SYS2/CCCP: An Example Low-Level Program

For Java and Python Programmers

Dr David Griffin

Example Scenario

The scenario is making a user-space application that controls some hardware through a driver

This driver provides a header which contains various `#defines` and function prototypes

The driver also provides documentation on how to communicate with the hardware

The task is to make a user-space application that reads instructions from a file and executes those instructions on the hardware

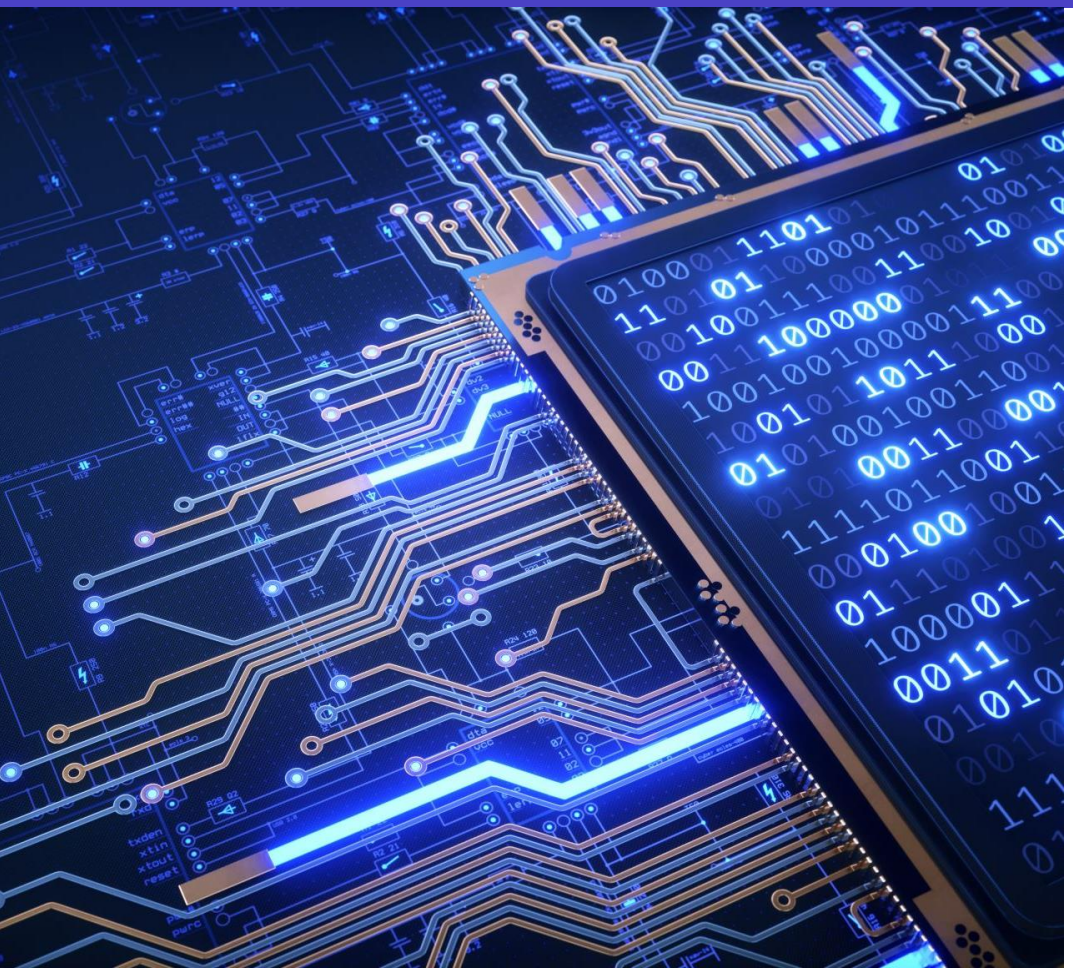
Device Specification

Input Byte

Output Byte

- Device has
  - An array of 3 LEDs
  - A buzzer
  - A 4-bit motor speed controller
  - A 4-bit revolution sensor (speed of motor)
  - A 1-bit motor status sensor (on/off)
  - 3 1-bit alarm sensors (should be recorded)
- Communication by a specially formatted 8-bit input/output bytes passed to the driver

Program Requirements



Read a sequence of commands from a text file given on command line

Execute those commands on the hardware

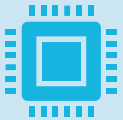
Monitor and record any alarms that happen

If the revolution sensor and speed the motor was set to disagree by more than 10%, stop the motor and quit the program

Write the alarms to a binary file

Step 0: Design

Our program is software, not hardware



We need to give its own idea of what the hardware is doing

So we'll make a struct which is our software's model of what the hardware is doing

Hardware Model

```
struct hw_state {
  char led1_on;  // bool-like
  char led2_on;
  char led3_on;
  char buzz_on;
  uint8_t motor_speed;} state;
/*
   motor speed is a uint8_t as
   our control byte only gives us
   4 bits for motor speed
   It'd probably be OK to use a char as
   well
*/
```

Step 1: Loading Data

Data is a list, so an obvious candidate data structure is the linked list

Each element must contain the type of instruction and parameters

But how do we get to the command line?

Command Line Input

- `main()` has two possible function signatures
- The second signature is `int main(int argc, char** argv)`
  - `argc` = Number of command line components
  - `argv` = Array of null-terminated strings of length `argc`
- So to get our input path
  - Check `argc` = 2 (1st argument is program name)
  - Input path string is `argv[1]`

program.c

```
int main(int argc, char** argv){
 if (argc != 2){
 printf("Wrong number of arguments,
  expected 1, got %d", argc);
 return 1;
 }
 char* input_path = argv[1]
 /* ... */
}
```

Reading in data with fscanf

- Each line of our input is of the form COMMAND_NUMBER COMMAND_ARG
  - For example 4 10
- Both are integers
- So we can read these with `fscanf(f, "%d %d", &x, &y);`
- Can actually do a little better and put them directly into the linked list!
- Remember that fscanf returns the amount of data it reads. When it returns -1, there's no more data and we can stop

program.c

```
struct cmdls {
 int cmd_no;
 int cmd_arg;
 stuct cmdls* next;
};

struct cmdls first_cmd;
struct cmdls* current = &first_cmd;
struct cmdls* next = &first_cmd;
while (fscanf(inp_fp, "%d %d", &current-
 >cmd_no, &current->cmd_arg) != -1){
 next = calloc(1, sizeof(struct cmdls));
 current->next = next;
 current = next;
}
```

Step 2: Main loop and updating our Model State

- A simple loop on the current pointer
  - i.e. loop while it's not the NULL (0) pointer
- To update our model state, one big switch which decodes command numbers
  - A bit too long to show here though

program.c

```
struct cmdls* current = &first_cmd;

while (current){
  switch (current->cmd_no){
    case 1:
      state.motor_speed = current->cmd_arg; break;
    case 2:
      state.buzz_on = current->cmd_arg; break;
    /* other cases */
    default:
      /* error handling */
  }
  /* the rest of the program */
}
```

Step 3: Telling the Device What to Do

- Need to construct an input byte
- Packed structs are one option… But compilers might still mangle the layout as we don't control it exactly
- We could use a `char` and work out what to put into it, but a `char` might not be one byte
- So instead we use bitwise operators and a `uint_8t` from `stdint.h`
- On each step we construct a new input byte based on our model

Using bit flags

- The device driver specifies a bunch of #defines in its header
- Each of these corresponds to a number which is a single bit switched on in binary i.e. $2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$…
- So we can combine the flags we want into a single byte using bitwise OR

driver.h

```
#define BUZZ_ON 8
#define LED1_ON 4
#define LED2_ON 2
#define LED3_ON 1
```

program.c

```
uint8_t inp_byte = 0;
if (state.led1_on) {
  inp_byte |= LED1_ON;
}
if (state.led2_on) {
  inp_byte |= LED2_ON;
}
if (state.led3_on) {
  inp_byte |= LED3_ON;
}
if (state.buzz_on) {
  inp_byte |= BUZZ_ON;
}
```

Using left shift

program.c

```
/* continued from previous
  slide*/

uint8_t shifted_speed =
  state.motor_speed << 4;
inp_byte |= shifted_speed;


driver_send_input(inp_byte);
```

The motor speed is the upper most bits of the input byte

So we need to put the motor speed of our current state there

We do this by using left shift 4 and bitwise OR

- As a bonus, if we somehow set our current speed too high, this will shift those bits off the top of the input byte. It'll probably still cause problems, but at least we won't tell the hardware to do something impossible

Input byte is now constructed, can send it hardware

Step 4: Checking for Alarms

- We need to extract the alarms from the output byte
- We do this by using bit masks
- The header has another set of #defines corresponding to the powers of two numbers
- So we can use bitwise AND with these to check if the bit is set in the output byte
- And then save the alarms that happened into another linked list
  - We'll add this to the linked list regardless of whether an alarm happened or not

driver.h

```
#define M_STATUS 8
#define ALARM_1 4
#define ALARM_2 2
#define ALARM_3 1
```

program.c

```
uint8_t out_byte =
  driver_get_byte();
struct alarm_info* =
  calloc(1, sizeof(struct
  alarm_info));
alarm_info->alarm_1 =
  out_byte & ALARM_1;
/* . . . */
prev_alarm_info->next =
  alarm_info;
prev_alarm_info =
  alarm_info;
```

Step 5: Checking the Motor Speed

program.c

```
/* continued */
uint8_t revs = (output_byte >>
  4) & 0xf;
// check we won't divide by zero
if (state.motor_speed){
float ratio = (float) revs /
  state.motor_speed;
if (ratio < 0.9 ||
     ratio > 1.1){
 /* somethings gone wrong */
 input_byte &= 0xf;
 driver_input_byte(input_byte);
 break; /* break out of our main
  loop */
}}
```

- The motor speed is in the top 4 bits of the output byte
- So we need to use right shift to put it back into the lower bits
- We then should AND with 15 – i.e. 00001111
  - Why? Depending on architecture and variable type, right shift might extend 1's into the rest of the variable if the top bit is 1. By and'ing with the lower 4 bits, we set the upper 4 bits to 0 and avoid this problem
  - Sometimes this will be written in hexadecimal as 0xf – hexadecimal can be convenient for writing binary
- We can then compare this with our model's record of what the speed is
- And if necessary, update out input byte to turn the motor off, send it, and break out of the loop

Step 6: Recording Alarms

- Need to be a bit careful here – we stored the alarms in a linked list
  - And while I didn't show it, it makes things easier if you create an "empty" header element which we then ignore here
- So that means we can't just fwrite them, we need to follow the pointers manually
- And fwrite the data from each struct
- It also means we really should document how we're writing them – just knowing the struct isn't enough
- And while we're at it, we can free the data we no longer need
  - But we need to be a bit careful not to accidentally free our next pointer as well!

program.c

```
/* continued – and after main loop*/

char output[3];
FILE* fp;
fp = fopen("out.data", "wb");
struct alarm_info* current_alarm =
first_alarm_info.next_alarm;
struct alarm_info* next_alarm;
while (current_alarm){
 output[0] = current_alarm->alarm_1;
 output[1] = current_alarm->alarm_2;
 output[2] = current_alarm->alarm_3;
 fwrite(output, sizeof(char), 3, fp);
 next_alarm = current_alarm->next_alarm;
 free(current_alarm);
 current_alarm = next_alarm;
}
fclose(fp);
```

Putting It All Together

Gives more than one slide of code, so instead of putting it here, it's in the assignment

General rule: In C, you'll almost certainly have to write more code than you think you should have to.

Conclusions

Everyone here should have seen

An example of a low-level program	How to manipulate bitfield data structures	One way in which drivers communicate with a program	How to avoid some of the pitfalls of data structures

That's pretty much it. If you've followed everything this far, you should have a pretty good idea of how to use C now.

Remember: If you need to, always look at the documentation. It'll explain how to do things.

Assignment

This lectures assignment is all about changing how this system works, and making it more efficient or useful

You'll make a program that can turn the text input files into binary, and then modify the program to read those binary files

Then another modification to only save the steps which triggered alarms, and when they happened

And finally, a way for a user to control the system manually

If you can do this, you'll have the basics down for how to design and program in C. Congratulations!