Gaming System Software Guide

Author: Gabriel Rodgers
Sponsoring Club: Embedded Systems Design Club
Date: 6/5/2025

Miscellaneous Information

- If you want to add more header files (i.e., like `#include <hardware/dma.h>`), then you need to add the file name (hardware_dma for the previous example) to the target_link_libraries variable in the CMakeLists.txt file.
- The added code for this project is additional to the Pico-GB project and uses mostly Pico SDK function calls to complete rotary encoder/ADC communications.
- Capstone Members: Gabriel Rodgers, Justin Gonzalez, Jaedo Oh, Mitchell Hopkins, Atom Rousseau
- This Capstone project was completed for Winter/Spring terms of 2024-2025 for the Interdisciplinary Capstone Courses, ENGR415 & ENGR416.
- Capstone project ID: ENGR.512

Volume Control Implementation

- Used 2 GPIO pins (10 and 11) to connect to the rotary encoder's DT (GP10) and CLK (GP11) pins.
- Used an interrupt on GP10 for rising and falling edges to detect a rotation.
- I also use GPIO pin 16 to connect to the rotary encoder's switch pin (for muting), though I did not yet implement mute functionality. A user can mute the volume by scrolling counterclockwise for at least 15 dentents.

Rotary encoder variables:

```
    /* Volatile globals */
    volatile uint8_t g_volume = 0;
    volatile uint8_t g_mute = 0;

    typedef enum{
        VOLUME_INCREASE,
        VOLUME_DECREASE,
        VOLUME_STABLE
    }volume_state_t;
```

- The volume_state_t typedef enumeration is used to determine what state the volume is in (should it be increased, decreased, or remain the same?). The g_volume global variable is set to one of the volume_state_t values, depending on what the volume should do.

Interrupt logic:

```
void gpio_vol_scroll(uint gpio, uint32_t event_mask) {
    uint8_t clk = gpio_get(GPIO_VOL_CLK);
    uint8_t dt = gpio_get(GPIO_VOL_DT);

    if (dt == clk) { //cw
        g_volume = VOLUME_INCREASE;
    }
    else {   //ccw
        g_volume = VOLUME_DECREASE;
    }
}
```

- Once the interrupt is triggered, it first reads the value of the CLK and DT pins.
- then it compares the two: if they are the same, then a clockwise rotation has occurred, and a global volume variable is set to a value called VOLUME_INCREASE. Otherwise, a counterclockwise rotation has occurred and a global volume variable is set to a value called VOLUME_DECREASE.

Initializing the rotary encoder pins:

```
gpio_set_function(GPIO_VOL_DT, GPIO_FUNC_SIO);
    gpio_set_function(GPIO_VOL_CLK, GPIO_FUNC_SIO);
    gpio_set_function(GPIO_VOL_MUTE, GPIO_FUNC_SIO);
    gpio_set_dir(GPIO_VOL_DT, false);
    gpio_set_dir(GPIO_VOL_CLK, false);
    gpio_set_dir(GPIO_VOL_MUTE, false);

    /* Set up interrupts for Rotary Encoder. */
    gpio_set_irq_enabled_with_callback(GPIO_VOL_DT, GPIO_IRQ_EDGE_RISE |
GPIO_IRQ_EDGE_FALL, true, gpio_vol_scroll);
```

- Mostly self explanatory; the gpio_set_irq_enabled_with_callback function call at the end basically creates an interrupt called gpio_vol_scroll, makes it trigger on the GPIO_VOL_DT pin (GP10) for both falling and rising edges.

Main loop logic:

```
  if(g_volume == VOLUME_INCREASE) {
            i2s_increase_volume(&i2s_config);
            g_volume = VOLUME_STABLE;
        }
        else if(g_volume == VOLUME_DECREASE) {
            i2s_decrease_volume(&i2s_config);
            g_volume = VOLUME_STABLE;
        }
```

- Compares the global volume variable affected by the interrupt to values to call volume change functions (either to increase or decrease the volume). Then the global volume variable is set to VOLUME_STABLE to ensure only changes done by the interrupt change the volume of the system.

Analog Joystick Implementation

- The analog joystick is connected to an 8-bit precision, 8-channel ADS7830 analog to digital converter (ADC), which is connected to the Pico via I2C.
- The Analog Joystick's xout and yout pins are connected to the ADC's Channel 1 and Channel 0 pins, respectively.
- Used 2 GPIO pins (0 and 1) to connect to the ADC's SDA (GP0) and SCL (GP1) pins.
- Used polling and waiting to synchronize communication between the ADC and the Pico.

ADC I2C address/commands:

```
#define ADS7830_ADDRESS     0x48    // | 1 for read
    #define ADS7830_CMD_CH0     0x8C    // last nibble = internal ref =
on, A/D converter = on
    #define ADS7830_CMD_CH1     0xCC
```

- The ADS7830_ADDRESS macro defines the address that all I2C communications must use to talk to the ADC.
- The ADS7830_CMD_CH0 macro defines the command that is sent to the ADC to receive the y position of the analog joystick.
- The ADS7830_CMD_CH0 macro defines the command that is sent to the ADC to receive the x position of the analog joystick.

Analog joystick global variables:

```
/* joystick position globals */
    uint8_t g_default_x;
    uint8_t g_default_y;
    uint8_t g_x;
    uint8_t g_y;
```

- There are two sets of global variables: g_default_x/y, which contains the initial power-up position of the analog joystick, and g_x/y, which contains the current position of the analog joystick.
- The "position" in any x or y direction of the joystick is just a voltage represented as a value between 0 and 255.

Initializing the ADC pins and getting the analog joystick's initial position:

```
    gpio_set_function(GPIO_I2C0_SDA, GPIO_FUNC_I2C);
    gpio_set_function(GPIO_I2C0_SCL, GPIO_FUNC_I2C);
    gpio_pull_up(GPIO_I2C0_SDA);
    gpio_pull_up(GPIO_I2C0_SCL);
```

```
    /* Get the default x and y coordinates of the joystick*/
    const uint8_t write_ch0 = ADS7830_CMD_CH0;
    const uint8_t write_ch1 = ADS7830_CMD_CH1;

    i2c_init(i2c0, 400 * 1000);

    uint8_t retval;
    retval = i2c_write_blocking(i2c0, ADS7830_ADDRESS, &write_ch0, 1,
false);
    if (retval == 1) {
        retval = i2c_read_blocking(i2c0, ADS7830_ADDRESS, &g_default_y, 1,
false);
    }

    retval = i2c_write_blocking(i2c0, ADS7830_ADDRESS, &write_ch1, 1,
false);
    if (retval == 1) {
        retval = i2c_read_blocking(i2c0, ADS7830_ADDRESS, &g_default_x, 1,
false);
    }
```

- The first 4 lines set up the GPIO pins on the Pico for I2C communication.
- The next 2 lines create variables to send via I2C that contain commands to get the x and y coordinates of the analog joystick (by instructing the ADC to send over its Channel 1 and Channel 0 voltage values as an unsigned, 8-bit integer). Note the const keyword - this is used to ensure that the commands are not changed by any of the communications.
- The next line sets the I2C peripheral used to communicate with the ADC to run at 400KHz.
- A variable, retval is used to hold the status of each I2C write/read. If retval is 1 after an i2c_read/write_blocking call, then that means that the message was read/written without issue.
- The next lines complete 4 I2C transactions:
    - Write the command to the ADC to specify a read from Channel 0 of the ADC
    - Read the value of Channel 0 of the ADC (and assign the read value to the default global position for y)
    - Write the command to the ADC to specify a read from Channel 1 of the ADC
    - Read the value of Channel 1 of the ADC (and assign the read value to the default global position for x)
- By completing these 4 I2C transactions BEFORE the main while loop, the program is able to get an initial, resting position for the joystick to use as reference when determining future joystick movements.

Main loop logic:

```
        uint8_t b_up = gpio_get(GPIO_UP);
        uint8_t b_down = gpio_get(GPIO_DOWN);
        uint8_t b_left = gpio_get(GPIO_LEFT);
        uint8_t b_right = gpio_get(GPIO_RIGHT);

        retval = i2c_write_blocking(i2c0, ADS7830_ADDRESS, &write_ch0, 1,
false);
        if (retval == 1) {
            retval = i2c_read_blocking(i2c0, ADS7830_ADDRESS, &g_y, 1,
false);
        }

        retval = i2c_write_blocking(i2c0, ADS7830_ADDRESS, &write_ch1, 1,
false);
        if (retval == 1) {
            retval = i2c_read_blocking(i2c0, ADS7830_ADDRESS, &g_x, 1,
false);
        }

        //logic for joystick movement
        if (g_x < (g_default_x - 50)) { //left
            gb.direct.joypad_bits.left = 1;
            gb.direct.joypad_bits.right = 0;
        }
        else if (g_x > (g_default_x + 50)) { //right
            gb.direct.joypad_bits.left = 0;
            gb.direct.joypad_bits.right = 1;
        }
        else {
            gb.direct.joypad_bits.left = b_left;
            gb.direct.joypad_bits.right = b_right;
        }

        if (g_y > (g_default_y + 50)) { //down
            gb.direct.joypad_bits.down = 0;
            gb.direct.joypad_bits.up = 1;
        }
```

```
        else if (g_y < (g_default_y - 50)) { //up
            gb.direct.joypad_bits.down = 1;
            gb.direct.joypad_bits.up = 0;
        }
        else {
            gb.direct.joypad_bits.down = b_down;
            gb.direct.joypad_bits.up = b_up;
        }
```

- The first four lines assign the button inputs to intermediary variables. These are assigned to the actual state for the up/down/left/right inputs (which are shared between the analog joystick and the buttons; the analog joystick holds priority) depending on whether the analog joystick has been moved.
- The next section with retvals is used to get the current x and y coordinates of the analog joystick by reading the Channel 1 and Channel 0 voltages of the ADC. This takes 4 I2C transactions and is the exact same as the 4 transactions in the previous section, except that the read values are put into the current x/y position variables (g_x and g_y).
- The section after the comment, “//logic for joystick movement” is the logic that determines whether the joystick has been moved from its default position - and if so, how should the up/down/left/right values be changed?
    - There are two chains of if/else if/else statements to do this logic; the first is for the x position and left/right inputs, and the second is for the y position and up/down inputs.
    - Each if/else if statement compares the current value for a position (g_x or g_y) to its default position (g_default_x or g_default_y) with a decent margin of error (+/- 50) to determine if the current position of the joystick constitutes as a movement. The joystick is too sensitive and inaccurate to use by comparing the default position to the current position, which is why the program uses +/- 50.
    - If the joystick’s x or y position is determined to not be moved, then its respective game inputs (i.e., for x it would be left/right and for y it would be up/down) are set to the button inputs that were read in the first 4 lines.
- By doing all of this the program is able to prioritize analog joystick movements for gaming and also retain button input functionality.