Useful Knowledge

Articles about C language How to build good C code modules etc.

Basics of C

Struct Address

Objective

- Learn basics of data structures
- Learn how memory may be padded within data structures

Review Basics

Here is the basic use of data structures in C:

```c
// Declare data structure in C using typedef
typedef struct {
  int i;
  char c;
  float f;
} my_struct_t;
// Pass data structure as a copy
void struct_as_param(my_struct_t s) {
  s.i = 0;
  s.c = 'c';
}
// Pass data structure as a pointer
void struct_as_pointer(my_struct_t *p) {
  p->i = 0;
  p->c = 'c';
}
// Zero out the struct
void struct_as_pointer(my_struct_t *p) {
  memset(p, 0, sizeof(*p));}
```

Padding

1. Use the struct below, and try this sample code
   - Note that there may be a compiler error in the snippet below that you are expected to resolve on your own
   - Struct should ideally be placed before the main() and the `printf()` should be placed inside of the `main()`
   - You should use your SJ embedded board because the behavior may be different on a different compiler or the board

2. Now un-comment the `packed` attribute such that the compiler packs the fields together, and print them again.

```
typedef struct {
  float f1; // 4 bytes
  char c1;  // 1 byte
  float f2;
  char c2;
} /*__attribute__((packed))*/ my_s;
// TODO: Instantiate a struct of type my_s with the name of "s"
printf("Size : %d bytes\n"
       "floats 0x%p 0x%p\n"
       "chars  0x%p 0x%p\n",
       sizeof(s), &s.f1, &s.f2, &s.c1, &s.c2);
```

Note:

- Important: In your submission (could be comments in your submitted code), provide your summary of the two print-outs. Explain why they are different, and try to draw conclusions based on the behavior.

Design a code module

This article demonstrates how to design a new code module.

Header File

A header file:

- Shall have `#pragma` once attribute (google it for the reason)
- Shall NEVER have variables defined

```
//- Note: Remove all lines from your code that start with //-
//- Put this line as the very first line in your header module
#pragma once
//- #include all header files that THIS header needs
//- Do not include headers here that are not needed
//- For example, we do not need gpio.h file here, but maybe you can move this to switch_led.c
#include "gpio.h"

//- DO NOT put any variables here, like so:
static int do_not_do_this;
int definitely_do_not_do_this;
//- All functions without paramters should be marked as (void)
void switch_led_logic__initialize(void);
void switch_led_logic__run_once(void);
```

`#pragma` once is a replacement of

```
#ifndef YOUR_FILE_NAME__
#define YOUR_FILE_NAME__
```

```
void your_api(void);
#endif
```

Intent of `#pragma` once and `#ifndef`

- When other code modules `#include` your header file, you only want functions to be declared once
- The name of `#ifndef` can be anything unique, but must not conflict with other files
- `#include` literally copies and pastes the contents of the file in the file wherever you have the `#include`

Source File

A source file:

- Shall have all variables defined as static; this will keep their visibility private to their file

```
//- Note: Remove all lines from your code that start with //-
//- Include the header file for which this code modules belongs to
#include "switch_led_logic.h"
//- Declare all variables as STATIC
static gpio_s my_led;
//- Define your public functions (part of this module's header file)
void switch_led_logic__initialize(void) {
  my_led = gpio__construct_as_output(GPIO__PORT_2, 0);
}
void switch_led_logic__run_once(void) {
  gpio__set(my_led);
}
```

Unit Test file

A unit-test file:

- Shall `#include` the headers that you want (those that should not be "mocked")
- Shall `#include` Mock headers to generate stubs (rather than the full implementation)

Useful stuff

Clang auto-formatter will format the source code for you. It will also sort the #includes, so it is recommended to put an empty line such that it sorts the #includes more elegantly. For example, you can separate the FreeRTOS includes, system includes, and other includes.

```
//- Note: Remove all lines from your code that start with //-
//- Include system includes first
#include <stdio.h>
//- FreeRTOS requires this header file inclusion before any of its soure code
//- This only applies to code included from FreeRTOS
#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"
//- Clang will sort these
#include "abc.h"#include "def.h"
```

Try the following

- Have two code modules, such as `main.c` and `periodic_callbacks.c` include a header file that does not have `#pragma once` and observe what happens when you compile

Function Pointer

Pointers

Pointers are the data types that can be used to store the address of some data stored in a computer's memory. Pointers are mostly used as a data type that would store the address of other variables.

Pointers can point to data/functions where data could be stored as a constant or a variable. We can also use pointers to dereference and get the value at whatever address the pointer is pointing at.

```
// <varaible_type> *<name>
// example:
int data;int *pointer_to_integer = &data;
```

Function Pointers

Function pointers are used to store the address of functions. We need function pointers to make "callbacks", but let us understand the basic syntax first.

Function Pointer Syntax

1. If the function return type is void

```
void (*func_pointer)(void);
```

Let us re-read the syntax, `*func_pointer` is the pointer to a function. `void` is the return type of that function, and finally `void` is the argument type of that function. The parenthesis around the function pointer is a must otherwise it will change the meaning of the function pointer declarations.

2. If a function returns an `int` and has a `char*` as an input parameter, then the code looks like this:

```
int (*func_pointer)(char *)
```

In this example:

1. `*func_pointer` is the function pointer
2. `int` is the return type of that function
3. `char*` is the type of argument.

Examples

Code Example 1: Function pointers with an int as an argument

```c
#include <stdio.h>
void function(int arg) {
  printf("Function being called and arg is: %d\n", arg);
}
int main(void) {
  void (*func_pointer)(int);

  // assign function to the function pointer
  func_pointer = &function;

  // call the function pointer
  (*func_pointer)(6);

  // Or call it like this:
  func_pointer(6);}
```

Code Example 2: Function pointer returns and taking argument as void data type.

```c
// Let us "typedef" the function pointer: void void_function(void);
typedef void (*void_function_t)(void);
void foo(void) {
  puts("Hello");
}
int main(void) {
```

```
  // assign function to the function pointer

  void_function_t func_pointer = foo;


  // call the function pointer

  func_pointer();}
```

Code Example 3: How to use an array of functions using function pointers.

```
/* Example 1 */

void foo(void) { puts("foo"); }

void bar(void) { puts("bar"); }

// Typedef a function with void argument, returning nothing (void)

typedef void (*void_function_t)(void);

int main(void) {

  // assign array of functions to the function pointer

  void_function_t func_pointers[] = {foo, bar};


  // call the function pointers

  func_pointers[0]();

  func_pointers[1]();

}
/* Example 2 */

/* For simplicity considering number_one > number_two */

int add(int number_one, int number_two) { return number_one+number_two; }

int sub(int number_one, int number_two) { return number_one-number_two;  }

int multiply(int number_one, int number_two) { return number_one*number_two; }

int divide(int number_one, int number_two) { if(number_two !=0) return (number_one/number_two); else re

int main(void) {

  int x = 10, y = 2;

  int choice,result;


  // assign array of functions to the function pointer

  int (*function_pointer[4])(int,int) = {add, sub, multiply, divide};


  printf("Enter 0: For Addition, 1 for subtraction, 2 for multiplication, and 3 for division: ");

  scanf("%d", &choice);
```

```c
    // call the required function pointer
    result = function_pointer[choice](x, y);


    printf("Result: %d\r\n", result);
    return 0;}
```

Miscellaneous

Switch / Case Statements

Normally switch / case statements are encouraged.

Here is an article that is sort of against it.

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
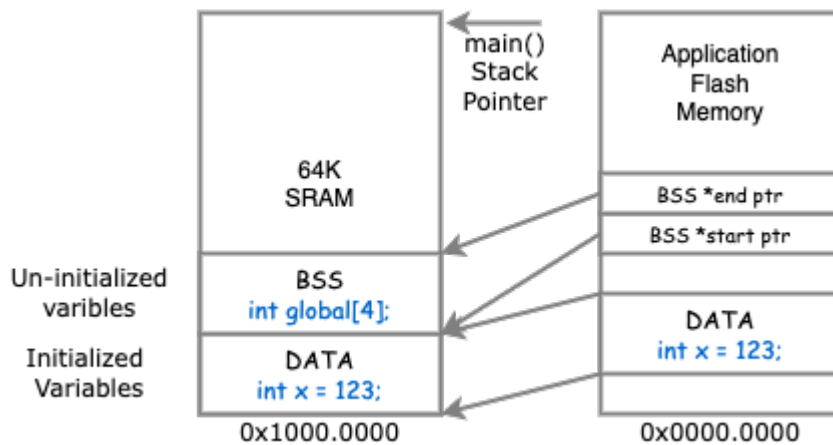
Embedded

Embedded

System Calls

TODO

stdio

memory

Startup

TODO: Preet to write content regarding startup, bss memory and data memory

Here are snippets of code the zero initialize the BSS and copy the DATA section from ROM to RAM.

```
static void startup__init_data_sram(void) {
  extern void *_bdata_lma;
  extern void *_bdata_vma;
  extern void *_data_end;
  uint8_t *src_flash = (uint8_t *)&_bdata_lma; // Flash
  uint8_t *dest_ram = (uint8_t *)&_bdata_vma;  // RAM
  while (dest_ram < (uint8_t *)&_data_end) {
    *dest_ram = *src_flash;
    dest_ram++;
    src_flash++;
  }
}
static void startup__init_bss_sram(void) {
  extern void *_bss_start;
  extern void *_bss_end;
  uint8_t *sram_ptr = (uint8_t *)&_bss_start;
  while (sram_ptr < (uint8_t *)&_bss_end) {
```

```
        *sram_ptr = 0U;

        sram_ptr++;
    }
}
```