that it emits. You now have a new tool in your programming toolbox for cases where your "subject" can not ever know about all possible "observers" and so you understand when it is the right action to take to use a callback mechanism instead of a direct call.

Quiz

- What is the main problem that is being solved by using the callback mechanism?
- How does the observer/callback pattern help maintain object oriented code organization in C?
- What would happen if you delivered the event using a direct call instead? What implications would it have for structure of your code?
- Why are we always enclosing our callback in a struct? How would it negatively impact our design if we did not do this?
- Why do we use `CONTAINER_OF` to get our context and not just make the context global or pass it directly from the subject to the callback?