Design

Code Modularity

Code should be broken apart into smaller pieces which has advantages:

- Small code modules are easier to test
- Provides better focus for a problem and avoid "God Object" software anti-pattern
- Usually there are very few bugs when code modules integrate together

What this will avoid is building "god objects" or also know as The Blob Antipattern (please read this article).

Problem

Imagine the following problem scenario given to you during an interview.

```
/**
 * An interrupt is occurring periodically and providing us burst of data to work with.
 * Invoke the process_data_chunk() function for batches of 8 bytes at a time
 * Data input (data_size_in_bytes) may be variable amount of data, and not necessarily 8 bytes
 */
void periodic_interrupt_routine(const char *data, size_t data_size_in_bytes) {
}
void process_data_chunk(const char data_chunk[8]);
```

Naive Solution

A naive solution assumes a common mistake to directly jump into the problem without thinking about **code modularity** and **separation of responsibility**.

The following code is:

- Hard to read
- Is trying to solve too many problems at once (functions should be less than 10 lines of code)

Another problem is that although the code below seems to be done well, and has plenty of comments, it is actually not good code. At first you may believe that commented code is great, and is of high quality, but it is not true. Code should express its thoughts without comments, and the fact that **you need a comment means there is something wrong**. Compare this snippet of code to the Elegant Solution below and observe how the elegant solution is self-expressive and does not need comments. The comments in this Naive Solution are actually poor and require a lot of future maintenance.

```
static char buffer[8+8];             // Buffer space
static size_t buffer_offset;         // Write offset into the buffer
static const size_t chunk_size = 8;  // Size of chunks we work with

void periodic_interrupt_routine(const char *data, size_t data_size_in_bytes) {
  size_t remaining = data_size_in_bytes;
  size_t data_offset = 0;

  while (remaining > 0) {
    // Add small chunks at a time to avoid having a really large buffer size
    const size_t bytes_to_copy = MIN_OF(chunk_size, remaining);

    // Copy the data into our buffer
    memcpy(buffer, (buffer+buffer_offset), (data + data_offset), bytes_to_copy);
    buffer_offset += bytes_to_copy;
    // If our buffer contains minimum chunk size, then process it
    if (buffer_offset >= chunk_size) {
      char chunk[8] = { };

      // Copy from buffer to the chunk we will pass to process_data_chunk()
      memcpy(chunk, buffer, chunk_size);
      // Move the buffer contents back to the beginning
      memmove(buffer, (buffer + chunk_size), (buffer_offset - chunk_size));
      buffer_offset -= chunk_size;
      process_data_chunk(chunk);
    }

    data_offset += bytes_to_copy;
    remaining -= buffer_size;
  }
}
void process_data_chunk(const char data_chunk[8]);
```

Elegant Solution

The solution here takes a top down approach to software, and assumes a module that will buffer our data such that we are not dealing with the data manipulation inside of the periodic data routine. The advantages this solution provides is that:

- Data buffering is part of another code module's responsibility
- Code is easier to read, and maintainable
- We potentially re-use a buffer module (maybe it already exists in your code base)

Notice how easy the code is to read and it is **self expressive** and doesn't need further comments. Yes, there is of course another code module to design (`buffer_module.h`), however, its code can be independently tested, and doesn't introduce distraction to solving our periodic data buffering issue.

```
#include "buffer_module.h"
static buffer_module data_buffer;
static const size_t chunk_size = 8;

void periodic_interrupt_routine(const char *data, size_t data_size_in_bytes) {
  size_t remaining = data_size_in_bytes;
  const char * read_pointer = data;
```

```
    while (remaining > 0) {
      // Add small chunks at a time to avoid having a really large buffer size
      const size_t size_to_process = MIN_OF(chunk_size, remaining);
      buffer_copy_in(&data_buffer, read_pointer, size_to_process);
      read_pointer += size_to_process;
      // If our buffer contains minimum chunk size, then process it
      if (buffer_get_size(&data_buffer) >= chunk_size) {
        char chunk[8] = { };
        buffer_copy_out(&data_buffer, chunk, chunk_size);
        process_data_chunk(chunk);
      }

      remaining -= size_to_process;
    }
  }
  void process_data_chunk(const char data_chunk[8]);
```

Conclusion

You want to be sensitive in terms of what the code module is responsible to do. You can start by observing what is the file name and what is the function name.

For example, if the file name is `handle_periodic_data.c`, then it should do exactly that. It should delegate the buffering responsibility to another code module, and focus on what you do with the data, and not the internals of how the data is buffered.

If a function is called `buffer_copy_in()`, then it should only copy the data, and not deal with things like blinking an LED or talking to another code module.

Further experience will definitely help you see different code modules that should be used to solve a problem. You can also start by:

- Ensure header files only include those modules they really use
- Avoid adding un-related responsibilities to code modules
  - Software should be loosely coupled and have high cohesion