SSD1306 OLED Controller

SSD1306 0.96" I2C OLED display is very popular as it is used in many project to display system status as it has multiple communication protocols and consumes very low power. It mainly exposes I2C interface or SPI interface. SH1106 is another controller which is similar to SSD1306, and in many cases, it shares the same control commands.

#oled #ssd1306

Last update: 2021-07-06 14:39:33

Table of Content

1. SSD1306 Controller

SSD1306 is a single-chip CMOS OLED/PLED driver to control display with 128 segments and 64 commons. This IC is designed for Common Cathode type OLED panel.

The SSD1306 embeds with contrast control, display RAM and oscillator, which reduces the number of external components and power consumption. It has 256-step brightness control. Data/Commands are sent from general MCU through the hardware selectable 6800/8000 series compatible Parallel Interface, I2C interface or Serial Peripheral Interface. It is suitable for many compact portable applications, such as mobile phone sub-display, MP3 player and calculator, etc.

1.1. Features

- Resolution: 128 x 64 dots
- VDD = 1.65V to 3.3V for IC logic
- On-Chip Oscillator
- 256 step contrast brightness
- Embedded 128 x 64 bit SRAM display buffer
- Programmable Frame Rate and Multiplexing Ratio
- Row Re-mapping and Column Re-mapping to rotate the screen
- Scrolling function in both horizontal and vertical direction
- Pin selectable MCU Interfaces:
    - 8-bit 6800/8080-series parallel interface
    - 3 /4 wire Serial Peripheral Interface
    - I2C Interface

SSD1306 128x64 OLED

SSD1306 128x32 OLED

1.2. I2C Interface

SSD1306 has to recognize the slave address before transmitting or receiving any information by the I2C-bus. The device will respond to the slave address following by the slave address bit (`SA0` bit) and the read/write select bit (`R/W#` bit) with the following byte format:

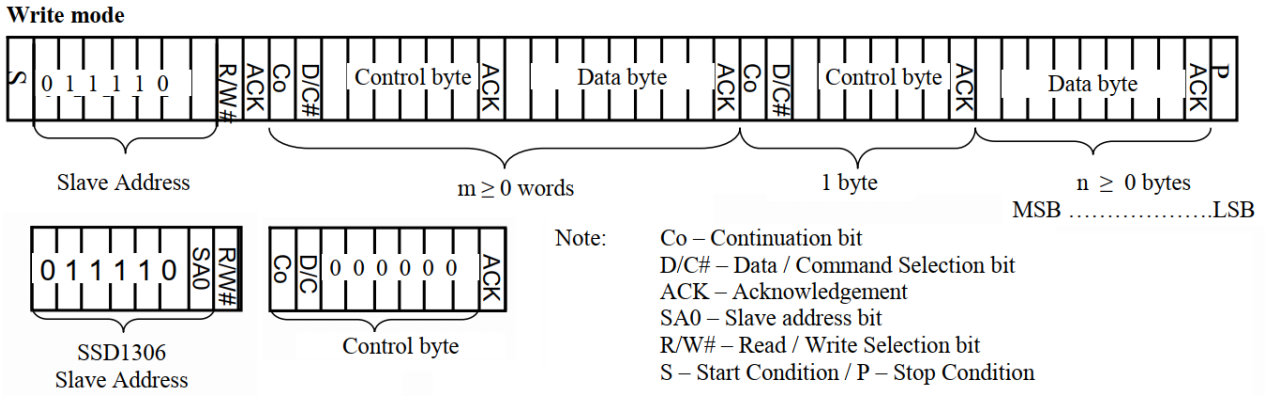
```
b7  b6  b5  b4  b3  b2  b1  b0
0   1   1   1   1   0   SA0 R/W#
```

It means the address of OLED is either `0x3C` or `0x3D` in 7-bit address mode, or `0x78` and `0x7A` in 8-bit address mode.

> Use I2C Scanner to see the address of connected devices on I2C bus.

1.2.1. I2C Message

In the **Write Mode**, the Master device has to transfer a control byte following by one or more data bytes in a transfer (between the start and the stop bit).



SSD1306 I2C Message

Control Byte: Have the `Co#` bit, the `D/C#` bit, and six `0` bits:

- Bit `Co#` set to `0` means the next stream bytes. Bit `Co#` set to `1` means the next byte is a single byte
- Bit `D/C#` set to `0` means the next bytes is a Command and Command params. Bit `D/C#` set to `1` means the next bytes is Data which will be stored in the Embedded GDDRAM.

Therefore, the Control Byte can be:

- Single Command mode: `0x80`
- Stream Command mode: `0x00`
- Single Data mode: `0xC0`
- Stream Data mode: `0x40`

1.3. Display clock

The Display Clock for the Display Timing Generator is derived from the operation CLK (either the internal RC Oscillator or an external Clock).

The command `0xD5` and its one byte param `A` are used to select the internal RC frequency F_{osc} via `A[7:4]`, and the clock divide ratio `D` via `A[3:0]`.

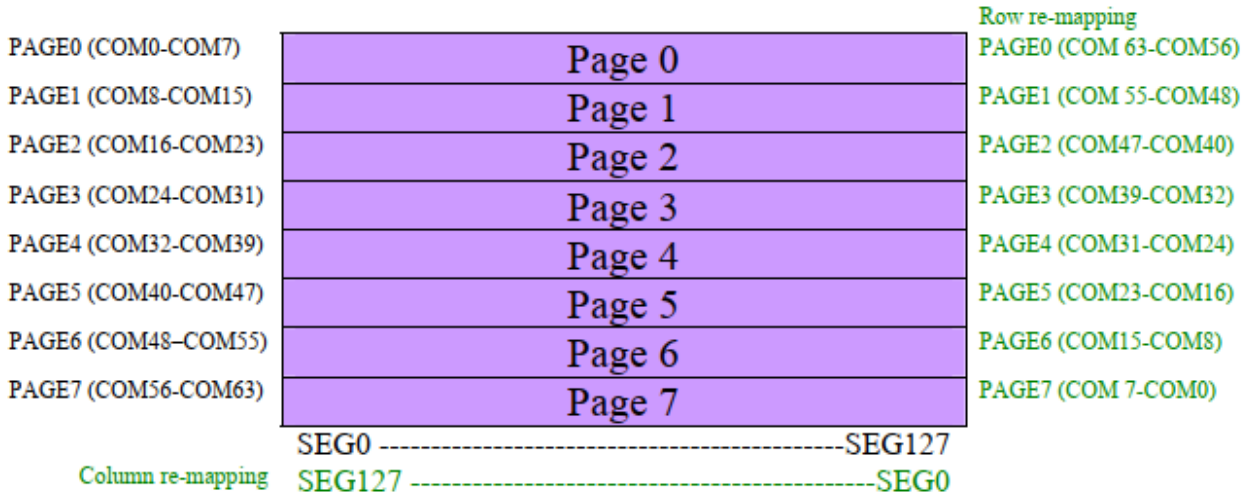
1.4. Graphic Display Data RAM

> **Dimension**
>
> **Segments** mean **Columns**, define the **Width**.
> **Commons** mean **Rows**, define the **Height**.

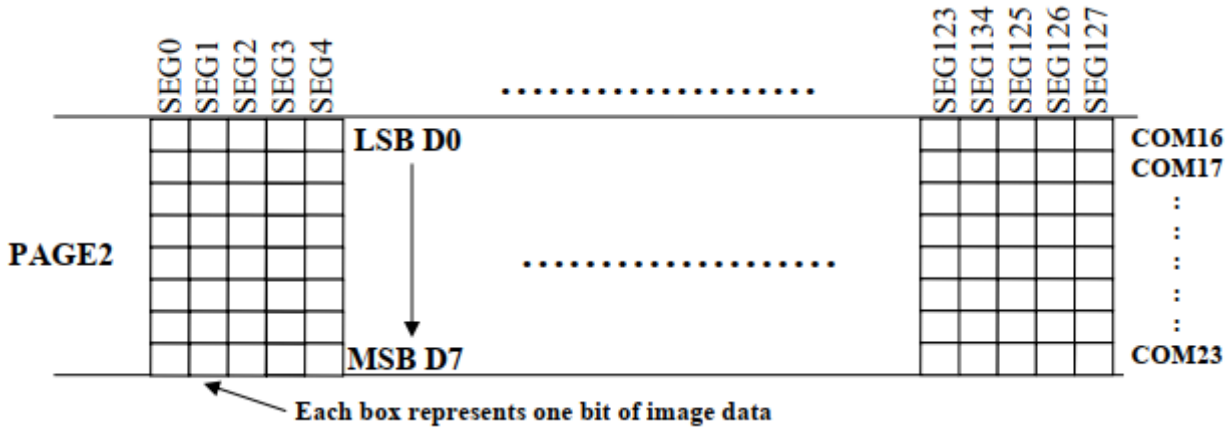
The width of the screen is the number of activated Segments. There are 128 segments, from `SEG0` to `SEG127`.

The height of the screen is the number of activate Commons. There are 64 commons, from `COM0` to `COM63`.

The GDDRAM is a bit mapped static RAM holding the bit pattern to be displayed. The size of the RAM is 128 x 64 bits. The Commons are grouped in to 8 pages, from `PAGE0` to `PAGE7`. Each segment in a page is one BYTE which actually drives 8 commons. Data bit `D0` is written into the top row, while data bit `D7` is written into bottom row.



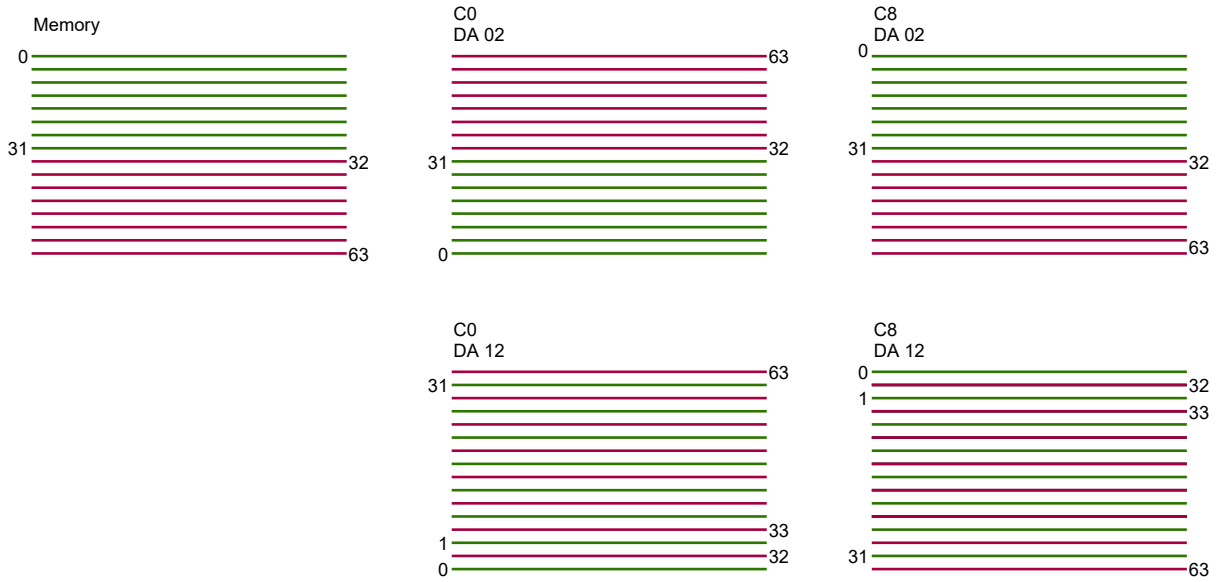
Pages with its Commons



Segments in a Page and the corresponding bit of a pixel

The command `0xA8` and its one byte param `A` are used to set the number of activated rows (in 0-index) which is usually the height of the desired screen's resolution.

The command `0xDA` and its one byte param `A` are used to set the **Memory Scan Mode** of the commons (rows) from the memory to the display. If `A=0x02`, each half of memory will be pushed on a half of screen. If `A=0x12`, rows are interleaved.



Memory scan mode

128 x 32 Double Buffer

SSD1306 Controller divides the GDDRAM 128x64 into 2 halves with different scan modes. Therefore, if using only half resolution at 128x32, 2 halves of memory can be used as a double buffer to reduce the flicker or to show different images without redrawing. Read more in this project.

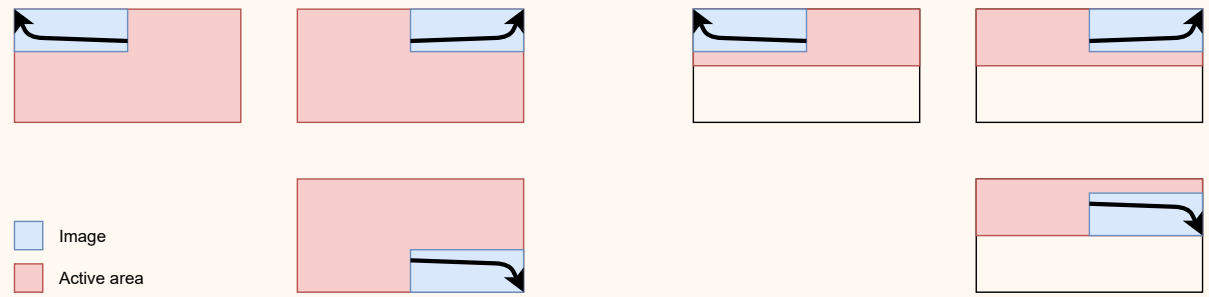
Mirroring

The command `0xA0` and `0xA1` are use to remap the segments (columns). This causes the screen is vertically flipped.

The command `0xC0` and `0xC8` are use to remap the commons (rows). This causes the screen is horizontally flipped.

Flip behavior

Horizontal Flip affects only activated commons (rows). Please note the case that the activated rows is less then the screen height.



Flip behavior in different activating rows

1.5. Memory Addressing Mode

When ever a data is written into GDDRAM, the current memory point is increased by one to point to the next bytes of rows in the next segment. There are 3 different memory addressing mode in SSD1306: page addressing mode, horizontal addressing mode and vertical addressing mode.

The command `0x20` and its one byte param `A` are used to set the **Memory Addressing Mode**.

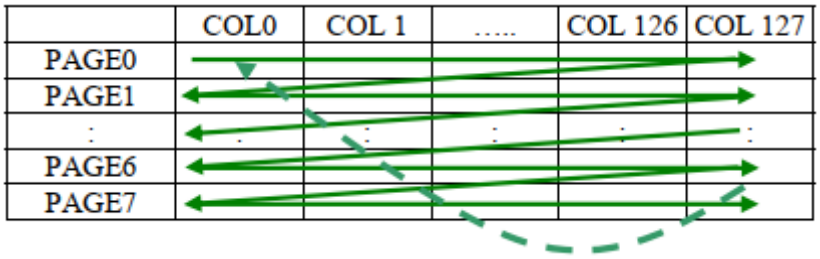
- If `A=0` , it sets Horizontal Mode
- If `A=1` , it sets Vertical Mode
- If `A=2` , it sets Page Mode

Horizontal Mode

- In horizontal addressing mode, after the display RAM is read/written, the column address pointer is increased automatically by 1.
- If the column address pointer reaches column end address, the column address pointer is reset to column start address and page address pointer is increased by 1.
- When both column and page address pointers reach the end address, the pointers are reset to column start address and page start address

The command `0x21` and its two bytes param `A` and `B` are use to set the start and the end segment (column). This also set the address pointer to the start segment.

The command `0x22` and its two bytes param `A` and `B` to set the start and the end page (rows). This also set the address pointer to the start page.

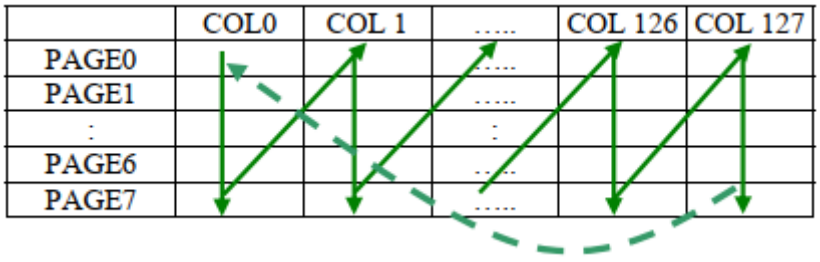


Memory Horizontal Addressing mode

Vertical Mode

- In vertical addressing mode, after the display RAM is read/written, the page address pointer is increased automatically by 1.
- If the page address pointer reaches the page end address, the page address pointer is reset to page start address and column address pointer is increased by 1.
- When both column and page address pointers reach the end address, the pointers are reset to column start address and page start address

The command `0x21` and its two bytes param `A` and `B` are use to set the start and the end segment (column). This also set the address pointer to the start segment.
The command `0x22` and its two bytes param `A` and `B` to set the start and the end page (rows). This also set the address pointer to the start page.



Memory Vertical Addressing mode

Page Mode

- In page addressing mode, after the display RAM is read/written, the column address pointer is increased automatically by 1.
- If the column address pointer reaches column end address, the column address pointer is reset to column start address and page address pointer is not changed.

- Users have to set the new page and column addresses in order to access the next page RAM content

The command from `0x00 ~ 0x0F` are used to set the lower nibble of the start segment (column). The command from `0x10 ~ 0x1F` are used to set the higher nibble of the start segment (column).

The command from `0xB0 ~ 0xB7` are used to set the active page.

1.6. Brightness

The command `0x81` and its one byte param `A` are used to set the brightness value which is in range `[0:255]`.

1.7. Display Start Line

The command from `0x40 ~ 0x7F` are used to set the starting address of display RAM. With value equal to 0, RAM row 0 is mapped to COM0. With value equal to 1, RAM row 1 is mapped to COM0 and so on.

Then command `0xD3` and its one byte param is to specify the mapping of the display start line to one of `COM0` to `COM63` (assuming that `COM0` is the display start line then the display start line register is equal to 0).

For example, to move the `COM16` towards the `COM0` direction by 16 lines the 6-bit data in the second byte should be given as `010000b`. To move in the opposite direction by 16 lines the 6-bit data should be given by 64 – 16, so the second byte would be `100000b`.

1.8. Display content

There are two modes of display ON can be selected.

The command `0xA4` will set screen to display from RAM;

The command `0xA5` will force screen to display all pixels ON regardless of the contents of the display data RAM.

1.9. Display Inverse

The command `0xA6` will set screen to display in normal mode. In normal display a RAM data of 1 indicates an “ON” pixel while in inverse display a RAM data of 0 indicates an “ON” pixel.

The command `0xA7` will force screen to inverse the display.

1.10. Display On/Off

The command `0xAE` will set screen to off; the segment and common output are in VSS state and high impedance state, respectively

The command `0xAF` will turn on the screen.

1.11. Charge Pump (DC-DC)

The internal regulator circuit in SSD1306 accompanying only 2 external capacitors can generate a 7.5V voltage supply, V_{CC}, from a low voltage supply input, V_{BAT}. The V_{CC} is the voltage supply to the OLED driver block. This is a switching capacitor regulator circuit, designed for handheld applications. This regulator can be turned on/off by software command setting.

The command `0x8D` and its one byte param is used to turn On or Off the OLED driver block. If the param is `0x14`, it enables the charge pump. If the param is `0x10`, it disables the charge pump.

Note that the command to turn on OLED `0xAF` should be sent after enabling the charge pump.

2. Implementation

Below implementation should not depend on any specific protocol. It can work without knowing the protocol, except 4 functions: `SSD1306_Init()`, `SSD1306_CMD_Send()`, `SSD1306_DATA_Send()` and `SSD1306_Screen_Update()`;

2.1. Protocol Settings

SSD1306 can accept I2C speed at Standard Mode (100 Kbps), Fast Mode (400 Kbps), and Fast Plus Mode (1 Mbps). Note to set Pull-up resistors on the `SCL` and `SDA` pins when there is no external pull-up resistors.

> **SPI interface**
>
> Using SPI will achieve higher speed upto 15 Mbps. Refer to this F051R8_SPI_OLED_SH1106.

2.2. Start-up commands

It's highly recommended to send a full set of commands to setup OLED when startup or reboot. This help to get right configs. This is a minimal setup for setting up OLED with resolution being 128x32, Horizontal Memory Addressing Mode, no flipping.

A buffer with the first byte being the CONTROL COMMAND byte is used to store a list of commands:

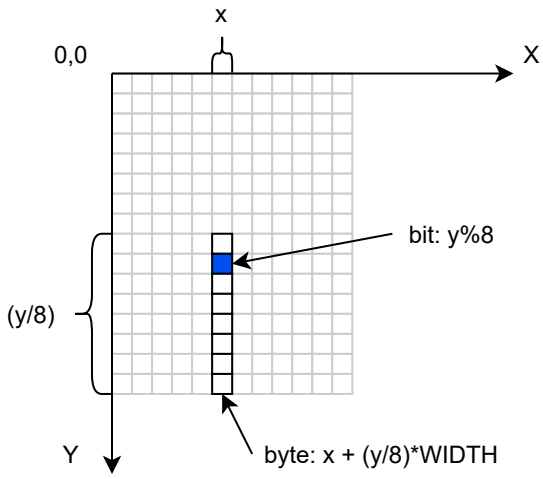
```
uint8_t ssd1306_cmd_buffer[SSD1306_BUFFER_CMD_MAX + 1] = {0x00, 0};
```

Then add the commands and their params, as below:

```
void SSD1306_Init(I2C_HandleTypeDef *hi2c) {
  // startup sequence
  SSD1306_CMD_Int();
  // Display off
  SSD1306_CMD_Add0(0xAE);
  // OSC default=0x81
  SSD1306_CMD_Add1(0xD5, 0x81);
  // Brightness in range 0~255, default=0x7F
  SSD1306_CMD_Add1(0x81, 0x7F);
  // Memory Address Mode: Horizontal=0, Vertical=1, Page=default=2
  SSD1306_CMD_Add1(0x20, 0);
  // Set Display Offset in range 0~63
  SSD1306_CMD_Add1(0xD3, 0);
  // Set Display start line in range 0x40~0x7F
  SSD1306_CMD_Add0(0x40);
  // Set multiplex number (activated rows): rows=height-1, default=63
  SSD1306_CMD_Add1(0xA8, SSD1306_HEIGHT-1);
  // Reduce a half of height
  SSD1306_CMD_Add1(0xDA, 0x02);
  // Segment (Column) normal mode, Inverse=0xA1
  SSD1306_CMD_Add0(0xA0);
  // Common (Row) normal mode, inverse=0xC8
  SSD1306_CMD_Add0(0xC0);
  // Display normal mode, inverse=0xA7
  SSD1306_CMD_Add0(0xA6);
  // Disable Scroll
  SSD1306_CMD_Add0(0x2E);
  // Pre-charge period, default=2
  SSD1306_CMD_Add1(0xD9, 2);
  // Charge Pump: On=0x14, Off=0x10
  SSD1306_CMD_Add1(0x8D, 0x14);
  // Display on
  SSD1306_CMD_Add0(0xAF);
  SSD1306_CMD_Send(hi2c);
```

2.3. Screen Buffer

The GDDRAM is accessible at byte level only, therefore, to manipulate bits, it's better to use an external memory which have the same size of the GDDRAM and then draw on that buffer. When screen needs to be updated, that buffer will be sent to GDDRAM.



Memory Mapping

A memory buffer is used to hold the screen buffer with the first byte is the DATA CONTROL byte:

```
uint8_t ssd1306_data_buffer[SSD1306_BUFFER_DATA_MAX + 1] = {0x40, 0 };
```

The whole buffer is sent to SSD1306 after setting the address point to first byte (Segment = 0, Page = 0):

> ⚠ **Separate Command and Data stream**
>
> When an I2C transfer is started, the first byte is used to set the type of whole transfer. From the second byte, they are considered as the data bytes of the command. It could not be mixed between a command stream and a data stream.

```
void SSD1306_Screen_Update(I2C_HandleTypeDef *hi2c) {
    SSD1306_CMD_Int(); // point to the starting byte of screen
    SSD1306_CMD_Add2(0x21, 0, SSD1306_WIDTH-1); // Segment (column)
    SSD1306_CMD_Add2(0x22, 0, (SSD1306_HEIGHT/8)-1); // Page (row)
    SSD1306_CMD_Send(hi2c);

    SSD1306_DATA_Send(hi2c); // Send screen buffer
}
```

2.4. Draw a pixel

As seen in the above picture of memory mapping, any pixel at location `(x, y)` can be set or unset with below method:

- Byte location: `ssd1306_data_buffer[1+ x + (y >> 3) * SSD1306_WIDTH]`
- Bit location: `(1 << (y % 8))`

Therefore, to set a pixel:

```
ssd1306_data_buffer[1+ x + (y >> 3) * SSD1306_WIDTH] |= (1 << (y % 8));
```

and to clear a pixel:

```
ssd1306_data_buffer[1+ x + (y >> 3) * SSD1306_WIDTH] &= ~(1 << (y % 8));
```

The `SSD1306_DrawPixel` also takes a `color` param to present a set/cleared pixel: White = set, Black = clear.

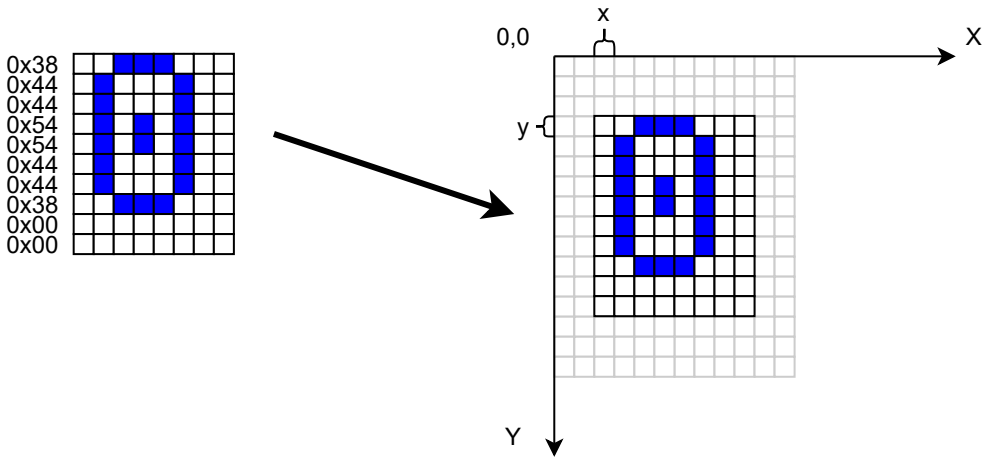
```
void SSD1306_DrawPixel(int16_t x, int16_t y, SSD1306_COLOR_t color) {
  if (x < 0 || x >= SSD1306_WIDTH || y < 0 || y >= SSD1306_HEIGHT) {
    return;
  }
  if(color == SSD1306_WHITE) {
    ssd1306_data_buffer[1+x+(y>>3)*SSD1306_WIDTH] |= (1 << (y % 8));
  } else {
    ssd1306_data_buffer[1+x+(y>>3)*SSD1306_WIDTH] &= ~(1 << (y % 8));
  }
}
```

Based on the function `SSD1306_DrawPixel()`, other shapes can be drawn too. Here are some functions to draw primitive shapes:

```
void SSD1306_DrawLine(              int16_t x0, int16_t y0,
                                    int16_t x1, int16_t y1,
                                    SSD1306_COLOR_t color);
void SSD1306_DrawRectangle(         int16_t x, int16_t y,
                                    int16_t w, int16_t h,
                                    SSD1306_COLOR_t color);
void SSD1306_DrawFilledRectangle(   int16_t x, int16_t y,
                                    int16_t w, int16_t h,
                                    SSD1306_COLOR_t color);
void SSD1306_DrawTriangle(          int16_t x1, int16_t y1,
                                    int16_t x2, int16_t y2,
                                    int16_t x3, int16_t y3,
                                    SSD1306_COLOR_t color);
void SSD1306_DrawFilledTriangle(    int16_t x1, int16_t y1,
                                    int16_t x2, int16_t y2,
                                    int16_t x3, int16_t y3,
                                    SSD1306_COLOR_t color);
void SSD1306_DrawCircle(            int16_t x0, int16_t y0,
                                    int16_t r,
                                    SSD1306_COLOR_t color);
void SSD1306_DrawFilledCircle(      int16_t x0, int16_t y0,
                                    int16_t r,
                                    SSD1306_COLOR_t color);
```

2.5. Bitmap & Font & String

Bitmap file can be easily read out, as it is sliced into bytes. For example, a character `0` below are encoded into 10 bytes which will be mapped to 7px width by 10px height character. It also can be a 8px width by 10px height bitmap.



Bitmap Mapping

```
void SSD1306_DrawBitmap(int16_t x, int16_t y,
    const uint8_t* bitmap, int16_t w, int16_t h) {

    int16_t byteWidth = (w + 7) / 8; // Bitmap scan line pad = whole byte
    uint8_t byte = 0;

    for(int16_t j=0; j<h; j++, y++) {
        for(int16_t i=0; i<w; i++) {
            if(i & 7) {
                byte <<= 1; // shift to get a bit
            } else {
                // read one byte when i=0
                byte = bitmap[j * byteWidth + i / 8];
            }
            // bit is 1
            if(byte & 0x80) {
              SSD1306_DrawPixel(x+i, y, SSD1306_WHITE);
            }
        }
    }
}
```

Here is an example of a 7x10 font is saved, each 10 bytes represent a character, and the first character is *space* causing an lookup offset of 32 when retrieving bitmap for a character.

If the width of a character is bigger than 8, the array will have `uint16_t` type to hold at max 16 bits.

```
static const int8_t Font7x10 [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  // sp (32)
...
0x38, 0x44, 0x44, 0x54, 0x54, 0x44, 0x44, 0x38, 0x00, 0x00,  // 0
0x10, 0x30, 0x50, 0x10, 0x10, 0x10, 0x10, 0x10, 0x00, 0x00,  // 1
0x38, 0x44, 0x44, 0x04, 0x08, 0x10, 0x20, 0x7C, 0x00, 0x00,  // 2
...
```

To write a character:

```
void SSD1306_WriteChar(int16_t x, int16_t y, char ch, FontDef_t* Font) {
    int16_t x0, y0, b;

    // Translate font to screen buffer
    for (y0 = 0; y0 < Font->height; y0++) {
        b = Font->data[(ch - 32) * Font->height + y0];
        for (x0 = 0; x0 < Font->width; x0++) {
            if ((b << x0) & 0x8000) {
              SSD1306_DrawPixel(x + x0, y + y0, SSD1306_WHITE);
            }
        }
    }
}
```

To write a string:

```
void SSD1306_WriteString(int16_t x, int16_t y, char* str, FontDef_t* Font) {
    int16_t n = 0;
    // Write until null-byte or the first cutoff char
    while (*str) {
        SSD1306_WriteChar(x + n*Font->width, y, *str, Font);
        n++;
        str++;
    }
}
```

Transferring modes

The communication with screen can be done in either Polling, Interrupt or DMA mode. In DMA mode, the interrupt of I2C port must be enabled as it is in the Interrupt mode. A callback `HAL_I2C_MasterTxCpltCallback()` is called when a transfer is done.

It should check whenever the screen buffer is dirty to update on display to avoid overhead of data communication.

I2C is dropped when using Interrupt at Ultra fast mode

Take an example of I2C Ultra Fast mode at 5 Mbps, which will generate 625000 interrupts per second (!) in case of 8-bit transfer. An STM32F0xx running at 48 MHz will have a room of only 77

cycles to process an Interrupt. However, calling an interrupt takes 12 cycle, exiting an interrupt costs 10 cycles (ideal state without waiting). Therefore, an interrupt code must be served in only 55 cycles !

Should use DMA when the bit rate is high, because DMA only causes 2 interrupts: “Half-transfer” and “Transfer-Complete” in a routine of transfer a large data buffer.