I²C EEPROM (AT24C256)

General instructions

The grove EEPROM module that is used in this assignment has pull-up resistors for I2C-bus. You can find the schematic of the Pico development board in the course workspace Documents/Project folder in case you need to check wiring. The development board silk screen has also some wiring information.

The EEPROM chip on the module is compatible with AT24C256. You can find the datasheet for AT24C256 in the course workspace Documents/Project folder. Pins A0, A1, and WP of the EEPROM chip are wired to the ground in the module.

The most important timing parameter when working with persistent memory chips is know how long a write operation will take to complete after the bus transaction is over. After a write transaction the chip will perform an internal write operation which prevents the chip from responding to any following transaction until the internal write is complete.

Start development step by step. First write functions for reading/writing a single byte to an address in the EEPROM. Test that the functions work with different addresses and different values for example write 0xA5 to address 0 and 0xBC to address 1 and read them back and check if you got the right values. Then test with two different values to see that programming and reading really works.

When single byte operations work then extend the functions to handle multibyte reads/writes and test them with some different arrays of bytes.

When you have multibyte reads and writes working then start developing the higher level logic.

Exercise 1 – Store the state of the program to EEPROM

Implement a program that switches LEDs on and off and remembers the state of the LEDs across reboot and/or power off. The program should work as follows:

- When the program starts it reads the state of the LEDs from EEPROM. If no valid state is found in the EEPROM the middle LED is switched on and the other two are switched off. The program must print number of seconds since power up and the state of the LEDs to stdout. Use time_us_64() to get a timestamp and convert that to seconds.
- Each of the buttons SW0, SW1, and SW2 on the development board is associated with an LED. When user presses a button, the corresponding LED toggles. Pressing and holding the button may not make the LED to blink or to toggle multiple times. When state of the LEDs is changed the new state must be printed to stdout with a number of seconds since program was started.
- When the state of an LEDs changes the program stores the state of all LEDs in the EEPROM and prints the state to LEDs to the debug UART. The program must employ a method to validate that settings read from the EEPROM are correct.
- The program must use the highest possible EEPROM address to store the LED state.

A simple way to validate the LED state is store it to EEPROM twice: normally (un-inverted) and inverted. When the state is read back both values are read, the inverted value is inverted after reading, and then the values are compared. If the values match then LED state was stored correctly in the EEPROM. By storing an inverted value, we can avoid case where both bytes are identical, a typical case with erased/out of the box memory, to be accepted as a valid value.

For example:

```
typedef struct ledstate {
    uint8_t state;
    uint8_t not_state;
} ledstate;

void set_led_state(ledstate *ls, uint8_t value)
{
    ls->state = value;
    ls->not_state = ~value;
}
```

Above is a helper function that sets both values in the struct. By using helper functions, we can ensure that both values are set correctly in the structure. In the same style we can write a function that validates the integrity of the structure that was read from the EEPROM.

```
bool led_state_is_valid(ledstate *ls) {
    return ls->state == (uint8_t) ~ls->not_state;
}
```

Typecast to `uint8_t` is needed for the compare to work correctly because operand of bitwise not gets promoted to an integer. Typecast to 8-bit value discards the extra bits that got added in the promotion.

Exercise 2 – Store log strings in EEPROM

Improve Exercise 1 by adding a persistent log that stores messages to EEPROM. When the program starts it writes “Boot” to the log and every time when state or LEDs is changed the state change message, as described in Exercise 1, is also written to the log.

The log must have the following properties:

- Log starts from address 0 in the EEPROM.
- First two kilobytes (2048 bytes) of EEPROM are used for the log.
- Each log entry is reserved 64 bytes.
    - First entry is at address 0, second at address 64, third at address 128, etc.
    - Log can contain up to 32 entries.
- A log entry consists of a string that contains maximum 61 characters, a terminating null character (zero) and two-byte CRC that is used to validate the integrity of the data. A maximum length log entry uses all 64 bytes. A shorter entry will not use all reserved bytes. The string must contain at least one character.
- When a log entry is written to the log, the string is written to the log including the terminating zero. Immediately after the terminating zero follows a 16-bit CRC, MSB first followed by LSB.
    - Entry is written to the first unused (invalid) location that is available.
    - If the log is full then the log is erased first and then entry is written to address 0.
- User can read the content of the log by typing read and pressing enter.
    - Program starts reading and validating log entries starting from address zero. If a valid string is found it is printed and program reads string from the next location.
    - A string is valid if the first character is not zero, there is a zero in the string before index 62, and the string passes the CRC validation.
    - Printing stops when an invalid string is encountered or the end log are is reached.
- User can erase the log by typing erase and pressing enter.
    - Erasing is done by writing a zero at the first byte of every log entry.

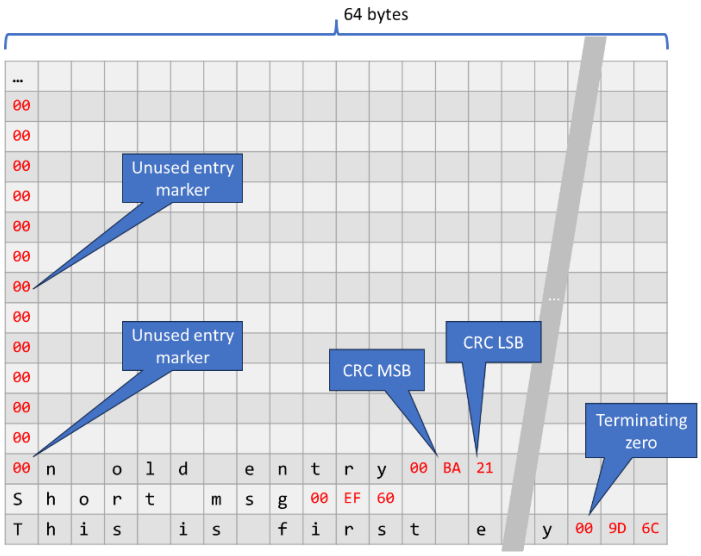

Figure 1 Structure of log in EEPROM

Use following code for CRC-calculation
(adapted from: https://stackoverflow.com/questions/10564491/function-to-calculate-a-crc16-checksum)

```
uint16_t crc16(const uint8_t *data_p, size_t length) {
    uint8_t x;
    uint16_t crc = 0xFFFF;

    while (length--) {
        x = crc >> 8 ^ *data_p++;
        x ^= x >> 4;
        crc = (crc << 8) ^ ((uint16_t) (x << 12)) ^ ((uint16_t) (x << 5)) ^ ((uint16_t) x);
    }
    return crc;
}
```

The property of CRC is such that when a CRC that is calculated over a number of data bytes is placed immediately after the bytes and the CRC is calculated over the data bytes plus the CRC-bytes the result is zero, provided that the data has not been modified after CRC was calculated.

For example:

```
uint8_t buffer[10] = { 51, 32, 93, 84, 75, 16, 17, 28 };
uint16_t crc = crc16(buffer, 8);
// put CRC after data
buffer[8] = (uint8_t) (crc >> 8);
buffer[9] = (uint8_t) crc;
// validate data
if(crc16(buffer, 10) != 0) {
    printf("Error\n");
}
```