Firmware Design

Homework 2

Ghady Youssef, 230707
ghady.youssef@net.usj.edu.lb

October 2024

Exercise 1

The `union` in C is a data structure that allows us to define multiple properties that are stored in a single location in memory. Unlike `structs` whom take the space of every field defined. Take for example a simple `struct` that contains two `unit16_t` fields. It will have a size of 32 bits or 4 bytes. If we have the same two fields defined in a `union`; we will have a variable that takes up only 16 bits or 2 bytes. The rule is that `unions` take up the space of the largest field in memory which makes sense; since we need to account for the largest element possible that can be stored in this specific memory location.

A practical use-case for this C construct is in the embedded programming domain. Typically, if we want to access individual bits from a register or a hardware-specific component; we can use a combination of `unions` and `structs` that provide us two "APIs" for our variable: we could either treat it as a bit-array or handle it as a standard integer.

Exercise 2

In all the code snippets, we assume the appropriate headers are included.

```c
#include <stdint.h>
#include <stdio.h>
```

Listing 1: `get_font_info` with the traditional approach

```c
void get_font_info( uint16_t font, uint8_t *type,
                    uint8_t *size, uint8_t *bold,
                    uint8_t *italic)
{
    *bold = (font >> 15) & 1;    // extract the first bit
    *italic = (font >> 14) & 1; // extract the second bit
    *size = (font >> 8) & 0x3F; // extract the 6 bits for the size
    *type = font & 0xFF;         // extract the lower byte
}

int main() {
    uint16_t font = 0xFFFF;

    uint8_t type = 0, size = 0, italic = 0, bold = 0;

    get_font_info(font, &type, &size, &bold, &italic);
```

```
17
18      printf("The font contains the folliwng properties:\n");
19      printf("Bold: %i\n", bold);
20      printf("Italic: %i\n", italic);
21      printf("Size: %i\n", size);
22      printf("Type: %i\n", type);
23
24      return 0;
25  }
```

Exercise 3

First, we will define a `union` that contains two fields: a `struct` that contains the font information. This information is stored according to what was given: 1 bit for bold field, 1 bit for italic field, 6 bits for the size and 8 bits for the type. The second field `font` is a `uint16_t`, this will act as the value of the variable.

The utility of the `union` is to allow a single location in memory to be accessed in different ways; either by accessing it as a standard integer value or by accessing individual bits by name using the `struct`'s fields. As you can see in the code below, for the `struct`'s fields we are specifying the size of each "property" using the `:` notation.

Listing 2: The font type

```
1  typedef union font
2  {
3      struct info
4      {
5          uint8_t bold : 1;
6          uint8_t italic : 1;
7          uint8_t size : 6;
8          uint8_t type : 8;
9      } info;
10
11     uint16_t font;
12 } font;
```

Next, we redefine the `get_font_info` to accept a font `union` and output the relevant information to standard output.

Listing 3: `get_font_info` using `union`s and `struct`s

```
1  void get_font_info(union font *f) {
2      printf("The font extracted contains the folliwng properties:\n");
3      printf("Bold: %i\n", f->info.bold);
4      printf("Italic: %i\n", f->info.italic);
5      printf("Size: %i\n", f->info.size);
6      printf("Type: %i\n", f->info.type);
7  }
```

As we can see, the function can simply access the properties directly. Instead of having to write bit manipulation code to extract each property according to the space it is occupying. This is possible since we are defining the space of each field in the `struct`'s definition. Also, as you can notice, we are utilizing the `struct` "API" of our type. In Listing 4 we will see that we use the `uint16_t` "API".

The main function becomes as follows.

Listing 4: The main function

```
int main() {
    union font f;
    f.font = 0xFFFF;

    get_font_info(&f);
}
```

Here, we can see that we are not accessing specific fields (even if we could have done that), we are instead setting the variable's value using a hexadecimal constant.

Exercise 4

To achieve the requested task; we modified the IOs example shown in class to handle both scenarios and allow them to co-exist.

Mainly, we have two cases: 1. PB1 is pressed and **held**; we have to handle both D1 and D2's flashing 2. PB2 is pressed-and-**released**; we only have to handle D2's flashing. To handle the former case, we listen for a press on PB1; once pressed, we incrementally light D1 for a period of 0.5s. This will allow us then light D2 properly. To elaborate further; we are turning D1 on or off depending on `b1_delay` which can have the values of 1s or 2s; so we use the lighting times of D1 as a delay for D2. If we did not take this approach we would have extra delay.

```
output_toggle(PIN_C0);
delay_ms(B0_DELAY);

output_toggle(PIN_C1);
delay_ms(b1_delay - B0_DELAY);
```

In this example, we toggle D1 then wait, we then toggle D2 and wait the duration of `b1_delay` minus the delay from D1. However, since this is a loop; waiting times are accumulating and we are waiting **synchronously** and these two scenarios do not **co-exist** properly.

To handle the latter, we just have to listen for a press on PB2 and wait for the release. Then, we can use `toggle_delay()` to switch from 1s to 2s (and vice-versa). Keep in mind when we wait for the release of PB2, we have to keep flashing on the current delay time.

All the code presented in this assignment are located in a GitHub repository [2]

Listing 5: Firmware code

```
#include "main.h"
#include "hardware.h"
#include "system.h"
#include <stdint.h>

bool toggle = false;       // Toggle delay from 1s to 2s (and vice-versa)
uint16_t B0_DELAY = 500;   // Delay for PIN_B0
uint16_t b1_delay = 1000; // Delay for PIN_B1

void toggle_delay()
{
    if (toggle)
    {
        b1_delay = 1000;
    }
    else
    {
        b1_delay = 2000;
    }
    toggle = !toggle;
}

void main()
{
    init_hw();      // Initialize HW (IO ports)
    delay_ms(100); // Power up delay, wait for 100ms

    while (1) // Infinite loop
    {
        if (!input(PIN_B0)) // Handle case when D1 and D2 are flashing
        {
            while (!input(PIN_B0))
            {
                /*
                    Distribute output in 500ms periods to
                    incrementally light D1 depending on D2's delay
                 */
                for (uint8_t i = 0; i < b1_delay / B0_DELAY; i++)
                {
                    output_toggle(PIN_C0);
                    delay_ms(500);
                }

                output_toggle(PIN_C1);
            }
        }
        else // D2 is only flashing
        {
            output_low(PIN_C0);
            if (input(PIN_B1))
            {
                while (input(PIN_B1))
                {
                    output_toggle(PIN_C1);
                    delay_ms(b1_delay);
                }

                toggle_delay();
```

```
59                }
60                else
61                {
62                    output_toggle(PIN_C1);
63                    delay_ms(b1_delay);
64                }
65            }
66        }
67    }
```

References

[1] Why do we need C Unions? StackOverflow [Online]

[2] The GitHub Repository that contains the code in this paper. GitHub [Online]