```
/*
#############################################################################
#                                                                           #
# dds-sweeper.c                                                             #
#                                                                           #
# Copyright 2022                                                            #
#                                                                           #
# Serial communication code based on the PineBlaster and PrawnBlaster       #
#   https://github.com/labscript-suite/pineblaster                          #
#   Copyright 2013, Christopher Billington                                  #
#   https://github.com/labscript-suite/prawnblaster                         #
#   Copyright 2013, Philip Starkey                                          #
#                                                                           #
# This file is used to flash a Raspberry Pi Pico microcontroller            #
# prototyping board to create a DDS-Sweeper (see readme.txt                 #
# This file is licensed under the Simplified BSD License.                   #
# See the license.txt file for the full license.                            #
#                                                                           #
#############################################################################
*/

#include <stdio.h>
#include <string.h>

#include "ad9959.h"
#include "hardware/clocks.h"
#include "hardware/dma.h"
#include "hardware/flash.h"
#include "hardware/pio.h"
#include "hardware/spi.h"
#include "pico/multicore.h"
#include "pico/stdlib.h"
#include "trigger_timer.pio.h"

#define VERSION "0.1.1"

// Default Pins to use
#define PIN_MISO 12
#define PIN_MOSI 15
#define PIN_SCK 14
#define PIN_SYNC 10
#define PIN_CLOCK 21
#define PIN_UPDATE 22
#define P0 19
#define P1 18
#define P2 17
#define P3 16
#define TRIGGER 8

#define PIO_TRIG pio0
#define PIO_TIME pio1

// Mutex for status
static mutex_t status_mutex;
static mutex_t wait_mutex;
```

```c
#define FLASH_TARGET_OFFSET (256 * 1024)

// STATUS flag
#define STOPPED 0
#define RUNNING 1
#define ABORTING 2
int status = STOPPED;

// PIO VALUES IT IS LOOKING FOR
#define UPDATE 0

#define MAX_SIZE 249856
#define TIMERS 5000
#define TIMING_OFFSET (MAX_SIZE - TIMERS * 4)

// For responding OK to successful commands
#define OK() printf("ok\n")

// ============================================================================
// global variables
// ============================================================================
ad9959_config ad9959;
char readstring[256];
bool DEBUG = true;
bool timing = false;

uint triggers;

uint timer_dma;

uint INS_SIZE = 0;
uint8_t instructions[MAX_SIZE];

// ============================================================================
// Utility Functions
// ============================================================================

void init_pin(uint pin) {
    gpio_init(pin);
    gpio_set_dir(pin, GPIO_OUT);
    gpio_put(pin, 0);
}

void init_pio() {
    uint offset = pio_add_program(PIO_TRIG, &trigger_program);
    trigger_program_init(PIO_TRIG, 0, offset, TRIGGER, P3, PIN_UPDATE);
    offset = pio_add_program(PIO_TIME, &timer_program);
    timer_program_init(PIO_TIME, 0, offset, TRIGGER);
}

int get_status() {
    mutex_enter_blocking(&status_mutex);
    int temp = status;
    mutex_exit(&status_mutex);
    return temp;
```

```
}

void set_status(int new_status) {
    mutex_enter_blocking(&status_mutex);
    status = new_status;
    mutex_exit(&status_mutex);
}

void measure_freqs(void) {
    // From https://github.com/raspberrypi/pico-examples under BSD-3-Clause License
    uint f_pll_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_SYS_CLKSRC_PRIMARY);
    uint f_pll_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_USB_CLKSRC_PRIMARY);
    uint f_rosc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_ROSC_CLKSRC);
    uint f_clk_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_SYS);
    uint f_clk_peri = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_PERI);
    uint f_clk_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_USB);
    uint f_clk_adc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_ADC);
    uint f_clk_rtc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_RTC);

    printf("pll_sys = %dkHz\n", f_pll_sys);
    printf("pll_usb = %dkHz\n", f_pll_usb);
    printf("rosc = %dkHz\n", f_rosc);
    printf("clk_sys = %dkHz\n", f_clk_sys);
    printf("clk_peri = %dkHz\n", f_clk_peri);
    printf("clk_usb = %dkHz\n", f_clk_usb);
    printf("clk_adc = %dkHz\n", f_clk_adc);
    printf("clk_rtc = %dkHz\n", f_clk_rtc);
}

void readline() {
    int i = 0;
    char c;
    while (true) {
        c = getchar();
        if (c == '\n') {
            readstring[i] = '\0';
            return;
        } else {
            readstring[i++] = c;
        }
    }
}

void update() { pio_sm_put(PIO_TRIG, 0, UPDATE); }

void sync() {
    gpio_put(PIN_SYNC, 1);
    sleep_ms(1);
    gpio_put(PIN_SYNC, 0);
    sleep_ms(1);
}

void reset() {
    sync();
    ad9959.sweep_type = 1;
    ad9959.channels = 1;
```

```
    clear();
    update();
}

void wait(uint channel) {
    pio_sm_get_blocking(PIO_TRIG, 0);
    triggers++;
}

void abort_run() {
    if (get_status() == RUNNING) {
        set_status(ABORTING);

        // take control of trigger pin from PIO
        init_pin(TRIGGER);
        gpio_put(TRIGGER, 1);
        sleep_ms(1);
        gpio_put(TRIGGER, 0);

        // reinit PIO to give Trigger pin back
        init_pio();
    }
}

// ============================================================================
// Set Table Instructions
// ============================================================================

void set_time(uint32_t addr, uint32_t time) {
    uint32_t cycles = time;
    if (addr == 0) {
        cycles -= 18;
    } else {
        cycles -= 10;
    }
    *((uint32_t *)(instructions + TIMING_OFFSET + 4 * addr)) = cycles;
}

bool set_ins(uint type, uint channel, uint addr, double s0, double e0, double delta, uint rate) {
    uint8_t ins[30];

    // for each step of buffered execution there is 1 byte of profile pin
    // instructions followed by the actual instruction for each channel

    // add one at the end here for this steps profile pin byte
    uint offset = (INS_SIZE * ad9959.channels + 1) * addr + 1;

    // offset from the beginning of this step to where this channel's instruction goes
    uint channel_offset = INS_SIZE * channel;

    // check there is enough space for this instruction
    uint tspace = timing ? TIMERS * 4 : 0;
    if (offset + channel_offset + INS_SIZE + tspace >= MAX_SIZE || (timing && addr > TIMERS)) {
        printf("Invalid Address\n");
        return false;
```

```
}

// address flow control instructions
if (channel == 4 || channel == 5) {
    instructions[offset - 1] = 0x00;
    if (channel == 5)
        // repeat instrcution
        instructions[offset] = 0xff;
    else
        // end instruction
        instructions[offset] = 0x00;
    return true;
}

// set csr
ins[0] = 0x00;
if (ad9959.channels == 1) {
    ins[1] = 0xf2;
} else {
    ins[1] = (1u << (channel + 4)) | 0x02;
}

if (type == 0) {
    // SINGLE STEP

    // Memory Map (12 bytes)
    // [ 0x00, CSR                      *Channel Select Register
    //   0x04, FTW3, FTW2, FTW1, FTW0,  *Frequcney Tuning Word
    //   0x05, POW1, POW0,              *Phase Offset Word
    //   0x06, ACR2, ACR1, ACR0         *Amplitude Control Register
    // ]

    ins[2] = 0x04;
    ins[7] = 0x05;
    ins[10] = 0x06;

    // profile pins do not matter for single tone mode, but the pio program
    // still expects a nonzero value for the profile pin mask
    instructions[offset - 1] |= 0x01;

    // calculate tuning values from real values
    uint8_t asf[3], ftw[4], pow[2];
    double freq, amp, phase;
    amp = get_asf(e0, asf);
    freq = get_ftw(&ad9959, s0, ftw);
    phase = get_pow(delta, pow);

    // write instruction
    memcpy(ins + 3, (uint8_t *)&ftw, 4);
    memcpy(ins + 8, (uint8_t *)&pow, 2);
    memcpy(ins + 11, (uint8_t *)&asf, 3);

    if (DEBUG) {
        printf(
            "Set ins #%d for channel %d with amp: %3lf %% freq: %3lf Hz phase: %3lf "
            "deg\n",
```

```
            addr, channel, amp, freq, phase);
    }

} else {
    // SWEEPS

    // Profile pins are the same for all sweep types
    // Profile pins getupdated twice for each trigger
    // the first update puts the profile pin high no matter what
    // then only if it is a downward sweep drop the profile pin low

    // The profile pin directions for a single table step are stored in a single byte
    // The least signifigant 4 bits in the byte corespond to the first value the
    // profile pin hits during an update. Since the pin should always go high first,
    // that means the least signifigant nibble should always be 0xf.
    if (s0 <= e0 && ad9959.channels == 1) {
        // case: upward sweep single channel mode
        instructions[offset - 1] = 0xff;
    } else if (s0 <= e0) {
        // case: upward sweep on this channel
        instructions[offset - 1] |= (1u << (3 - channel)) | (1u << (7 - channel));
    } else if (ad9959.channels == 1) {
        // case: downward sweep single channel mode
        instructions[offset - 1] = 0x0f;
    } else {
        // case: downward sweep on this channel
        instructions[offset - 1] &= ~(1u << (7 - channel));
        instructions[offset - 1] |= 1u << (3 - channel);
    }

    if (type == 1) {
        // AMP sweep
        // Memory Map
        // [ 0x00, CSR                                *Channel Select Register
        //   0x06, ACR2, ACR1, ACR0,                  *Amplitude Control Register
        //   0X07,  FRR,  RRR,                        *Linear Sweep Ramp Rate Register
        //   0x08, RDW3, RDW2, RDW1, RDW1,            *Rising Delta Word Register
        //   0x09, FDW3, FDW2, FDW1, FDW1,            *Falling Delta Word Register
        //   0x0a,  CW3,  CW2,  CW1,  CW0,            *Sweep Endpoint,
        //   0x03, CFR3, CFR2, CFR1, CFR0             *Channel Function Register
        // ]
        ins[2] = 0x06;
        ins[6] = 0x07;
        ins[9] = 0x08;
        ins[14] = 0x09;
        ins[19] = 0x0a;
        ins[24] = 0x03;

        // calculations: go from percentages to integers between 0 and 1024
        s0 = round(s0 * 1024);
        e0 = round(e0 * 1024);
        delta = round(delta * 1024);

        // ensure inside range
        if (delta < 1) delta = 1;
        if (s0 < 0) s0 = 0;
```

```
    if (e0 < 0) e0 = 0;

    if (delta > 1023) delta = 1023;
    if (s0 > 1023) s0 = 1023;
    if (e0 > 1023) e0 = 1023;

    // bit alignment
    uint32_t rate_word;
    rate_word = ((((uint32_t)delta) & 0x3fc) >> 2) | ((((uint32_t)delta) & 0x3) << 14);

    uint32_t lower, higher;
    if (s0 <= e0) {
        // UP SWEEP
        lower = (uint32_t)s0;
        higher = (uint32_t)e0;

        ins[7] = 0x01;
        ins[8] = rate;

        memcpy(ins + 10, (uint8_t *)&rate_word, 4);
        memcpy(ins + 15, "\xff\xc0\x00\x00", 4);
    } else {
        // DOWN SWEEP
        lower = (uint32_t)e0;
        higher = (uint32_t)s0;

        ins[7] = rate;
        ins[8] = 0x01;

        memcpy(ins + 10, "\xff\xc0\x00\x00", 4);
        memcpy(ins + 15, (uint8_t *)&rate_word, 4);
    }

    // bit alignments
    // the lower point needs to be in the bottom 10 bits of ACR
    lower = ((lower & 0xff) << 16) | (lower & 0xff00);
    memcpy(ins + 3, (uint8_t *)&lower, 3);
    // higher point goes in the top of CW1
    higher = ((higher & 0x3fc) >> 2) | ((higher & 0x3) << 14);
    memcpy(ins + 20, (uint8_t *)&higher, 4);

    // set FRC for Amplitude Sweep mode with sweep accumulator set to autoclear
    memcpy(ins + 25, "\x40\x43\x10", 3);

    if (DEBUG) {
        printf(
            "Set ins #%d for channel %d from %3lf%% to %3lf%% with delta %3lf%% "
            "and rate of %d\n",
            addr, channel, s0 / 10.23, e0 / 10.23, delta / 10.23, rate);
    }

} else if (type == 2) {
    // FREQ Sweep
    // Memory Map
    // [ 0x00, CSR                          *Channel Select Register
    //   0x04, FTW3, FTW2, FTW1, FTW0       *Frequency Tuning Word (Start point of sweep)
```

```
    //   0X07,  FRR, RRR,                *Linear Sweep Ramp Rate Register
    //   0x08, RDW3, RDW2, RDW1, RDW1,    *Rising Delta Word Register
    //   0x09, FDW3, FDW2, FDW1, FDW1,    *Falling Delta Word Register
    //   0x0a,  CW3,  CW2,  CW1,  CW0,    *Sweep Endpoint
    //   0x03, CFR3, CFR2, CFR1, CFR0     *Channel Function Register
    // ]
    ins[2] = 0x04;
    ins[7] = 0x07;
    ins[8] = rate;
    ins[9] = rate;
    ins[10] = 0x08;
    ins[15] = 0x09;
    ins[20] = 0x0a;
    ins[25] = 0x03;

    uint8_t s0_word[4], e0_word[4], rate_word[4];
    double start_point, end_point, rampe_rate;
    start_point = get_ftw(&ad9959, s0, s0_word);
    end_point = get_ftw(&ad9959, e0, e0_word);
    rampe_rate = get_ftw(&ad9959, delta, rate_word);

    // write instruction
    uint8_t *lower, *higher;
    if (s0 <= e0) {
        // SWEEP UP
        lower = s0_word;
        higher = e0_word;

        ins[8] = 0x01;
        memcpy(ins + 11, (uint8_t *)&rate_word, 4);
        memcpy(ins + 16, "\x00\x00\x00\x00", 4);
    } else {
        // SWEEP DOWN
        ins[9] = 0x01;
        lower = e0_word;
        higher = s0_word;

        memcpy(ins + 11, "\xff\xff\xff\xff", 4);
        memcpy(ins + 16, (uint8_t *)&rate_word, 4);
    }
    // set FRC for Freq Sweep mode with sweep accumulator set to autoclear
    memcpy(ins + 26, "\x80\x43\x10", 3);

    memcpy(ins + 3, lower, 4);
    memcpy(ins + 21, higher, 4);

    if (DEBUG) {
        printf(
            "Set ins #%d for channel %d from %4lf Hz to %4lf Hz with delta %4lf "
            "Hz and rate of %d\n",
            addr, channel, start_point, end_point, rampe_rate, rate);
    }

} else if (type == 3) {
    // PHASE Sweep
    // Memory Map
```

```
// [ 0x00, CSR                          *Channel Select Register
//   0x05, POW1, POW0                   *Phase Offset Word (Start point of sweep)
//   0X07,  FRR, RRR,                   *Linear Sweep Ramp Rate Register
//   0x08, RDW3, RDW2, RDW1, RDW1,      *Rising Delta Word Register
//   0x09, FDW3, FDW2, FDW1, FDW1,      *Falling Delta Word Register
//   0x0a,  CW3,  CW2,  CW1,  CW0,      *Sweep Endpoint
//   0x03, CFR3, CFR2, CFR1, CFR0       *Channel Function Register
// ]
ins[2] = 0x05;
ins[5] = 0x07;
ins[6] = rate;
ins[7] = rate;
ins[8] = 0x08;
ins[13] = 0x09;
ins[18] = 0x0a;
ins[23] = 0x03;

// convert from degrees to tuning words
s0 = round(s0 / 360.0 * 16384.0);
e0 = round(e0 / 360.0 * 16384.0);
delta = round(delta / 360.0 * 16384.0);

// validate params
if (delta > 16384 - 1) delta = 16384 - 1;
if (delta < 1) delta = 1;
if (s0 > 16384 - 1) s0 = 16384 - 1;
if (e0 > 16384 - 1) e0 = 16384 - 1;

// bit shifting to flip endianness
uint32_t rate_word;
rate_word = ((((uint32_t)delta) & 0x3fc0) >> 6) | ((((uint32_t)delta) & 0x3f) << 10);

uint32_t lower, higher;
if (s0 <= e0) {
    // sweep up
    ins[6] = 0x01;
    lower = (uint32_t)s0;
    higher = (uint32_t)e0;

    memcpy(ins + 9, (uint8_t *)&rate_word, 4);
    memcpy(ins + 14, "\x00\x00\x00\x00", 4);
} else {
    // sweep down
    ins[7] = 0x01;
    lower = (uint32_t)e0;
    higher = (uint32_t)s0;

    memcpy(ins + 9, "\xff\xff\xff\xff", 4);
    memcpy(ins + 14, (uint8_t *)&rate_word, 4);
}

lower = ((lower & 0xff) << 8) | ((lower & 0xff00) >> 8);
higher = ((higher & 0x3fc0) >> 6) | ((higher & 0x3f) << 10);

memcpy(ins + 3, (uint8_t *)&lower, 2);
memcpy(ins + 19, (uint8_t *)&higher, 4);
```

```
            memcpy(ins + 24, "\xc0\x43\x10", 3);

            if (DEBUG) {
                printf(
                    "Set ins #%d for channel %d from %4lf deg to %4lf deg with delta "
                    "%4lf deg and rate of %d\n",
                    addr, channel, s0 / 16384.0 * 360, e0 / 16384.0 * 360, delta / 16384.0 * 360,
                    rate);
            }
        }
    }

    // write the instruction to main memory
    memcpy(instructions + offset + channel_offset, ins, INS_SIZE);
    return true;
}

// ============================================================================
// Table Running Loop
// ============================================================================

void background() {
    // let other core know ready
    multicore_fifo_push_blocking(0);

    int hwstart = 0;
    while (true) {
        // wait for a start command
        hwstart = multicore_fifo_pop_blocking();

        set_status(RUNNING);

        // pre-calculate spacing vars
        uint step = INS_SIZE * ad9959.channels + 1;
        uint offset = 0;

        // count instructions to run
        bool repeat = false;
        int num_ins = 0;
        int i = 0;
        while (true) {
            // If an instruction is empty that means to stop
            if (instructions[offset] == 0x00) {
                if (instructions[offset + 1]) {
                    repeat = true;
                }
                break;
            }
            offset = step * ++i;
        }

        num_ins = i;
        offset = i = 0;
        triggers = 0;

        // sync just to be sure
```

```
        sync();

        if (hwstart) {
            pio_sm_put(PIO_TIME, 0, 0);
        }

        while (status != ABORTING) {
            // check if last instruction
            if (i == num_ins) {
                if (repeat) {
                    i = offset = 0;
                } else {
                    break;
                }
            }

            // prime PIO
            pio_sm_put(PIO_TRIG, 0, instructions[offset]);

            // send new instruciton to AD9959
            spi_write_blocking(spi1, instructions + offset + 1, step - 1);

            // if on the first instruction, begin the timer
            if (i == 0 && timing) {
                dma_channel_transfer_from_buffer_now(timer_dma, instructions + TIMING_OFFSET,
                                                     num_ins);
            }

            wait(0);

            offset = step * ++i;
        }

        // clean up
        dma_channel_abort(timer_dma);
        pio_sm_clear_fifos(PIO_TRIG, 0);
        pio_sm_clear_fifos(PIO_TIME, 0);
        set_status(STOPPED);
    }
}

// ============================================================================
// Serial Communication Loop
// ============================================================================

void loop() {
    readline();
    int local_status = get_status();

    if (strncmp(readstring, "version", 7) == 0) {
        printf("%s\n", VERSION);
    } else if (strncmp(readstring, "status", 6) == 0) {
        printf("%d\n", local_status);
    } else if (strncmp(readstring, "debug on", 8) == 0) {
        DEBUG = 1;
        OK();
```

```
    } else if (strncmp(readstring, "debug off", 9) == 0) {
        DEBUG = 0;
        OK();
    } else if (strncmp(readstring, "getfreqs", 8) == 0) {
        measure_freqs();
    } else if (strncmp(readstring, "numtriggers", 11) == 0) {
        printf("%u\n", triggers);
    } else if (strncmp(readstring, "reset", 5) == 0) {
        abort_run();
        reset();
        set_status(STOPPED);
        OK();
    } else if (strncmp(readstring, "abort", 5) == 0) {
        abort_run();
        OK();
    }
    // ====================================================
    // Stuff that cannot be done while the table is running
    // ====================================================
    else if (local_status != STOPPED) {
        printf(
            "Cannot execute command \"%s\" during buffered execution. Check "
            "status first and wait for it to return %d (stopped or aborted).\n",
            readstring, STOPPED);
    } else if (strncmp(readstring, "readregs", 8) == 0) {
        single_step_mode();
        update();
        read_all();
        OK();
    } else if (strncmp(readstring, "load", 4) == 0) {
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wstringop-overread"
        memcpy(instructions, ((uint8_t *)(XIP_BASE + FLASH_TARGET_OFFSET)), MAX_SIZE);
#pragma GCC diagnostic pop

        OK();
    } else if (strncmp(readstring, "save", 4) == 0) {
        uint32_t ints = save_and_disable_interrupts();
        // erase sections
        flash_range_erase(FLASH_TARGET_OFFSET, MAX_SIZE);
        // reprogram
        flash_range_program(FLASH_TARGET_OFFSET, instructions, MAX_SIZE);
        restore_interrupts(ints);
        OK();
    } else if (strncmp(readstring, "setchannels", 11) == 0) {
        uint channels;

        int parsed = sscanf(readstring, "%*s %u", &channels);

        if (parsed < 1) {
            printf("Missing Argument - expected: setchannels <num:int>\n");
        } else if (channels < 1 || channels > 4) {
            printf("Invalid Channels - expected: num must be in range 0-3\n");
        } else {
            ad9959.channels = channels;
            OK();
```

```
    }
} else if (strncmp(readstring, "setfreq", 7) == 0) {
    // setfreq <channel:int> <frequency:float>

    uint channel;
    double freq;
    int parsed = sscanf(readstring, "%*s %u %lf", &channel, &freq);
    if (parsed < 2) {
        printf(
            "Missing Argument - too few arguments - expected: setfreq "
            "<channel:int> <frequency:double>\n");
    } else if (channel < 0 || channel > 3) {
        printf("Invalid Channel - num must be in range 0-3\n");
    } else {
        uint8_t ftw[4];
        freq = get_ftw(&ad9959, freq, ftw);
        send_channel(0x04, channel, ftw, 4);
        update();

        if (DEBUG) {
            printf("set freq: %lf\n", freq);
        }

        OK();
    }
} else if (strncmp(readstring, "setphase", 8) == 0) {
    // setphase <channel:int> <phase:float>

    uint channel;
    double phase;
    int parsed = sscanf(readstring, "%*s %u %lf", &channel, &phase);
    if (parsed < 2) {
        printf(
            "Missing Argument - too few arguments - expected: setphase "
            "<channel:int> <frequency:double>\n");
    } else if (channel < 0 || channel > 3) {
        printf("Invalid Channel - channel must be in range 0-3\n");
    } else {
        uint8_t pow[2];
        phase = get_pow(phase, pow);
        send_channel(0x05, channel, pow, 2);
        update();

        if (DEBUG) {
            printf("Phase: %12lf\n", phase);
        }

        OK();
    }
} else if (strncmp(readstring, "setamp", 6) == 0) {
    // setamp <channel:int> <amp:float>

    uint channel;
    double amp;
    int parsed = sscanf(readstring, "%*s %u %lf", &channel, &amp);
    if (parsed < 2) {
```

```
            printf(
                "Missing Argument - expected: setamp <channel:int> "
                "<amp:double>\n");
        } else if (channel < 0 || channel > 3) {
            printf("Invalid Channel - channel must be in range 0-3\n");
        } else {
            uint8_t asf[3];
            amp = get_asf(amp, asf);
            send_channel(0x06, channel, asf, 3);
            update();

            if (DEBUG) {
                printf("Amp: %12lf\n", amp);
            }

            OK();
        }
    } else if (strncmp(readstring, "setmult", 7) == 0) {
        uint mult;

        int parsed = sscanf(readstring, "%*s %u", &mult);

        if (parsed < 1) {
            printf("Missing Argument - expected: setmult <pll_mult:int>\n");
        } else if (mult != 1 || !(mult >= 4 && mult <= 20)) {
            printf("Invalid Multiplier: multiplier must be 1 or in range 4-20\n");

        } else {
            // could do more validation to make sure it is a valid
            // multiply/system clock freq
            set_pll_mult(&ad9959, mult);
            update();

            OK();
        }
    } else if (strncmp(readstring, "setclock", 8) == 0) {
        uint src;   // 0 = internal, 1 = external
        uint freq;  // in Hz (up to 133 MHz)
        int parsed = sscanf(readstring, "%*s %u %u", &src, &freq);
        if (parsed < 2) {
            printf("Missing Argument - expected: setclock <mode:int> <freq:int>\n");
        } else {
            if (src > 1) {
                printf("Invalid Mode - mode must be in range 0-1\n");
            } else {
                // Set new clock frequency
                if (src == 0) {
                    if (set_sys_clock_khz(freq / 1000, false)) {
                        set_ref_clk(&ad9959, freq);
                        clock_configure(clk_peri, 0,
                                        CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS, 125 *
MHZ,
                                        125 * MHZ);
                        clock_gpio_init(PIN_CLOCK, CLOCKS_CLK_GPOUT0_CTRL_AUXSRC_VALUE_CLK_SYS,
1);
                        stdio_init_all();
                        OK();
```

```
                } else {
                    printf("Failure. Cannot exactly achieve that clock frequency.");
                }
            } else {
                set_ref_clk(&ad9959, freq);
                gpio_deinit(PIN_CLOCK);
                if (DEBUG) printf("AD9959 requires external reference clock\n");
                OK();
            }
        }
    }
} else if (strncmp(readstring, "mode", 4) == 0) {
    // mode <type:int> <timing:int>

    uint type, _timing;
    int parsed = sscanf(readstring, "%*s %u %u %u", &type, &_timing);

    if (parsed < 2) {
        printf("Missing Argument - expected: mode <type:int> <timing:int>\n");
    } else if (type > 3) {
        printf("Invalid Type - table type must be in range 0-3\n");
    } else {
        uint8_t sizes[] = {14, 28, 29, 27};
        INS_SIZE = sizes[type];
        ad9959.sweep_type = type;
        timing = _timing;

        if (ad9959.sweep_type == 0) {
            uint8_t cfr[] = {0x03, 0x00, 0x03, 0x04};
            // uint8_t ftw[] = {0x04, 0x00, 0x00, 0x00, 0x00};
            // uint8_t pow[] = {0x05, 0x00, 0x00};
            // uint8_t acr[] = {0x06, 0x00, 0x00, 0x00};

            uint8_t csr[] = {0x00, 0xf2};
            spi_write_blocking(spi1, csr, 2);
            spi_write_blocking(spi1, cfr, 4);
        }

        OK();
    }
} else if (strncmp(readstring, "set ", 4) == 0) {
    // set <channel:int> <addr:int> <start_point:double> <end_point:double>
    // <rate:double> (<time:int>)

    if (ad9959.sweep_type == 0) {
        // SINGLE TONE MODE
        uint32_t channel, addr, time;
        double freq, amp, phase;
        int parsed = sscanf(readstring, "%*s %u %u %lf %lf %lf %u", &channel, &addr, &freq,
                            &amp, &phase, &time);

        if (parsed > 1 && channel > 5) {
            printf(
                "Invalid Channel - expected 0-3 for channels or 4/5 for stop/repeat "
                "instrcution\n");
        } else if (channel > 3 && parsed < 2) {
```

```
            printf("Missing Argument - expected: set <channel:int> <addr:int> \n");
        } else if (!timing && parsed < 5 && channel < 4) {
            printf(
                "Missing Argument - expected: set <channel:int> <addr:int> <frequency:double>
"
                "<amplitude:double> <phase:double> (<time:int>)\n");
        } else if (timing && parsed < 6 && channel < 4) {
            printf(
                "No Time Given - expected: set <channel:int> <addr:int> "
                "<frequency:double> <amplitude:double> <phase:double> "
                "<time:int>\n");
        } else {
            bool succsess = set_ins(ad9959.sweep_type, channel, addr, freq, amp, phase, 0);
            if (succsess && timing) {
                set_time(addr, time);
            }
        }

        OK();
    } else if (ad9959.sweep_type <= 3) {
        // SWEEP MODE
        // set <channel:int> <addr:int> <start_point:double>
        // <end_point:double> <rate:double> <ramp-rate:int> (<time:int>)
        uint32_t channel, addr, ramp_rate, time;
        double start, end, rate;
        int parsed = sscanf(readstring, "%*s %u %u %lf %lf %lf %u %u", &channel, &addr,
&start,
                            &end, &rate, &ramp_rate, &time);

        if (parsed > 1 && channel > 5) {
            printf(
                "Invalid Channel - expected 0-3 for channels or 4/5 for stop/repeat "
                "instrcution\n");
        } else if (channel > 3 && parsed < 2) {
            printf("Missing Argument - expected: set <channel:int> <addr:int> \n");
        } else if (parsed < 6 && channel < 4) {
            printf(
                "Missing Argument - expected: set <channel:int> <addr:int> "
                "<start_point:double> <end_point:double> <delta:double> "
                "<rate:int> (<time:int>)\n");
        } else if (timing && parsed < 7 && channel < 4) {
            printf(
                "No Time Given - expected: set <channel:int> <addr:int> "
                "<start_point:double> <end_point:double> <delta:double> "
                "<rate:int> <time:int>\n");
        } else {
            bool succsess = set_ins(ad9959.sweep_type, channel, addr, start, end, rate,
ramp_rate);
            if (succsess && timing) {
                set_time(addr, time);
            }
        }

        OK();
    } else {
        printf(
            "Invalid Command - \'mode\' must be defined before "
```

```
                "instructions can be set\n");
        }
    } else if (strncmp(readstring, "start", 5) == 0) {
        if (ad9959.sweep_type == -1) {
            printf(
                "Invalid Command - \'mode\' must be defined before "
                "a table can be started\n");
        } else {
            pio_sm_clear_fifos(PIO_TRIG, 0);
            pio_sm_clear_fifos(PIO_TIME, 0);

            // start the other core
            multicore_fifo_push_blocking(0);
            OK();
        }
    } else if (strncmp(readstring, "hwstart", 7) == 0) {
        if (ad9959.sweep_type == -1) {
            printf(
                "Invalid Command - \'mode\' must be defined before "
                "a table can be started\n");
        } else {
            pio_sm_clear_fifos(PIO_TRIG, 0);
            pio_sm_clear_fifos(PIO_TIME, 0);

            // start the other core
            multicore_fifo_push_blocking(1);
            OK();
        }
    } else {
        printf("Unrecognized Command: \"%s\"\n", readstring);
    }
}

// ============================================================================
// Initial Setup
// ============================================================================

int main() {
    init_pin(PICO_DEFAULT_LED_PIN);
    gpio_put(PICO_DEFAULT_LED_PIN, 1);

    stdio_init_all();

    set_sys_clock_khz(125 * MHZ / 1000, false);

    // output sys clock on a gpio pin to be used as REF_CLK for AD9959
    clock_gpio_init(PIN_CLOCK, CLOCKS_CLK_GPOUT0_CTRL_AUXSRC_VALUE_CLK_SYS, 1);

    // attatch spi to system clock so it runs at max rate
    clock_configure(clk_peri, 0, CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS, 125 * MHZ,
                    125 * MHZ);

    // init SPI
    spi_init(spi1, 100 * MHZ);
    spi_set_format(spi1, 8, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
```

```
    gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);

    // launch other core
    multicore_launch_core1(background);
    multicore_fifo_pop_blocking();

    // initialise the status mutex
    mutex_init(&status_mutex);
    mutex_init(&wait_mutex);

    // init the PIO
    init_pio();

    // setup dma
    timer_dma = dma_claim_unused_channel(true);

    // if pico is timing itself, it will use dma to send all the wait
    // lengths to the timer pio program
    dma_channel_config c = dma_channel_get_default_config(timer_dma);
    channel_config_set_dreq(&c, DREQ_PIO1_TX0);
    channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
    dma_channel_configure(timer_dma, &c, &PIO_TIME->txf[0], instructions + TIMING_OFFSET, 0,
false);

    // put AD9959 in default state
    init_pin(PIN_SYNC);
    set_ref_clk(&ad9959, 125 * MHZ);
    set_pll_mult(&ad9959, 4);
    reset();

    while (true) {
        loop();
    }
    return 0;
}
```