

/*

Andrii Natochii , TSNUK 2015 , andrii.natochii@gmail.com

Pick the best Clock:

Timer Definition

TIMER_CLOCK1	MCK / 2	84Mhz/2 = 42.000 MHz
TIMER_CLOCK2	MCK / 8	84Mhz/8 = 10.500 MHz
TIMER_CLOCK3	MCK / 32	84Mhz/32 = 2.625 MHz
TIMER_CLOCK4	MCK /128	84Mhz/128 = 656.250 KHz

ISR/IRQ TC Channel Due pins

TC0	TC0 0	2, 13
TC1	TC0 1	60, 61
TC2	TC0 2	58
TC3	TC1 0	none
TC4	TC1 1	none
TC5	TC1 2	none
TC6	TC2 0	4, 5
TC7	TC2 1	3, 10
TC8	TC2 2	11, 12

The SAM3X8E CPU has 3 Timer Counters (TC) they are called TC0, TC1, TC2.

Every Timer Counter contains 3 Channels numbered 0, 1 and 2 (this give us a total of 9 Channels).

Every Channel has its own counters and interrupt handler that are independent from other Channels.

In other words each Channel can be considered as a separate "Timer", and is like having 9 separate timers.

To initialize a Channel you need the following parameters:

TC0/TC1/TC2 - The Timer Counter instance

0 / 1 / 2 - The Channel number inside Timer Counter

If you want to use interrupts you must enable the NVIC (Nested Vector Interrupt Controller) for that channel with:

Code:

```
NVIC_EnableIRQ(TCx_IRQn);
```

where TCx_IRQn is the ID of the interrupt to enable. These id are listed in the following table together with the ISR handler function name:

TC	Channel	NVIC "irq"	IRQ handler function	PMC id
TC0	0	TC0_IRQn	TC0_Handler	ID_TC0
TC0	1	TC1_IRQn	TC1_Handler	ID_TC1
TC0	2	TC2_IRQn	TC2_Handler	ID_TC2
TC1	0	TC3_IRQn	TC3_Handler	ID_TC3
TC1	1	TC4_IRQn	TC4_Handler	ID_TC4

TC1	2	TC5_IRQn	TC5_Handler	ID_TC5
TC2	0	TC6_IRQn	TC6_Handler	ID_TC6
TC2	1	TC7_IRQn	TC7_Handler	ID_TC7
TC2	2	TC8_IRQn	TC8_Handler	ID_TC8

Every peripheral in the SAM3X is off by default (to save power) and should be turned on. To turn on you need to run the following command:

Code:

```
pmc_enable_periph_clk(id);
```

where id is found on the last column of the above table (ID_TCx).

It happened that ID_TCx constant equals TCx_IRQn, so Sebastian Vik has simplified a bit the function using TCx_IRQn as input for pmc_enable_periph_clk:

Code:

```
pmc_enable_periph_clk((uint32_t)irq);
```

```
void NVIC_EnableIRQ(IRQn_t IRQn) // Enable IRQn
```

```
void NVIC_DisableIRQ(IRQn_t IRQn) // Disable IRQn
```

```
void NVIC_SetPriorityGrouping(uint32_t priority_grouping) // Задать
группы/подгруппы приоритетов
```

```
void NVIC_EnableIRQ(IRQn_t IRQn) // Включить IRQn
```

```
void NVIC_DisableIRQ(IRQn_t IRQn) // Выключить IRQn
```

```
uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn) // Вернуть true (точнее
IRQ-Number) если IRQn в ожидании
```

```
void NVIC_SetPendingIRQ (IRQn_t IRQn) // Поставить IRQn в ожидание
```

```
void NVIC_ClearPendingIRQ (IRQn_t IRQn) // Выкинуть из очереди на ожидание
IRQn
```

```
uint32_t NVIC_GetActive (IRQn_t IRQn) // Возвращает номер текущего
активного прерывания если такое имеется
```

```
void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority) // Задать приоритет IRQn
```

```
uint32_t NVIC_GetPriority (IRQn_t IRQn) // Считать приоритет IRQn
```

```
void NVIC_SystemReset (void) // Reset the system
```

The SPI Bus

The MCP4241 connects and communicates with a microcontroller (such as an Arduino or a Teensy) via the SPI Bus.

SPI is a protocol that allows microcontrollers to interface easily with a large number of external chips and sensors.

SPI is a host / slave type bus. A single host microcontroller can interface with one or more slave chips or sensors.

SPI uses up to four pins for communication:

--> CS (may also be called SS): chip select:

Used by the microcontroller to select each device. The host microcontroller has a CS output pin for every slave device that is to be used.

--> SCK: serial clock:

Used by the host microcontroller to time the data that is moved to and from the slave device.

There is one SCK connection that is shared amongst all SPI devices.

--> SDI (MOSI): slave data in (master out slave in):

Used by the host microcontroller to send data to a slave device.

There is one SDI connection that is shared amongst all SPI devices.

--> SDO (MISO): slave data out (master in slave out):

Used by the host microcontroller to receive data from a slave device.

There is one SDO connection that is shared amongst all SPI devices.

All SPI slave devices require the CS and SCK pins.

However, many device may not require both the SDI and SDO connections, as they might only either send or receive data.

With the MCP4241 chip, we want to set the resistance of a digital potentiometer using data from a host microntroller.

As a result, we only need CS, SCK and SDI (MOSI), because the information needs to travel only from the host to the slave device.

The layout of the 14-pin MCP42010 chip:

Pin 01	--> CS	Chip Select
Pin 02	--> SCK	Serial Clock
Pin 03	--> SI	Serial Data Input
Pin 04	--> VSS	Ground
Pin 05	--> PB1	Terminal B Connection For Pot 1
Pin 06	--> PW1	Wiper Connection For Pot 1
Pin 07	--> PA1	Terminal A Connection For Pot 1
Pin 08	--> PA0	Terminal A Connection For Pot 0
Pin 09	--> PW0	Wiper Connection For Pot 0
Pin 10	--> PB0	Terminal B Connection For Pot 0
Pin 11	--> RS	Reset Input
Pin 12	--> SHDN	Shutdown Input
Pin 13	--> SO	Data Out for Daisy-Chaining
Pin 14	--> VDD	Power

- Connect DUE_PIN_D2 --> CS (pin 1) of the MCP42010
- Connect DUE_PIN_D3 --> SCK (pin 2) of the MCP42010
- Connect DUE_PIN_D4 --> MOSI (pin 3) of the MCP42010
- Connect GRND --> Vss (pin 4) of the MCP42010.
- Connect NC --> PB1 (pin 5) of the MCP42010
- Connect NC --> PW1 (pin 6) of the MCP42010
- Connect NC --> PA1 (pin 7) of the MCP42010

- Connect IN --> PA0 (pin 8) of the MCP42010
- Connect OUT --> PW0 (pin 9) of the MCP42010
- Connect GRND --> PB0 (pin 10) of the MCP42010
- Connect 5V --> RS (pin 11) of the MCP42010
- Connect 5V --> SHDN (pin 12) of the MCP42010
- Connect NC --> MISO (pin 13) of the MCP42010
- Connect 5V --> Vdd (pin 14) of the MCP42010.

*/

```
#include <math.h>
```

```
#include <stdlib.h>
```

```
#include "DueFlashStorage.h"
```

```
DueFlashStorage dueFlashStorage;
```

```
#define FIRST_CLOCK_PERIOD 1000000 // usec
```

```
#define SECOND_CLOCK_PERIOD 2 // usec
```

```
#define MAX_POINTS 100000
```

```
#define MAX_POINT_PER_SIGNAL 1000
```

```
#define CHANNEL_NUMBER 2048
```

```
#define START_STOP_PIN 40
```

```
#define FLASH_MEM_PIN 30
```

```
#define LED_BOARD_PIN 13
```

```
#define PIN_SS1 2 // set slave select 1 pin
```

```
#define PIN_CLK 3 // set clock pin
```

```
#define PIN_MOUT 4 // set master out, slave in pin
```

```
int signal_func[MAX_POINT_PER_SIGNAL] = {};
```

```
uint32_t signal_iterator = 0, point_iterator = 1, n = 0;
```

```
boolean run_st = false;
```

```
uint32_t X_spec[CHANNEL_NUMBER] = {}, Y_spec[CHANNEL_NUMBER] = {}, pointNum = 0,
```

```
Y_spec_SUM = 0;
```

```
/******
```

```
/**/ double EnrgRage = 2000;
```

```
/**/ double EoverN = EnrgRage/CHANNEL_NUMBER;
```

```
/**/ double Energy = 1500;
```

```
/**/ double Intensity = 10000;
```

```
/**/ double FWHM = 50;
```

```
/**/ double rho = 0.001;
```

```
/**/ int SpectrumKind = 1;
```

```
/**/ int average_frequency = 10000;
```

```
/**/ byte cmd_byte0 = B00010001; // command byte to write to pot 0, from the
```

MCP42XXX datasheet

```
/**/ byte work = B00000000; // setup a working byte, used to bit shift the
```

data out

```
/******
```

```
void setup()
```

```

{
    pinMode(PIN_SS1, OUTPUT);    // set CS pin to output
    pinMode(PIN_CLK, OUTPUT);    // set SCK pin to output
    pinMode(PIN_MOUT, OUTPUT);    // set MOSI pin to output
    digitalWrite(PIN_SS1, HIGH); // hold slave select 1 pin high, so that chip is
not selected to begin with

    pinMode (LED_BOARD_PIN, OUTPUT);
    pinMode (FLASH_MEM_PIN, OUTPUT);
    pinMode (START_STOP_PIN, OUTPUT);

    // if analog input pin 0 is unconnected, random analog
    // noise will cause the call to randomSeed() to generate
    // different seed numbers each time the sketch runs.
    // randomSeed() will then shuffle the random function.
    randomSeed(analogRead(0));
    digitalWrite(LED_BOARD_PIN, LOW);
    digitalWrite(FLASH_MEM_PIN, LOW);
    digitalWrite(START_STOP_PIN, LOW);

    Serial.begin(115200);
    Serial.flush();
    /* turn on the timer clock in the power management controller */
    pmc_set_writeprotect(false);    // disable write protection for pmc registers
    pmc_enable_periph_clk(ID_TC6);    // enable peripheral clock TC6
    pmc_enable_periph_clk(ID_TC7);    // enable peripheral clock TC7

    TC_Configure(/* clock */TC2, /* channel */0, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC |
TC_CMR_TCCLKS_TIMER_CLOCK1);
    TC_Configure(/* clock */TC2, /* channel */1, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC |
TC_CMR_TCCLKS_TIMER_CLOCK1);
    TC_SetRC(TC2, 0, 42*FIRST_CLOCK_PERIOD); // 42*FIRST_CLOCK_PERIOD/42000000 =
clock time duration in seconds
    TC_SetRC(TC2, 1, 42*SECOND_CLOCK_PERIOD); // 42*SECOND_CLOCK_PERIOD/42000000 =
clock time duration in seconds

    TC_Start(TC2, 0);
    TC_Start(TC2, 1);

    // enable timer interrupts on the timer
    TC2->TC_CHANNEL[0].TC_IER=TC_IER_CPCS;    // IER = interrupt enable register
    TC2->TC_CHANNEL[0].TC_IDR=~TC_IER_CPCS;    // IDR = interrupt disable register

    TC2->TC_CHANNEL[1].TC_IER=TC_IER_CPCS;    // IER = interrupt enable register
    TC2->TC_CHANNEL[1].TC_IDR=~TC_IER_CPCS;    // IDR = interrupt disable register

    /* Enable the interrupt in the nested vector interrupt controller */
    /* TCn_IRQn where n is: [the timer number] * [timer channels{3}] + [the channel
number] = (n*3)+m for timer{n} channel{m} */

```

```

NVIC_EnableIRQ(TC6_IRQn);
NVIC_EnableIRQ(TC7_IRQn);

NVIC_SetPriority (TC6_IRQn, 1);
NVIC_SetPriority (TC7_IRQn, 2);

/* DAC Configuration */
analogWriteResolution(12);
analogWrite(DAC0,0); // Enables DAC0
dacc_set_channel_selection(DACC_INTERFACE, 0); //select DAC channel 0
dacc_write_conversion_data(DACC_INTERFACE, 0); //write on DAC
}

void loop()
{}

// INTERRUPT HANDLERS
void TC6_Handler()
{
    // We need to get the status to clear it and allow the interrupt to fire again
    TC_GetStatus(TC2, 0);
    String content = "";
    if (Serial.available())
    {
        while (Serial.available())
        {
            char inChar = (char)Serial.read();
            if(inChar == 'a')
            {
                EnrgRage = atof(content.c_str());
                break;
            }
            if(inChar == 'b')
            {
                Energy = atof(content.c_str());
                break;
            }
            if(inChar == 'c')
            {
                Intensity = atoi(content.c_str());
                break;
            }
            if(inChar == 'd')
            {
                FWHM = atof(content.c_str());
                break;
            }
            if(inChar == 'e')
            {

```

```

        SpectrumKind = atoi(content.c_str());
        break;
    }
    if(inChar == 'f')
    {
        average_frequency = atoi(content.c_str());
        average_frequency = constrain(average_frequency, 1, 1000000);
        break;
    }
    if(inChar == 'g')
    {
        int value = map(atoi(content.c_str()),1,10,0,255);
        value = constrain(value, 0, 255);
        spi_out(PIN_SS1, cmd_byte0, value);           // send out data to chip
1, pot 0
        break;
    }
    if(inChar == 'h')
    {
        rho = atof(content.c_str());
        break;
    }
    if(inChar == 'i')
    {
        digitalWrite(START_STOP_PIN,HIGH);
        createSpectrum();
        createSignals();
        run_st = true;
        break;
    }
    if(inChar == 'j')
    {
        digitalWrite(START_STOP_PIN,LOW);
        run_st = false;
        break;
    }
    content += inChar;
}
}
}

```

```

void TC7_Handler()
{
    if(run_st)
    {
        TC_GetStatus(TC2, 1);
        if(point_iterator > n)
        {
            int signal_time = (int)(-1000000.0*log((double)random(1,

```

```

100)/100)/average_frequency);
    n = 1 + signal_time/SECOND_CLOCK_PERIOD;
    n = constrain(n,1,MAX_POINT_PER_SIGNAL);
    point_iterator = 0;
    signal_iterator++;
    if(signal_iterator > pointNum)
    {
        signal_iterator = 0;
    }
    for(int i = 0; i <= n; i++)
    {
        uint32_t value = 0;
        bytes2uint(dueFlashStorage.readAddress(signal_iterator),value);
        signal_func[i] = value*exp(-i*SECOND_CLOCK_PERIOD*rho);
    }
}

dacc_write_conversion_data(DACC_INTERFACE, signal_func[point_iterator]);
point_iterator++;
}
else
{
    dacc_write_conversion_data(DACC_INTERFACE, 0);
}
}

void spi_transfer(byte working)    // function to actually bit shift the data
byte out
{
    for(int i = 1; i <= 8; i++)    // setup a loop of 8 iterations, one for each
bit
    {
        if (working > 127)        // test the most significant bit
        {
            digitalWrite (PIN_MOUT,HIGH);    // if it is a 1 (ie. B1XXXXXXXX), set the
master out pin high
        }
        else
        {
            digitalWrite (PIN_MOUT, LOW);    // if it is not 1 (ie. B0XXXXXXXX), set
the master out pin low
        }
        digitalWrite (PIN_CLK,HIGH);    // set clock high, the pot IC will read
the bit into its register
        working = working << 1;
        digitalWrite(PIN_CLK,LOW);    // set clock low, the pot IC will stop
reading and prepare for the next iteration (next significant bit
    }
}

```

```

void spi_out(int SS, byte cmd_byte, byte data_byte) // SPI tranfer out function
begins here
{
    digitalWrite (SS, LOW);      // set slave select low for a certain chip, defined
in the argument in the main loop. selects the chip
    work = cmd_byte;             // let the work byte equal the cmd_byte, defined in
the argument in the main loop
    spi_transfer(work);          // transfer the work byte, which is equal to the
cmd_byte, out using spi
    work = data_byte;            // let the work byte equal the data for the pot
    spi_transfer(work);          // transfer the work byte, which is equal to the
data for the pot
    digitalWrite(SS, HIGH);      // set slave select high for a certain chip,
defined in the argument in the main loop. deselcts the chip
}

```

```

void createSignals()
{
    pointNum = 0;
    if(Y_spec_SUM > MAX_POINTS)
    {
        for(int i = 0; i < CHANNEL_NUMBER; i++)
        {
            Y_spec[i] = Y_spec[i]*MAX_POINTS/Y_spec_SUM;
        }
    }
    int state = 1;
    digitalWrite(FLASH_MEM_PIN,state);
    for(int i = 0; i < CHANNEL_NUMBER; i++)
    {
        for(int j = 0; j < Y_spec[i]; j++)
        {
            byte bytes[3];
            uint2bytes(X_spec[i],bytes);
            dueFlashStorage.write(pointNum, bytes, sizeof(bytes));
            pointNum++;
        }
        if(i % 100 == 0)
        {
            state = !state;
            digitalWrite(FLASH_MEM_PIN,state);
        }
    }
    digitalWrite(FLASH_MEM_PIN,LOW);
}

```

```

void createSpectrum()
{

```

```

// Gaussian
if(SpectrumKind == 1)
{
    Y_spec_SUM = 0;
    double Sigma = FWHM/(EoverN*2*sqrt(2*log(2)));
    double Amplitude = (double)Intensity/IntegralGaus(Energy,EoverN,Sigma,0,
CHANNEL_NUMBER);
    for(int i = 0; i < CHANNEL_NUMBER; i++)
    {
        X_spec[i] = i;
        Y_spec[i] =
Amplitude*exp(-(X_spec[i]-(Energy)/EoverN)*(X_spec[i]-(Energy)/EoverN)/(2*Sigma*Si
gma));
        Y_spec_SUM += Y_spec[i];
    }
}
// Gaussian + Compton
else if(SpectrumKind == 2)
{
    Y_spec_SUM = 0;
    double Sigma = FWHM/(EoverN*2*sqrt(2*log(2)));
    double Amplitude = (double)Intensity/IntegralGaus(Energy,EoverN,Sigma,0,
CHANNEL_NUMBER);
    for(int i = 0; i < CHANNEL_NUMBER; i++)
    {
        X_spec[i] = i;
        Y_spec[i] =
Amplitude*exp(-(X_spec[i]-(Energy)/EoverN)*(X_spec[i]-(Energy)/EoverN)/(2*Sigma*Si
gma))+Compton((Energy)/EoverN,X_spec[i],EoverN,Amplitude);
        Y_spec_SUM += Y_spec[i];
    }
}

double Compton(double ENERGY_F, double X_F, double K_F, double AMPLITUDE_F)
{
    double mc2 = 0.511;
    double Eg = ENERGY_F*K_F/1000; // [MeV]
    double a = Eg/mc2;
    double Ee = X_F*K_F/1000; // [MeV]
    double C = 2*a/(1+2*a);
    double Y_F = 0;
    if(Ee < (Eg*C))
    {
        Y_F =
(3*AMPLITUDE_F/(8*a*Eg))*(2-(2*Ee/(a*(Eg-Ee)))+(Ee*Ee/(a*a*(Eg-Ee)*(Eg-Ee)))+(Ee*E
e/(Eg*(Eg-Ee))));
    }
    return Y_F;
}

```

```

}

double IntegralGaus(double ENERGY_F, double K_F, double SIGMA_F, int XMIN,int
XMAX)
{
    double sum = 0;
    int nn = 0;
    for(int xx = XMIN; xx < XMAX; xx++)
    {
        nn++;
        sum += exp(-(xx-ENERGY_F/K_F)*(xx-ENERGY_F/K_F)/(2*SIGMA_F*SIGMA_F));
    }
    return sum/nn;
}

void uint2bytes(uint32_t val, byte* bytes)
{
    bytes[0] = (val >> 0) & 0xFF;
    bytes[1] = (val >> 8) & 0xFF;
    bytes[2] = (val >> 16) & 0xFF;
}

void bytes2uint(byte* bytes, uint32_t &val)
{
    val = (bytes[2] << 16) | (bytes[1] << 8) | (bytes[0] << 0);
}

```