```c
/**
 * \file
 * Signal acquisition-related functions
 */

// ASF includes
#include <adc.h>
#include <pmc.h>
#include <sysclk.h>

#include <inttypes.h>
#include "conf_board.h"

#include "acquisition.h"

static volatile uint8_t RECEIVED = 0;
static volatile uint32_t ADCVAL = 0;

void ADC_Handler(void)
{
    if (adc_get_status(ADC) & ADC_ISR_DRDY) {
        uint32_t result = adc_get_latest_value(ADC);
        ADCVAL = result;
        RECEIVED = 1;
    }
}

void adc_setup(void)
{
    pmc_enable_periph_clk(ID_ADC);
    adc_init(ADC, sysclk_get_main_hz(), ADC_CLOCK, 8);
    adc_configure_timing(ADC, 0, ADC_SETTLING_TIME_3, 1);
    adc_set_resolution(ADC, ADC_MR_LOWRES_BITS_12);
    adc_enable_channel(ADC, ADC_CHANNEL_3); // TODO: wrong channel
    adc_enable_interrupt(ADC, ADC_IER_DRDY);
    adc_configure_trigger(ADC, ADC_TRIG_SW, 0);
    NVIC_EnableIRQ(ADC_IRQn);
    adc_start(ADC);
}


double acq_get_values (unsigned count)
{
    double total = 0.0;
    size_t n;
    for (n = 0; n < count; ++n) {
        RECEIVED = 0;
        adc_start(ADC);
        while (!RECEIVED);
        total += ADCVAL;
    }
    total /= count;
    return total;
}
```

```c
/**
 * \file
 * Main loop and initializers.
 */

// libc includes
#include <assert.h>
#include <ctype.h>
#include <inttypes.h>
#include <stdio.h>
#include <string.h>

// Atmel ASF includes
#include <pio.h>
#include <spi.h>
#include <spi_master.h>
#include <stdio_usb.h>
#include <sysclk.h>
#include "conf_board.h"
#include "usb-functions.h"

// Software libraries
#include "scpi/scpi.h"
#include "scpi-def.h"
#include "util.h"

// Hardware support
#include "acquisition.h"
#include "synth.h"

static void pins_init(void);

/**
 * Initialize the board
 */
// Cannot declare static, as board.h declares this extern
void board_init(void)
{
    // Disable watchdog timer
    WDT->WDT_MR = WDT_MR_WDDIS;

        sysclk_enable_peripheral_clock(ID_PIOA);
        sysclk_enable_peripheral_clock(ID_PIOB);
        sysclk_enable_peripheral_clock(ID_PIOC);

    pins_init();
    pmc_enable_periph_clk(ID_PIOB);
}

/**
 * Initialize the SPI controller.
 */
static void spi_init(void)
{
    /* Configure an SPI peripheral. */
    spi_enable_clock(SPI_MASTER_BASE);
    spi_disable(SPI_MASTER_BASE);
    spi_reset(SPI_MASTER_BASE);
    spi_set_lastxfer(SPI_MASTER_BASE);
    spi_set_master_mode(SPI_MASTER_BASE);
    spi_disable_mode_fault_detect(SPI_MASTER_BASE);
    spi_set_peripheral_chip_select_value(SPI_MASTER_BASE, SPI_CHIP_PCS);
    spi_set_clock_polarity(SPI_MASTER_BASE, SPI_CHIP_SEL, SPI_CLK_POLARITY);
    spi_set_clock_phase(SPI_MASTER_BASE, SPI_CHIP_SEL, SPI_CLK_PHASE);
    spi_set_bits_per_transfer(SPI_MASTER_BASE, SPI_CHIP_SEL,
            SPI_CSR_BITS_8_BIT);
    spi_set_baudrate_div(SPI_MASTER_BASE, SPI_CHIP_SEL,
            (sysclk_get_cpu_hz() / 100000uL));
    spi_set_transfer_delay(SPI_MASTER_BASE, SPI_CHIP_SEL, SPI_DLYBS,
            SPI_DLYBCT);
```

```
    spi_enable(SPI_MASTER_BASE);
}

/**
 * Configure all device pins
 */
static void pins_init(void)
{
    size_t i;
    for (i = 0; PIN_TABLE[i].description; ++i) {
        if (!PIN_TABLE[i].index_str) continue; // Pin group, not pin
        pio_configure_pin(PIN_TABLE[i].index, PIN_TABLE[i].flags);
    }
}

/**
 * Main function.
 *
 * \return Unused (ANSI-C compatibility).
 */
int main(void)
{
    // Initialize all used peripherals.
    sysclk_init();
    irq_initialize_vectors();
    cpu_irq_enable();
    board_init();
    spi_init();
    adc_setup();
    stdio_usb_init();
    pio_set_pin_high(GPIO_LED1);

    SCPI_Init(&G_SCPI_CONTEXT);

    const size_t SMBUFFER_SIZE = 10;
    char smbuffer[10];
    for (;;) {
        // Get into smbuffer until either full, or a \r or \n
        size_t i;
        i = 0;
        while (i < SMBUFFER_SIZE - 1) {
            char ch = 0;
            if (G_CDC_ENABLED) {
                ch = udi_cdc_getc();
            } else {
                continue;
            }
            if (!ch) {
                continue;
            }
            if (ch == '\r' || ch == '\n') {
                smbuffer[i] = ch;
                ++i;
                break;
            } else {
                smbuffer[i] = ch;
                ++i;
            }
        }
        smbuffer[i] = 0; // Terminate!
        SCPI_Input(&G_SCPI_CONTEXT, smbuffer, strlen (smbuffer));
    }

    return 0;
}
```

```
// Atmel ASF includes
#include <pmc.h>

#include <stdio.h>
#include <inttypes.h>
#include "scpi/scpi.h"
#include "scpi-def.h"

/* These functions are required by the SCPI library to interact with the
 * console.
 */

/**
 * Write back to the SCPI console.
 * \param context   Active SCPI context
 * \param data      Data to write
 * \param len       Length of data
 * \return  Number of bytes written
 */
size_t SCPI_Write(scpi_t *context, const char * data, size_t len) {
    (void) context;
    return fwrite (data, 1, len, stdout);
}

/**
 * Flush the SCPI console. May be a no-op if not possible or sensible.
 * \param context   Active SCPI context
 * \return  Success or failure; returns success on no-op.
 */
scpi_result_t SCPI_Flush (scpi_t *context) {
    (void) context;
    return SCPI_RES_OK;
}

/**
 * Report a SCPI error. Not all SCPI interfaces report this; some require
 * polling. This may be a no-op in that case.
 * \param context   Active SCPI context
 * \param err       SCPI error number
 * \return  zero
 */
int SCPI_Error (scpi_t *context, int_fast16_t err) {
    (void) context;
    fprintf(stderr, "**ERROR: %" PRIdFAST16 ", \"%s\"\r\n", err, SCPI_ErrorTranslate(err)
);
    return 0;
}

/**
 * Handle the SCPI control commands.
 * \param context   Active SCPI context
 * \param ctrl      The control command given. See scpi_ctrl_name_t in
 *                  scpi/types.h for a list.
 * \param val       The value passed with the command.
 * \return  Success or failure
 */
scpi_result_t SCPI_Control(scpi_t * context, scpi_ctrl_name_t ctrl, scpi_reg_val_t val) {
    (void) context;
    if (SCPI_CTRL_SRQ == ctrl) {
        fprintf(stderr, "**SRQ: 0x%X (%d)\r\n", val, val);
    } else {
        fprintf(stderr, "**CTRL %02x: 0x%X (%d)\r\n", ctrl, val, val);
    }
    return SCPI_RES_OK;
}

/**
 * Handle the SCPI *TST? command.
 * \param context   Active SCPI context
 * \return  Success or failure
```

```
 */
scpi_result_t SCPI_Test(scpi_t * context) {
    (void) context;
    fprintf(stderr, "**Test\r\n");
    return SCPI_RES_OK;
}

/**
 * Handle the SCPI *RST command.
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t SCPI_Reset(scpi_t * context) {
    (void) context;

    // Here, we are going to reset the entire chip. This requires writing to
    // the RSTC (reset controller) control registers.

    // Zero out the control register first, so we don't reset prematurely
    RSTC->RSTC_CR = 0;

    // Set the mode register
    RSTC->RSTC_MR = 0
        | RSTC_MR_KEY_PASSWD    // Chip won't reset without the password
        ;

    // Set the control register
    RSTC->RSTC_CR = 0
        | RSTC_CR_KEY_PASSWD    // Chip won't reset without the password
        | RSTC_CR_PROCRST       // Reset processor
        ;

    // This return won't happen, of course :D
    return SCPI_RES_OK;
}
```

```
/**
 * \file
 *
 * \brief SCPI command declarations
 */

/*-
 * Modified heavily in 2014 by wcp52.
 * Copyright (c) 2012-2013 Jan Breuer,
 *
 * All Rights Reserved
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHORS ''AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
 * IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */


#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "scpi/scpi.h"
#include "scpi-def.h"
#include "scpi-test.h"
#include "scpi-lowlevel.h"

static const scpi_command_t scpi_commands[] = {
    /* IEEE Mandated Commands (SCPI std V1999.0 4.1.1) */
    { .pattern = "*CLS", .callback = SCPI_CoreCls,},
    { .pattern = "*ESE", .callback = SCPI_CoreEse,},
    { .pattern = "*ESE?", .callback = SCPI_CoreEseQ,},
    { .pattern = "*ESR?", .callback = SCPI_CoreEsrQ,},
    { .pattern = "*IDN?", .callback = SCPI_CoreIdnQ,},
    { .pattern = "*OPC", .callback = SCPI_CoreOpc,},
    { .pattern = "*OPC?", .callback = SCPI_CoreOpcQ,},
    { .pattern = "*RST", .callback = SCPI_CoreRst,},
    { .pattern = "*SRE", .callback = SCPI_CoreSre,},
    { .pattern = "*SRE?", .callback = SCPI_CoreSreQ,},
    { .pattern = "*STB?", .callback = SCPI_CoreStbQ,},
    { .pattern = "*TST?", .callback = SCPI_CoreTstQ,},
    { .pattern = "*WAI", .callback = SCPI_CoreWai,},

    /* Required SCPI commands (SCPI std V1999.0 4.2.1) */
    {.pattern = "SYSTem:ERRor[:NEXT]?", .callback = SCPI_SystemErrorNextQ,},
    {.pattern = "SYSTem:ERRor:COUNt?", .callback = SCPI_SystemErrorCountQ,},
    {.pattern = "SYSTem:VERSion?", .callback = SCPI_SystemVersionQ,},

    //{.pattern = "STATus:OPERation?", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:EVENt?", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:CONDition?", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:ENABle", .callback = scpi_stub_callback,},
    //{.pattern = "STATus:OPERation:ENABle?", .callback = scpi_stub_callback,},
```

```
    {.pattern = "STATus:QUEStionable[:EVENt]?", .callback = SCPI_StatusQuestionableEventQ
,},
    //{.pattern = "STATus:QUEStionable:CONDition?", .callback = scpi_stub_callback,},
    {.pattern = "STATus:QUEStionable:ENABle", .callback = SCPI_StatusQuestionableEnable,}
,
    {.pattern = "STATus:QUEStionable:ENABle?", .callback = SCPI_StatusQuestionableEnableQ
,},

    {.pattern = "STATus:PRESet", .callback = SCPI_StatusPreset,},

    // Low-level
    {.pattern = "LOWlevel:SETpin", .callback = LOWLEVEL_PIN_ACTION,},
    {.pattern = "LOWlevel:CLRpin", .callback = LOWLEVEL_PIN_ACTION,},
    {.pattern = "LOWlevel:GETpin", .callback = LOWLEVEL_PIN_ACTION,},
    {.pattern = "LOWlevel:LISTpins", .callback = LOWLEVEL_LIST_PINS,},

    /* Test commands */
    {.pattern = "Test:SPI", .callback = TEST_SPI,},
    {.pattern = "Test:INIF", .callback = TEST_INIF,}, /* Init interface */
    {.pattern = "Test:INCK", .callback = TEST_INCK,}, /* Init clock */
    {.pattern = "Test:FREQ", .callback = TEST_FREQ,},
    {.pattern = "Test:PHASE", .callback = TEST_PHASE,},
    {.pattern = "Test:AMPlitude", .callback = TEST_AMPLITUDE,},
    {.pattern = "Test:SAMple", .callback = TEST_SAMPLE,},
    {.pattern = "Test:CHannel", .callback = TEST_CHANNEL,},
    SCPI_CMD_LIST_END
};

static scpi_interface_t scpi_interface = {
    .error = SCPI_Error,
    .write = SCPI_Write,
    .control = SCPI_Control,
    .flush = SCPI_Flush,
    .reset = SCPI_Reset,
    .test = SCPI_Test,
};

#define SCPI_INPUT_BUFFER_LENGTH 256
static char scpi_input_buffer[SCPI_INPUT_BUFFER_LENGTH];
static scpi_reg_val_t scpi_regs[SCPI_REG_COUNT];
scpi_t G_SCPI_CONTEXT = {
    .cmdlist = scpi_commands,
    .buffer = {
        .length = SCPI_INPUT_BUFFER_LENGTH,
        .data = scpi_input_buffer,
    },
    .interface = &scpi_interface,
    .registers = scpi_regs,
    .units = scpi_units_def,
    .special_numbers = scpi_special_numbers_def,
    .idn = {"WCP52", "GPA1", "1", "0"},
};
```

```c
/**
 * @file
 * SCPI LOWlevel:* commands
 */

// Atmel ASF includes
#include <pio.h>
#include <string.h>

#include "scpi/scpi.h"
#include "scpi-lowlevel.h"
#include "conf_board.h"
#include "util.h"

/**
 * SCPI pin action: set
 */
scpi_result_t pin_action_set (uint32_t pin, uint32_t flags, const char *name)
{
    if ((flags & PIO_TYPE_Msk) != PIO_OUTPUT_0 && (flags & PIO_TYPE_Msk) != PIO_OUTPUT_1)
 {
        printf ("Pin '%s' is not an output!\r\n", name);
        return SCPI_RES_ERR;
    }

    pio_set_pin_high(pin);
    return SCPI_RES_OK;
}

/**
 * SCPI pin action: clr
 */
scpi_result_t pin_action_clr (uint32_t pin, uint32_t flags, const char *name)
{
    if ((flags & PIO_TYPE_Msk) != PIO_OUTPUT_0 && (flags & PIO_TYPE_Msk) != PIO_OUTPUT_1)
 {
        printf ("Pin '%s' is not an output!\r\n", name);
        return SCPI_RES_ERR;
    }

    pio_set_pin_low(pin);
    return SCPI_RES_OK;
}

/**
 * SCPI pin action: get
 */
scpi_result_t pin_action_get (uint32_t pin, uint32_t flags, const char *name)
{
    (void) flags;
    (void) name;
    printf ("%d\r\n", pio_get_pin_value(pin) ? 1 : 0);
    return SCPI_RES_OK;
}

/**
 * SCPI: low-level pin actions
 * LOWlevel:SETpin NAME
 * LOWlevel:CLRpin NAME
 * LOWlevel:GETpin NAME
 *
 * @param context - active SCPI context
 * @return success or failure
 */
scpi_result_t LOWLEVEL_PIN_ACTION (scpi_t *context)
{
    char const *str;
    size_t len;
    scpi_result_t (*pin_action) (uint32_t, uint32_t, const char *);
```

```
    if (!SCPI_ParamString(context, &str, &len, true)) {
        return SCPI_RES_ERR;
    }

    if (SCPI_IsCmd(context, "LOWlevel:SETpin")) {
        pin_action = &pin_action_set;
    } else if (SCPI_IsCmd(context, "LOWLEVEL:CLRpin")) {
        pin_action = &pin_action_clr;
    } else {
        pin_action = &pin_action_get;
    }

#define buflen 32
    char buffer[buflen];
    if (len > buflen - 1) {
        puts ("String too long\r");
        return SCPI_RES_ERR;
    }

    memcpy(buffer, str, len);
    buffer[len] = 0;

    bool found_pin = false;
    size_t i;
    for (i = 0; PIN_TABLE[i].description; ++i) {
        if (!PIN_TABLE[i].index_str) continue; // Pin group, not pin
        if (!strcmp (PIN_TABLE[i].pin_name_str, buffer)) {
            pin_action(PIN_TABLE[i].index, PIN_TABLE[i].flags, buffer);
            found_pin = true;
            break;
        }
    }

    if (!found_pin) {
        printf ("Pin ’%s’ not found!\r\n", buffer);
        return SCPI_RES_ERR;
    } else {
        return SCPI_RES_OK;
    }
}

scpi_result_t LOWLEVEL_LIST_PINS (scpi_t *context)
{
    (void) context;

    size_t i;
    for (i = 0; PIN_TABLE[i].description; ++i) {
        if (PIN_TABLE[i].index_str) {
            // Pin
            printf ("%-20s %-30s %s\r\n",
                    PIN_TABLE[i].pin_name_str,
                    PIN_TABLE[i].flags_str,
                    PIN_TABLE[i].description);
        } else {
            // Pin group
            printf ("==== %s ====\r\n", PIN_TABLE[i].description);
        }
    }

    return SCPI_RES_OK;
}
```

```c
/**
 * \file
 * \brief SCPI Test:* commands
 */

// Atmel ASF includes
#include <pio.h>
#include <spi.h>

#include "scpi/scpi.h"
#include "scpi-test.h"
#include "conf_board.h"
#include "synth.h"
#include "acquisition.h"
#include "util.h"

/**
 * SCPI: Send arbitrary SPI data.
 * Test:SPI DATA
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_SPI(scpi_t *context)
{
    int32_t val;
    if (!SCPI_ParamInt(context, &val, true)) {
        return SCPI_RES_ERR;
    }
    uint8_t val_byte = (uint8_t) val;
    printf("Transmitting value %02x\r\n", val_byte);
    spi_write(SPI_MASTER_BASE, val_byte, 0, 0);
    return SCPI_RES_OK;
}

/**
 * SCPI: Initialize DDS interface
 * Test:INIF
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_INIF(scpi_t *context)
{
    (void) context;
    synth_initialize_interface();
    return SCPI_RES_OK;
}

/**
 * SCPI: Initialize DDS system clock
 * Test:INCK
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_INCK(scpi_t *context)
{
    (void) context;
    synth_initialize_clock();
    return SCPI_RES_OK;
}

/**
 * SCPI: Set DDS frequency
 * Test:FREQ channel, frequency
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
```

```
scpi_result_t TEST_FREQ(scpi_t *context)
{
    int32_t ch;
    unsigned ch_uns;
    scpi_number_t freq;

    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }

    if (!SCPI_ParamNumber(context, &freq, true)) {
        return SCPI_RES_ERR;
    }

    ch_uns = ch;

    printf("Setting frequency %u to %f\r\n", ch_uns, freq.value);
    synth_set_frequency(ch_uns, freq.value);
    return SCPI_RES_OK;
}

/**
 * SCPI: Set DDS phase
 * Test:PHASE channel, phase
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_PHASE(scpi_t *context)
{
  int32_t ch;
  unsigned ch_uns;
  scpi_number_t phase;


    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }

    if (!SCPI_ParamNumber(context, &phase, true)) {
        return SCPI_RES_ERR;
    }

    ch_uns = ch;
    printf("Setting phase %u to %f\r\n", ch_uns, phase.value);
    synth_set_phase(ch_uns, phase.value);
    return SCPI_RES_OK;

}

/**
 * SCPI: Set DDS amplitude
 * Test:AMPlitude channel, amp
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_AMPLITUDE(scpi_t *context)
{
  int32_t ch;
  unsigned ch_uns;
  scpi_number_t amplitude;


    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }

    if (!SCPI_ParamNumber(context, &amplitude, true)) {
        return SCPI_RES_ERR;
```

```
    }

    ch_uns = ch;
    printf("Setting amplitude %u to %f\r\n", ch_uns, amplitude.value);
    synth_set_amplitude(ch_uns, amplitude.value);
    return SCPI_RES_OK;

}

/**
 * SCPI: Sample the input.
 * Test:SAMple
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_SAMPLE(scpi_t *context)
{
    int32_t num_samples;
    if (!SCPI_ParamInt(context, &num_samples, true)) {
        return SCPI_RES_ERR;
    }

    printf ("%f\r\n", acq_get_values (num_samples));
    return SCPI_RES_OK;
}

/**
 * SCPI: Set active input channel.
 * TEST:CHannel
 *
 * \param context   Active SCPI context
 * \return  Success or failure
 */
scpi_result_t TEST_CHANNEL(scpi_t *context)
{
  int32_t ch;


    if (!SCPI_ParamInt(context, &ch, true)) {
        return SCPI_RES_ERR;
    }


    util_set_pin (GPIO_CHANSEL, ch);

    return SCPI_RES_OK;
}
```

```
/**
 * \file
 * DDS synthesizer control functions
 */

// Atmel ASF includes
#include <delay.h>
#include <pio.h>
#include <spi.h>

#include "conf_board.h"
#include <string.h>

#include "synth.h"

// Synthesizer registers
uint8_t G_FR1[FR1_LEN];
uint8_t G_FR2[FR2_LEN];
uint8_t G_CFR[CFR_LEN];
uint8_t G_CFTW0[CFTW_LEN];
uint8_t G_CFTW1[CFTW_LEN];

uint8_t G_CPOW0[CPOW_LEN];
uint8_t G_CPOW1[CPOW_LEN];


uint8_t G_CACR0[CACR_LEN];
uint8_t G_CACR1[CACR_LEN];


/**
 * Reset the DDS IO system. This aborts a current IO cycle and prepares for
 * the next one.
 */
static void syncio(void)
{
    pio_set_pin_high(GPIO_DDS_SYNCIO);
    pio_set_pin_low(GPIO_DDS_SYNCIO);
}

/**
 * Commit loaded data into the DDS registers.
 */
static void io_update(void)
{
        pio_set_pin_high(GPIO_DDS_IOUPDATE);
        pio_set_pin_low(GPIO_DDS_IOUPDATE);
}

/**
 * Wait for a SPI transaction to finish.
 */
static void spi_wait(void)
{
        while (!spi_is_tx_empty(SPI_MASTER_BASE));
}

/**
 *  Send a control register to the DDS device.
 *  \param addr     Register address
 *  \param data     Register data array
 *  \param data_length  Length of data array
 */
static void send_control_register(
        uint8_t addr, const uint8_t *data, size_t data_length)
{
    pio_set_pin_low(GPIO_DDS_nCS);
    syncio();
    spi_write(SPI_MASTER_BASE, addr, 0, 0);
    for (size_t i = 0; i < data_length; ++i) {
```

```
        spi_write(SPI_MASTER_BASE, data[i], 0, 0);
    }
    spi_wait();
    pio_set_pin_high(GPIO_DDS_nCS);
    io_update();
}

/**
 * Send a channel register to the DDS device.
 * @param addr  Register address
 * @param data  Register data array
 * @param data_len      Length of data array
 * @param channel_num   Channel number (0 or 1)
 */
static void send_channel_register(
        uint8_t addr, const uint8_t *data,
        size_t data_length, unsigned channel_num)
{
    if (channel_num > 1) return;

    pio_set_pin_low(GPIO_DDS_nCS);
    syncio();

    // First, set the proper 'channel enable' bit
    spi_write(SPI_MASTER_BASE, 0x00, 0, 0);
    spi_write(SPI_MASTER_BASE, channel_num ? 0x42 : 0x82, 0, 0);

    // Now, send the data
    spi_write(SPI_MASTER_BASE, addr, 0, 0);
    for (size_t i = 0; i < data_length; ++i) {
        spi_write(SPI_MASTER_BASE, data[i], 0, 0);
    }

    spi_wait();
    pio_set_pin_high(GPIO_DDS_nCS);
    io_update();
}

/**
 * Initialize the DDS interface.
 */
void synth_initialize_interface(void)
{
    // Ensure sane pin defaults
        pio_set_pin_low(GPIO_DDS_PWRDN);
        pio_set_pin_low(GPIO_DDS_nCS);
        pio_set_pin_low(GPIO_DDS_IOUPDATE);

    // Make sure to delay after PWRDN goes low.
    // No idea how long! I can't find it in the datasheet...
    delay_ms(50);

    // Perform a master reset
        pio_set_pin_high(GPIO_DDS_MRST);
        pio_set_pin_low(GPIO_DDS_MRST);

    // Configure standard SPI mode
    syncio();
        spi_write(SPI_MASTER_BASE, 0x00, 0, 0);
        spi_write(SPI_MASTER_BASE, 0x02, 0, 0);

    spi_wait();
    pio_set_pin_high(GPIO_DDS_nCS);
    io_update();
}

/**
 * Initialize the DDS system clock.
 */
void synth_initialize_clock(void)
```

```
{
    memset(G_FR1, 0, FR1_LEN);
    REGSET(G_FR1, FR1_VCOGAIN, 1); /* VCO gain set for high frequency */
    REGSET(G_FR1, FR1_PLLRATIO, 20); /* PLL: 25 MHz crystal * 20 = 500 MHz */

    send_control_register(FR1_ADDR, G_FR1, FR1_LEN);

    memset(G_FR2, 0, FR2_LEN);
    REGSET(G_FR2, FR2_ALL_AUTOCLEAR_PHASE, 1); /* Autoclear phase accumulator on IOup*/
    send_control_register(FR2_ADDR, G_FR2, FR2_LEN);
}

/**
 * Set the DDS frequency on a channel.
 * \param channel   Channel number, either 0 or 1
 * \param freq      Frequency in Hz.
 */
void synth_set_frequency(unsigned channel, double freq)
{
    if (channel > 1) return;

    // First, convert 'freq' to a frequency tuning word.
    // From AD9958 datasheet, page 18: Fout = (FTW)(Fsys) / 2^32
    // So FTW = 2^32 * Fout / Fsys

    double ftw = (4294967296. * freq) / SYSCLK_FREQ;
    uint32_t ftw32 = (uint32_t) ftw;

    // Prepare the FTW as an array of four bytes
    uint8_t *cftw = channel ? G_CFTW1 : G_CFTW0;
    cftw[0] = (ftw32 & 0xff000000uL) >> 24;
    cftw[1] = (ftw32 & 0x00ff0000uL) >> 16;
    cftw[2] = (ftw32 & 0x0000ff00uL) >> 8;
    cftw[3] = (ftw32 & 0x000000ffuL);

    // Transmit the bytes
    send_channel_register(CFTW_ADDR, cftw, CFTW_LEN, channel);
}


/**
 * Set the DDS phase on a channel.
 * \param channel   Channel number, either 0 or 1
 * \param freq      Phase in degrees.
 */
void synth_set_phase(unsigned channel, double phase)
{

  if (channel > 1) return;
  // convert "phase" to phase offset word
  // equation according to datasheet page 18
  double pow =  (16384. * phase) / 360 ;
  uint16_t pow14 = (uint16_t)pow & 0x03FFF;

   //prepare pow as array of 2 bytes
    uint8_t *cpow = channel ? G_CPOW1 : G_CPOW0;
    cpow[0] = (pow14 & 0x0000ff00uL) >> 8;
    cpow[1] = (pow14 & 0x000000ffuL);

    send_channel_register(CPOW_ADDR, cpow, CPOW_LEN, channel);

}

/**
 * Set the DDS amplitude on a channel.
 * \param channel   Channel number, either 0 or 1
 * \param freq      Amplitude
 */
void synth_set_amplitude(unsigned channel, double amplitude)
{
```

```
     if (channel > 1) return;
     double acr =  1023 * amplitude ;
     //
     uint16_t acr16 = (uint16_t)acr & 0x03FF;
     //setting the amplitude enable bit high
     acr16 |= 0x1000;

    uint8_t *cacr = channel ? G_CACR1 : G_CACR0;
    cacr[0] = 0; // Ramp rate
    cacr[1] = (acr16 & 0x0000ff00uL) >> 8;
    cacr[2] = (acr16 & 0x000000ffuL);

    send_channel_register(CACR_ADDR, cacr, CACR_LEN, channel);
}
```

```c
// Atmel ASF includes
#include <pio.h>
#include <stdio_usb.h>

#include <inttypes.h>
#include "udi_cdc.h"
#include "usb_protocol_cdc.h"
#include "conf_board.h"

volatile bool G_CDC_ENABLED = false;

void main_sof_action(void) { }

void main_resume_action(void) { }

void main_suspend_action(void) { }

bool callback_cdc_enable(uint8_t port)
{
    (void) port;
    stdio_usb_enable();
    pio_set_pin_high(GPIO_LED2);
    G_CDC_ENABLED = true;
    return true;
}

void callback_cdc_disable(uint8_t port)
{
    (void) port;
    G_CDC_ENABLED = false;
    pio_set_pin_low(GPIO_LED2);
    stdio_usb_disable();
}

void callback_cdc_set_coding_ext(uint8_t port, usb_cdc_line_coding_t *cfg)
{
    (void) port;
    (void) cfg;
}

void callback_cdc_set_dtr(uint8_t port, bool enable)
{
    (void) port;
    (void) enable;
}

void callback_cdc_rx_notify(uint8_t port)
{
    (void) port;
}
```

```
/**
 * \file
 * Miscellaneous utilities
 */

#include "util.h"
#include "conf_board.h"

const struct pin_info PIN_TABLE[] = {

#define XPINGROUP(_grp) {.description = _grp},
#define XPIN(_pin, _idx, _flags, _descr) \
    {   .pin_name_str = #_pin, \
        .index_str = #_idx, \
        .flags_str = #_flags, \
        .description = _descr, \
        .index = PIO_ ## _idx ## _IDX, \
        .flags = (_flags) },
PIN_LIST
#undef XPINGROUP
#undef XPIN
    {NULL} // Sentinel

};
```