Basics of C

- Struct Address
- Design a code module
- Function Pointer

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:

```
// 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

```
#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.

```
// 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 r
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);

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

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