OLED

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/i2c.h"
#include "raspberry26x32.h"
#include "ssd1306_font.h"

/* Example code to talk to an SSD1306-based OLED display

   The SSD1306 is an OLED/PLED driver chip, capable of driving displays up to
   128x64 pixels.

   NOTE: Ensure the device is capable of being driven at 3.3v NOT 5v. The Pico
   GPIO (and therefore I2C) cannot be used at 5v.

   You will need to use a level shifter on the I2C lines if you want to run the
   board at 5v.

   Connections on Raspberry Pi Pico board, other boards may vary.

   GPIO PICO_DEFAULT_I2C_SDA_PIN (on Pico this is GP4 (pin 6)) -> SDA on display
   board
   GPIO PICO_DEFAULT_I2C_SCL_PIN (on Pico this is GP5 (pin 7)) -> SCL on
   display board
   3.3v (pin 36) -> VCC on display board
   GND (pin 38)  -> GND on display board
*/

// Define the size of the display we have attached. This can vary, make sure you
// have the right size defined or the output will look rather odd!
// Code has been tested on 128x32 and 128x64 OLED displays
#define SSD1306_HEIGHT              64
#define SSD1306_WIDTH               128

#define SSD1306_I2C_ADDR            _u(0x3C)

// 400 is usual, but often these can be overclocked to improve display response.
// Tested at 1000 on both 32 and 84 pixel height devices and it worked.
#define SSD1306_I2C_CLK             400
//#define SSD1306_I2C_CLK             1000


// commands (see datasheet)
#define SSD1306_SET_MEM_MODE        _u(0x20)
#define SSD1306_SET_COL_ADDR        _u(0x21)
#define SSD1306_SET_PAGE_ADDR       _u(0x22)
#define SSD1306_SET_HORIZ_SCROLL    _u(0x26)
#define SSD1306_SET_SCROLL          _u(0x2E)

#define SSD1306_SET_DISP_START_LINE _u(0x40)

#define SSD1306_SET_CONTRAST        _u(0x81)
#define SSD1306_SET_CHARGE_PUMP     _u(0x8D)

#define SSD1306_SET_SEG_REMAP       _u(0xA0)
#define SSD1306_SET_ENTIRE_ON       _u(0xA4)
#define SSD1306_SET_ALL_ON          _u(0xA5)
#define SSD1306_SET_NORM_DISP       _u(0xA6)
#define SSD1306_SET_INV_DISP        _u(0xA7)
#define SSD1306_SET_MUX_RATIO       _u(0xA8)
#define SSD1306_SET_DISP            _u(0xAE)
#define SSD1306_SET_COM_OUT_DIR     _u(0xC0)
#define SSD1306_SET_COM_OUT_DIR_FLIP _u(0xC0)

#define SSD1306_SET_DISP_OFFSET     _u(0xD3)
#define SSD1306_SET_DISP_CLK_DIV    _u(0xD5)
#define SSD1306_SET_PRECHARGE       _u(0xD9)
#define SSD1306_SET_COM_PIN_CFG     _u(0xDA)
#define SSD1306_SET_VCOM_DESEL      _u(0xDB)

#define SSD1306_PAGE_HEIGHT         _u(8)
#define SSD1306_NUM_PAGES           (SSD1306_HEIGHT / SSD1306_PAGE_HEIGHT)
#define SSD1306_BUF_LEN             (SSD1306_NUM_PAGES * SSD1306_WIDTH)

#define SSD1306_WRITE_MODE         _u(0xFE)
#define SSD1306_READ_MODE          _u(0xFF)


struct render_area {
    uint8_t start_col;
    uint8_t end_col;
    uint8_t start_page;
    uint8_t end_page;

    int buflen;
};

void calc_render_area_buflen(struct render_area *area) {
    // calculate how long the flattened buffer will be for a render area
    area->buflen = (area->end_col - area->start_col + 1) * (area->end_page - area->start_page + 1);
}
```

```
#ifdef i2c_default

void SSD1306_send_cmd(uint8_t cmd) {
    // I2C write process expects a control byte followed by data
    // this "data" can be a command or data to follow up a command
    // Co = 1, D/C = 0 => the driver expects a command
    uint8_t buf[2] = {0x80, cmd};
    i2c_write_blocking(i2c_default, (SSD1306_I2C_ADDR & SSD1306_WRITE_MODE), buf, 2, false);
}

void SSD1306_send_cmd_list(uint8_t *buf, int num) {
    for (int i=0;i<num;i++)
        SSD1306_send_cmd(buf[i]);
}

void SSD1306_send_buf(uint8_t buf[], int buflen) {
    // in horizontal addressing mode, the column address pointer auto-increments
    // and then wraps around to the next page, so we can send the entire frame
    // buffer in one gooooooo!

    // copy our frame buffer into a new buffer because we need to add the control byte
    // to the beginning

    uint8_t *temp_buf = malloc(buflen + 1);

    temp_buf[0] = 0x40;
    memcpy(temp_buf+1, buf, buflen);

    i2c_write_blocking(i2c_default, (SSD1306_I2C_ADDR & SSD1306_WRITE_MODE), temp_buf, buflen + 1, false);

    free(temp_buf);
}

void SSD1306_init() {
    // Some of these commands are not strictly necessary as the reset
    // process defaults to some of these but they are shown here
    // to demonstrate what the initialization sequence looks like
    // Some configuration values are recommended by the board manufacturer

    uint8_t cmds[] = {
        SSD1306_SET_DISP,               // set display off
        /* memory mapping */
        SSD1306_SET_MEM_MODE,           // set memory address mode 0 = horizontal, 1 = vertical, 2 = page
        0x00,                           // horizontal addressing mode
        /* resolution and layout */
        SSD1306_SET_DISP_START_LINE,    // set display start line to 0
        SSD1306_SET_SEG_REMAP | 0x01,   // set segment re-map, column address 127 is mapped to SEG0
        SSD1306_SET_MUX_RATIO,          // set multiplex ratio
        SSD1306_HEIGHT - 1,             // Display height - 1
        SSD1306_SET_COM_OUT_DIR | 0x08, // set COM (common) output scan direction. Scan from bottom up, COM[N-1] to COM0
        SSD1306_SET_DISP_OFFSET,        // set display offset
        0x00,                           // no offset
        SSD1306_SET_COM_PIN_CFG,        // set COM (common) pins hardware configuration. Board specific magic number.
                                        // 0x02 Works for 128x32, 0x12 Possibly works for 128x64. Other options 0x22, 0x32
#if ((SSD1306_WIDTH == 128) && (SSD1306_HEIGHT == 32))
        0x02,
#elif ((SSD1306_WIDTH == 128) && (SSD1306_HEIGHT == 64))
        0x12,
#else
        0x02,
#endif
        /* timing and driving scheme */
        SSD1306_SET_DISP_CLK_DIV,       // set display clock divide ratio
        0x80,                           // div ratio of 1, standard freq
        SSD1306_SET_PRECHARGE,          // set pre-charge period
        0xF1,                           // Vcc internally generated on our board
        SSD1306_SET_VCOM_DESEL,         // set VCOMH deselect level
        0x30,                           // 0.83xVcc
        /* display */
        SSD1306_SET_CONTRAST,           // set contrast control
        0xFF,
        SSD1306_SET_ENTIRE_ON,          // set entire display on to follow RAM content
        SSD1306_SET_NORM_DISP,           // set normal (not inverted) display
        SSD1306_SET_CHARGE_PUMP,        // set charge pump
        0x14,                           // Vcc internally generated on our board
        SSD1306_SET_SCROLL | 0x00,      // deactivate horizontal scrolling if set. This is necessary as memory writes will
corrupt if scrolling was enabled
        SSD1306_SET_DISP | 0x01, // turn display on
    };

    SSD1306_send_cmd_list(cmds, count_of(cmds));
}

void SSD1306_scroll(bool on) {
    // configure horizontal scrolling
    uint8_t cmds[] = {
        SSD1306_SET_HORIZ_SCROLL | 0x00,
        0x00, // dummy byte
        0x00, // start page 0
        0x00, // time interval
        0x03, // end page 3 SSD1306_NUM_PAGES ??
        0x00, // dummy byte
        0xFF, // dummy byte
        SSD1306_SET_SCROLL | (on ? 0x01 : 0) // Start/stop scrolling
    };

    SSD1306_send_cmd_list(cmds, count_of(cmds));
```

```
}

void render(uint8_t *buf, struct render_area *area) {
    // update a portion of the display with a render area
    uint8_t cmds[] = {
        SSD1306_SET_COL_ADDR,
        area->start_col,
        area->end_col,
        SSD1306_SET_PAGE_ADDR,
        area->start_page,
        area->end_page
    };

    SSD1306_send_cmd_list(cmds, count_of(cmds));
    SSD1306_send_buf(buf, area->buflen);
}

static void SetPixel(uint8_t *buf, int x,int y, bool on) {
    assert(x >= 0 && x < SSD1306_WIDTH && y >=0 && y < SSD1306_HEIGHT);

    // The calculation to determine the correct bit to set depends on which address
    // mode we are in. This code assumes horizontal

    // The video ram on the SSD1306 is split up in to 8 rows, one bit per pixel.
    // Each row is 128 long by 8 pixels high, each byte vertically arranged, so byte 0 is x=0, y=0->7,
    // byte 1 is x = 1, y=0->7 etc

    // This code could be optimised, but is like this for clarity. The compiler
    // should do a half decent job optimising it anyway.

    const int BytesPerRow = SSD1306_WIDTH ; // x pixels, 1bpp, but each row is 8 pixel high, so (x / 8) * 8

    int byte_idx = (y / 8) * BytesPerRow + x;
    uint8_t byte = buf[byte_idx];

    if (on)
        byte |=  1 << (y % 8);
    else
        byte &= ~(1 << (y % 8));

    buf[byte_idx] = byte;
}
// Basic Bresenhams.
static void DrawLine(uint8_t *buf, int x0, int y0, int x1, int y1, bool on) {

    int dx =  abs(x1-x0);
    int sx = x0<x1 ? 1 : -1;
    int dy = -abs(y1-y0);
    int sy = y0<y1 ? 1 : -1;
    int err = dx+dy;
    int e2;

    while (true) {
        SetPixel(buf, x0, y0, on);
        if (x0 == x1 && y0 == y1)
            break;
        e2 = 2*err;

        if (e2 >= dy) {
            err += dy;
            x0 += sx;
        }
        if (e2 <= dx) {
            err += dx;
            y0 += sy;
        }
    }
}

static inline int GetFontIndex(uint8_t ch) {
    if (ch >= 'A' && ch <='Z') {
        return  ch - 'A' + 1;
    }
    else if (ch >= '0' && ch <='9') {
        return  ch - '0' + 27;
    }
    else return  0; // Not got that char so space.
}

static uint8_t reversed[sizeof(font)] = {0};

static uint8_t reverse(uint8_t b) {
   b = (b & 0xF0) >> 4 | (b & 0x0F) << 4;
   b = (b & 0xCC) >> 2 | (b & 0x33) << 2;
   b = (b & 0xAA) >> 1 | (b & 0x55) << 1;
   return b;
}
static void FillReversedCache() {
    // calculate and cache a reversed version of fhe font, because I defined it upside down...doh!
    for (int i=0;i<sizeof(font);i++)
        reversed[i] = reverse(font[i]);
}

static void WriteChar(uint8_t *buf, int16_t x, int16_t y, uint8_t ch) {
    if (reversed[0] == 0)
        FillReversedCache();
```

```
    if (x > SSD1306_WIDTH - 8 || y > SSD1306_HEIGHT - 8)
        return;

    // For the moment, only write on Y row boundaries (every 8 vertical pixels)
    y = y/8;

    ch = toupper(ch);
    int idx = GetFontIndex(ch);
    int fb_idx = y * 128 + x;

    for (int i=0;i<8;i++) {
        buf[fb_idx++] = reversed[idx * 8 + i];
    }
}

static void WriteString(uint8_t *buf, int16_t x, int16_t y, char *str) {
    // Cull out any string off the screen
    if (x > SSD1306_WIDTH - 8 || y > SSD1306_HEIGHT - 8)
        return;

    while (*str) {
        WriteChar(buf, x, y, *str++);
        x+=8;
    }
}



#endif

int main() {
    stdio_init_all();

#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SCL_PIN)
#warning i2c / SSD1306_i2d example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    // useful information for picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C));
    bi_decl(bi_program_description("SSD1306 OLED driver I2C example for the Raspberry Pi Pico"));

    printf("Hello, SSD1306 OLED display! Look at my raspberries..\n");

    // I2C is "open drain", pull ups to keep signal high when no data is being
    // sent
    i2c_init(i2c_default, SSD1306_I2C_CLK * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);

    // run through the complete initialization process
    SSD1306_init();

    // Initialize render area for entire frame (SSD1306_WIDTH pixels by SSD1306_NUM_PAGES pages)
    struct render_area frame_area = {
        start_col: 0,
        end_col : SSD1306_WIDTH - 1,
        start_page : 0,
        end_page : SSD1306_NUM_PAGES - 1
        };

    calc_render_area_buflen(&frame_area);

    // zero the entire display
    uint8_t buf[SSD1306_BUF_LEN];
    memset(buf, 0, SSD1306_BUF_LEN);
    render(buf, &frame_area);

    // intro sequence: flash the screen 3 times
    for (int i = 0; i < 3; i++) {
        SSD1306_send_cmd(SSD1306_SET_ALL_ON);    // Set all pixels on
        sleep_ms(500);
        SSD1306_send_cmd(SSD1306_SET_ENTIRE_ON); // go back to following RAM for pixel state
        sleep_ms(500);
    }

    // render 3 cute little raspberries
    struct render_area area = {
        start_page : 0,
        end_page : (IMG_HEIGHT / SSD1306_PAGE_HEIGHT)  - 1
    };

restart:

    area.start_col = 0;
    area.end_col = IMG_WIDTH - 1;

    calc_render_area_buflen(&area);

    uint8_t offset = 5 + IMG_WIDTH; // 5px padding

    for (int i = 0; i < 3; i++) {
        render(raspberry26x32, &area);
        area.start_col += offset;
        area.end_col += offset;
    }
```

```
    SSD1306_scroll(true);
    sleep_ms(5000);
    SSD1306_scroll(false);

    char *text[] = {
        "A long time ago",
        "  on an OLED ",
        "   display",
        " far far away",
        "Lived a small",
        "red raspberry",
        "by the name of",
        "    PICO"
    };

    int y = 0;
    for (int i = 0 ;i < count_of(text); i++) {
        WriteString(buf, 5, y, text[i]);
        y+=8;
    }
    render(buf, &frame_area);

    // Test the display invert function
    sleep_ms(3000);
    SSD1306_send_cmd(SSD1306_SET_INV_DISP);
    sleep_ms(3000);
    SSD1306_send_cmd(SSD1306_SET_NORM_DISP);

    bool pix = true;
    for (int i = 0; i < 2;i++) {
        for (int x = 0;x < SSD1306_WIDTH;x++) {
            DrawLine(buf, x, 0,  SSD1306_WIDTH - 1 - x, SSD1306_HEIGHT - 1, pix);
            render(buf, &frame_area);
        }

        for (int y = SSD1306_HEIGHT-1; y >= 0 ;y--) {
            DrawLine(buf, 0, y, SSD1306_WIDTH - 1, SSD1306_HEIGHT - 1 - y, pix);
            render(buf, &frame_area);
        }
        pix = false;
    }

    goto restart;

#endif
    return 0;
}
```