TECHNICAL NOTE

SPI COMMUNICATION WITH SCC2000 SERIES

1 Introduction

The objective of this document is to show how to set up SPI communication between Murata SCC2000 series Combined Gyroscope & 3-axis Accelerometer component and an NXP LPC11U14 Cortex-M0 microcontroller. The code example contains the following operations:

- LPC11U14 MCU configuration.
- SCC2xxx component start-up.
- Measurement data is read and sent to serial port. To achieve the best noise performance the data is read out at 2300 Hz frequency using a timer interrupt and the SPI clock frequency is 8 MHz.
- Gyroscope and Accelerometer run-time errors are handled.

2 Development Hardware

In this example a Murata prototype board SCC2230-D08 PWB was connected to an NXP LPC11U14 LPCXpresso board, please see Figure 1 below. Depending on cable lengths an external supply bypass capacitor may need to be added close to the PWB between power supply lines (C1 in Figure 1). The serial communication in the LPCXpresso board is TTL level thus to send data to a terminal emulator an FTDI TTL to USB serial converter was used (TTL-232R-3V3-WE).

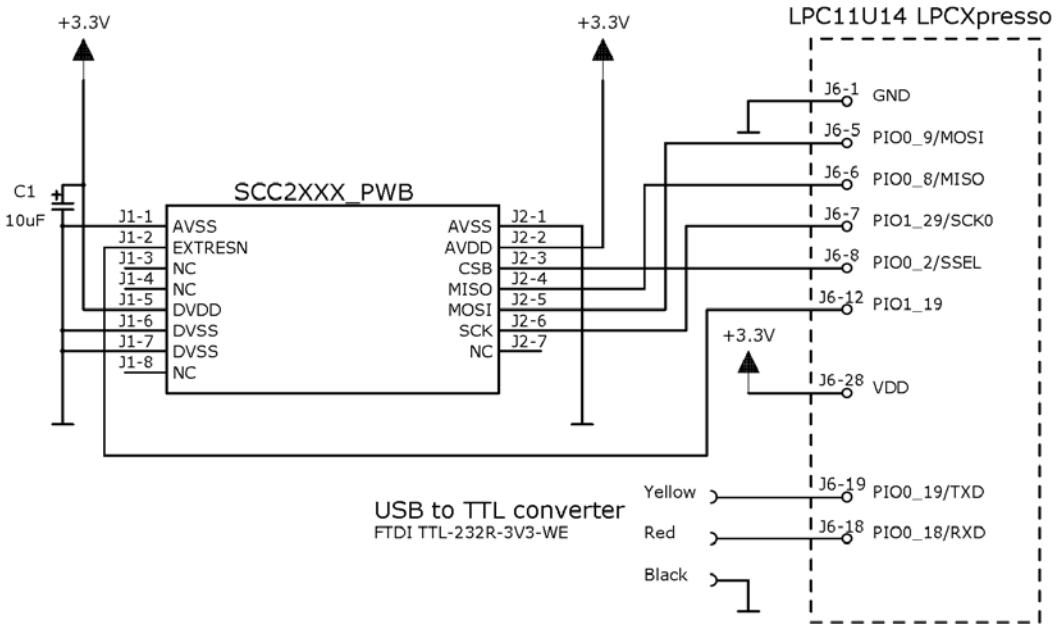

Figure 1. System schematic

3 C-Code Example

The example code was created for NXP LPC11U14 LPCXpresso Development Board using Keil uVision MDK-Lite Version 4.74 and Keil Ulink2 Debug Adapter.

C-language software example on the next pages shows how to implement basic communication with a SCC2000 series component using SPI0 block of the LPC11U14 MCU.

It must be noted that the SPI communication may interfere with the measured angular rate signal due to sensor internal capacitive coupling. If the harmonic overtones of the SPI communication activity are close to the gyro operational frequency, the SPI crosstalk can be seen as increased noise level in the angular rate signal.

This crosstalk can be eliminated by choosing the output data rate (sample rate) in a suitable way, i.e. avoiding the overtones of the gyro operation frequency. For optimum performance it is recommended that 2.3 kHz or 3.2 kHz output data rate is used with the maximum serial clock (SCK) frequency (8 MHz). The design performance should be verified carefully.

For these reasons in this example the SPI clock is set to 8 MHz and the output data rate is 2300 Hz. The LPC11U14 MHz internal system clock is 48 MHz.

3.1 Code Flowchart

The 2300 Hz sample rate used in this example is typically too high for applications and therefore it is here reduced to 230 Hz by averaging by ten. The interrupt routine collects the sum of ten samples for rate, accelerations and temperature and then passes them to the main data processing loop.

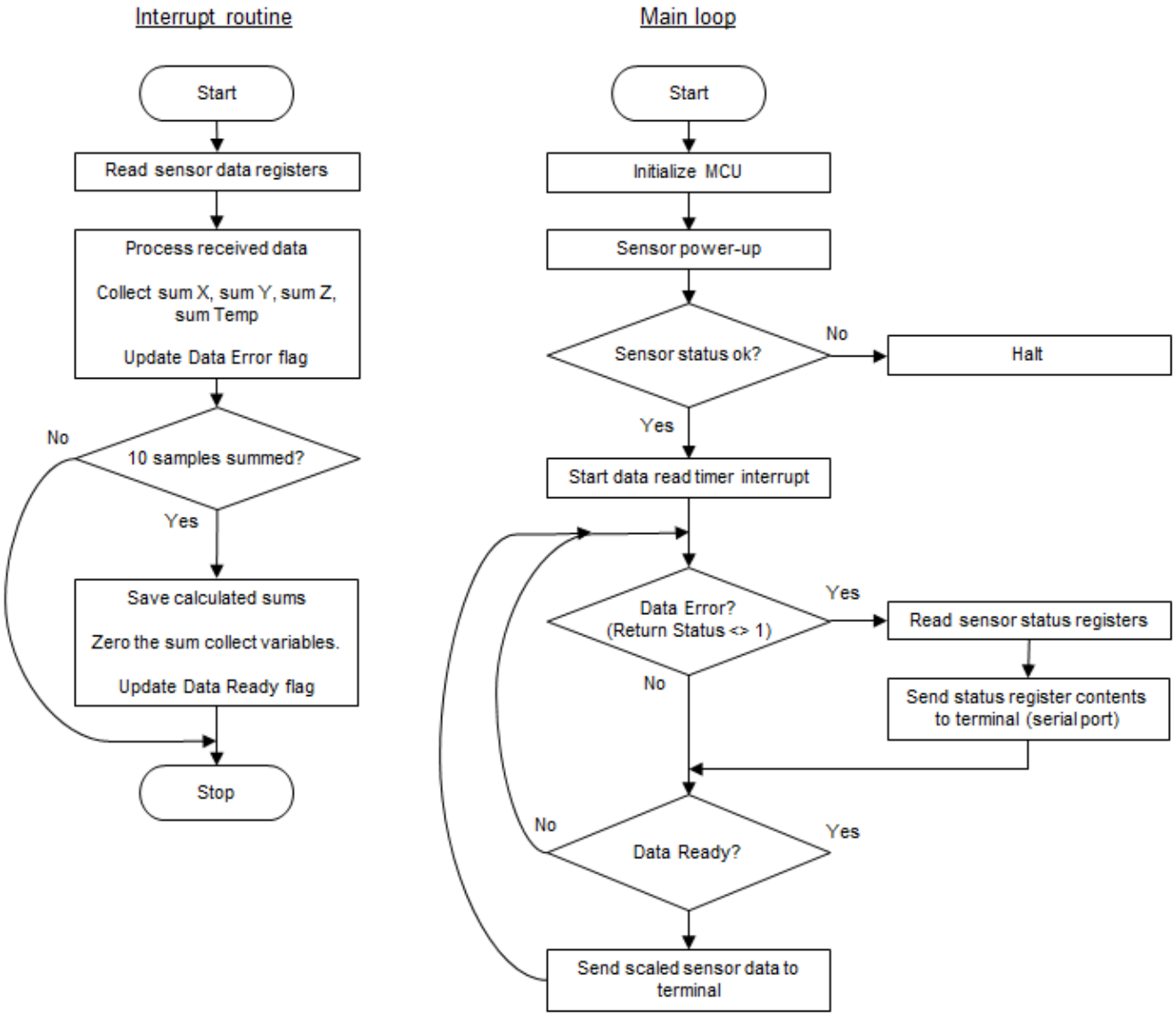

Figure 2. Example Code Flowchart

3.2 C-Code Listing

```
//*****************************************************************************
// SCC2xxx Demo - SPI Interface to SCC2000 series Combo Sensor (Gyroscope + Accelerometer)
//
// Uses NXP LPCXpresso Development Platform LPC11U14 (Cortex-M0). Measurement results
// sent to PC terminal software thru UART.
//
// The code supports SCC2x30-D08 gyro accelerometer combo part numbers as well as
// SCR2100 gyro only versions. If SCC2230-E02 is used, the accelerometer sensitivity
// should be updated from 1962LSB/g to 5886LSB/g.
//
//
// This software is released under the BSD license as follows.
// Copyright (c) 2014, Murata Electronics Oy.
// 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.
//     3. Neither the name of Murata Electronics Oy nor the names of its
//        contributors may be used to endorse or promote products derived
//        from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "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
// COPYRIGHT HOLDER 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 <stdio.h>
#include <stdbool.h>
#include "LPC11Uxx.h"                           // LPC11Uxx definitions

// Product types
#define SCR 0                                   // Gyro-only
#define SCC 1                                   // Gyro & Accelerometer

// Select the correct product type here
#define PRODUCT_TYPE         SCC
//#define PRODUCT_TYPE       SCR

// SCC2xxx definitions
// Gyroscope and accelerometer sensitivities
#define GYRO_SENSITIVITY     50                 // LSB/dps
#define ACC_SENSITIVITY      1962               // LSB/g


// SCC2xxx pin definitions
#define PIN_CSB              (1 << 2)
#define PIN_EXTRESN          (1 << 19)

// SCC2xxx status register bits
#define BIT_LOOPF_OK         0x0040
```

```
// Standard requests
#define REQ_READ_RATE         0x040000f7
#define REQ_READ_ACC_X        0x100000e9
#define REQ_READ_ACC_Y        0x140000ef
#define REQ_READ_ACC_Z        0x180000e5
#define REQ_READ_TEMP         0x1c0000e3
#define REQ_WRITE_FLT_60      0xfc200006
#define REQ_WRITE_FLT_10      0xfc1000c7
#define REQ_READ_STAT_SUM     0x7c0000b3
#define REQ_READ_RATE_STAT1   0x240000c7
#define REQ_READ_RATE_STAT2   0x280000cd
#define REQ_READ_ACC_STAT     0x3c0000d3
#define REQ_READ_COM_STAT1    0x6c0000ab

// Special requests
#define REQ_HARD_RESET        0xD8000431

// Frame field masks
#define OPCODE_FIELD_MASK     0xFC000000
#define RS_FIELD_MASK         0x03000000
#define DATA_FIELD_MASK       0x00FFFF00
#define CRC_FIELD_MASK        0x000000FF


// MCU definitions
// SSP Status register
#define SSPSR_TFE             (1 << 0)
#define SSPSR_TNF             (1 << 1)
#define SSPSR_RNE             (1 << 2)
#define SSPSR_RFF             (1 << 3)
#define SSPSR_BSY             (1 << 4)

// GPIO ports
#define PORT0                 0
#define PORT1                 1

// SPI read and write buffer size
#define FIFOSIZE              8

// SSP CR0 register
#define SSPCR0_DSS            (1 << 0)
#define SSPCR0_FRF            (1 << 4)
#define SSPCR0_SPO            (1 << 6)
#define SSPCR0_SPH            (1 << 7)
#define SSPCR0_SCR            (1 << 8)

// SSP CR1 register
#define SSPCR1_LBM            (1 << 0)
#define SSPCR1_SSE            (1 << 1)
#define SSPCR1_MS             (1 << 2)
#define SSPCR1_SOD            (1 << 3)


// SSP Interrupt Mask Set/Clear register
#define SSPIMSC_RORIM         (1 << 0)
#define SSPIMSC_RTIM          (1 << 1)
#define SSPIMSC_RXIM          (1 << 2)
#define SSPIMSC_TXIM          (1 << 3)

// USART Line Status Register
#define LSR_RDR               (0x01 << 0)
#define LSR_OE                (0x01 << 1)
#define LSR_PE                (0x01 << 2)
#define LSR_FE                (0x01 << 3)
#define LSR_BI                (0x01 << 4)
#define LSR_THRE              (0x01 << 5)
#define LSR_TEMT              (0x01 << 6)
#define LSR_RXFE              (0x01 << 7)
```

```
// Function prototypes
void SystemInit(void);
void Main_PLL_Setup(void);
void SSP_IOConfig(void);
void SSP_Init(void);
uint32_t SendRequest(uint32_t Request);
uint8_t CalculateCRC(uint32_t Data);
static uint8_t CRC8(uint32_t BitValue, uint8_t CRC);
void Wait_us(uint16_t us);
void Wait_ms(uint16_t ms);
void UARTInit(uint32_t baudrate);
void Print_String(char *str_ptr);
void ReadAndProcessData(void);


uint32_t Response_Rate;
uint32_t Response_Acc_X;
uint32_t Response_Acc_Y;
uint32_t Response_Acc_Z;
uint32_t Response_Temp;
int16_t Rate;
int16_t Acc_X;
int16_t Acc_Y;
int16_t Acc_Z;
int16_t Temp;
uint8_t RSdata;
uint8_t DataError;
uint8_t DataReady;
int32_t Sum_Rate;
int32_t Sum_X;
int32_t Sum_Y;
int32_t Sum_Z;
int32_t Sum_Temp;
int32_t Result_Rate;
int32_t Result_Acc_X;
int32_t Result_Acc_Y;
int32_t Result_Acc_Z;
int32_t Result_Temp;
uint16_t LoopCount;


// Wait us, depends on clock frequency so adjust accordingly
void Wait_us(uint16_t us)
{
    uint32_t a;
    uint32_t b;

    b = us << 2;
    for (a = b - 2; a > 0; a--)
    {
        __NOP();
        __NOP();
        __NOP();
    }
}


// Wait ms
void Wait_ms(uint16_t ms)
{
    uint16_t Count;

    for (Count = 0; Count < ms; Count++) Wait_us(1000);
}
```

```
// Initialize UART
void UARTInit(uint32_t baudrate)
{
    uint32_t Fdiv;
    volatile uint32_t regVal;

    LPC_IOCON->PIO0_18 &= ~0x07;                    // UART I/O config
    LPC_IOCON->PIO0_18 |= 0x01;                     // UART RXD
    LPC_IOCON->PIO0_19 &= ~0x07;
    LPC_IOCON->PIO0_19 |= 0x01;                     // UART TXD

    // Enable UART clock
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 12);
    LPC_SYSCON->UARTCLKDIV = 0x01;                  // Divided by 1
    LPC_USART->LCR = 0x83;                          // 8 bits, no Parity, 1 Stop bit

    // Set baud rate, AHB frequency = 48 MHz
    Fdiv = ((48000000UL/LPC_SYSCON->UARTCLKDIV)/16)/baudrate;
    LPC_USART->DLM = Fdiv / 256;
    LPC_USART->DLL = Fdiv % 256;
    LPC_USART->FDR = 0x10;                          // Default
    LPC_USART->LCR = 0x03;                          // DLAB = 0
    LPC_USART->FCR = 0x07;                          // Enable and reset TX and RX FIFO.

    // Read to clear the line status.
    regVal = LPC_USART->LSR;

    // Ensure a clean start, no data in either TX or RX FIFO.
    while (( LPC_USART->LSR & (LSR_THRE|LSR_TEMT)) != (LSR_THRE|LSR_TEMT));
    while ( LPC_USART->LSR & LSR_RDR )
    {
        regVal = LPC_USART->RBR;                    // Dump data from RX FIFO
    }

    return;
}


// Send string to UART
void Print_String(char *str_ptr)
{
    while (*str_ptr != 0x00)
    {
        while ((LPC_USART->LSR & 0x60) != 0x60);
        LPC_USART->THR = *str_ptr;
        str_ptr++;
    }
    return;
}


void SSP_Init(void)
{
    uint8_t i, Dummy = Dummy;                       // Just to suppress complier warning

    // Set DSS data to 16-bit, Frame format SPI, CPOL = 0, CPHA = 0 and SCR = 5
    LPC_SSP0->CR0 = 0x020F;
    // SSPCPSR clock prescale register, master mode, minimum divisor is 0x02
    LPC_SSP0->CPSR = 0x2;

    for (i = 0; i < FIFOSIZE; i++)
    {
        Dummy = LPC_SSP0->DR;                       // clear the RxFIFO
    }

    // Master mode
    LPC_SSP0->CR1 = SSPCR1_SSE;
    // Set SSPINMS registers to enable interrupts
    // enable all error related interrupts
    LPC_SSP0->IMSC = SSPIMSC_RORIM | SSPIMSC_RTIM;

    return;
}
```

```
void SSP_IOConfig(void)
{
    LPC_SYSCON->PRESETCTRL      |= (1 << 0);
    LPC_SYSCON->SYSAHBCLKCTRL   |= (1 << 11);
    LPC_SYSCON->SSP0CLKDIV      = 0x01;             // Divided by 1
    LPC_IOCON->PIO0_8           &= ~0x07;           // SSP I/O config
    LPC_IOCON->PIO0_8           |= 0x01;            // SSP MISO
    LPC_IOCON->PIO0_9           &= ~0x07;
    LPC_IOCON->PIO0_9           |= 0x01;            // SSP MOSI

    // SSP CLK can be routed to different pins, use PIO01_29 for SCK.
    LPC_IOCON->PIO1_29 &= ~0x07;                    // SSP CLK
    LPC_IOCON->PIO1_29 = 0x01;

    // Enable AHB clock to the GPIO domain.
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 6);

    LPC_IOCON->PIO0_2 &= ~0x07;                     // SSP SSEL (CSB) is a GPIO pin PIO0_2

    // Port0, bit 2 is set to GPIO output and high, CSB
    LPC_GPIO->DIR[PORT0] |= PIN_CSB;
    LPC_GPIO->SET[PORT0] = PIN_CSB;

    // Port1, bit 19 is set to GPIO output and high, EXTRESN
    LPC_GPIO->DIR[PORT1] |= PIN_EXTRESN;
    LPC_GPIO->SET[PORT1] = PIN_EXTRESN;

    return;
}


void Main_PLL_Setup ( void )
{

    LPC_SYSCON->SYSPLLCLKSEL = 0x01;                // Select system OSC as PLL input

    LPC_SYSCON->SYSPLLCLKUEN  = 0x01;               // Update clock source
    LPC_SYSCON->SYSPLLCLKUEN  = 0x00;               // Toggle Update Register once
    LPC_SYSCON->SYSPLLCLKUEN  = 0x01;
    while (!(LPC_SYSCON->SYSPLLCLKUEN & 0x01));     // Wait until updated

    // PSEL = 2 (Post divider ratio P = 4, division ratio = 2 x P = 8)
    // MSEL = 1 (Feedback divider value M = value + 1 = 2) -> MCLK = 48 MHz
    LPC_SYSCON->SYSPLLCTRL    = 0x0023;
    LPC_SYSCON->PDRUNCFG     &= ~(1 << 7);          // Power-up SYSPLL
    while (!(LPC_SYSCON->SYSPLLSTAT & 0x01));       // Wait until PLL locked

    LPC_SYSCON->MAINCLKSEL   = 0x03;                // Main clock source = PLL output
    LPC_SYSCON->MAINCLKUEN   = 0x01;                // Update MCLK clock source
    LPC_SYSCON->MAINCLKUEN   = 0x00;                // Toggle update register once
    LPC_SYSCON->MAINCLKUEN   = 0x01;
    while ( !(LPC_SYSCON->MAINCLKUEN & 0x01) );     // Wait until updated

    LPC_SYSCON->SYSAHBCLKDIV = 0x01;                // SYS AHB clock

    return;
}


void SystemInit(void)
{
    uint32_t i;

    // Bit 0 default is crystal bypass, bit1 0=0~20Mhz crystal input, 1=15~50Mhz crystal input.
    LPC_SYSCON->SYSOSCCTRL = 0x00;

    // Main system OSC run is cleared, bit 5 in PDRUNCFG register
    LPC_SYSCON->PDRUNCFG &= ~(1 << 5);
    // Wait for OSC to be stabilized, no status indication, dummy wait.
    for (i = 0; i < 0x100; i++) __NOP();

    Main_PLL_Setup();
```

```
    // Enable USB clock. USB clock bit 8 and 10 in PDRUNCFG.
    LPC_SYSCON->PDRUNCFG &= ~((1 << 8)|(1 << 10));

    // System clock to the IOCON needs to be enabled or most of the I/O related
    // peripherals won't work.
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 16);

    return;
}


// Send request to sensor and read back the response for previous request.
uint32_t SendRequest(uint32_t Request)
{
    uint32_t Response;

    LPC_GPIO->CLR[PORT0] = PIN_CSB;

    Response = LPC_SSP0->DR;                        // Read RX buffer just to clear interrupt flag

    // Move on only if NOT busy and TX FIFO not full
    while ((LPC_SSP0->SR & (SSPSR_TNF|SSPSR_BSY)) != SSPSR_TNF);
    LPC_SSP0->DR = Request >> 16;                   // Write Request high word to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);               // Wait until the Busy bit is cleared
    Response = LPC_SSP0->DR;                        // Read RX buffer (Response high word)
    Response <<= 16;

    LPC_SSP0->DR = Request & 0x0000FFFF;            // Write Request low word to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);               // Wait until the Busy bit is cleared
    Response |= LPC_SSP0->DR;                       // Read RX buffer (Response low word)

    LPC_GPIO->SET[PORT0] = PIN_CSB;

    return Response;
}


// Calculate CRC for 24 MSB's of the 32 bit dword
// (8 LSB's are the CRC field and are not included in CRC calculation)
uint8_t CalculateCRC(uint32_t Data)
{
    uint32_t BitMask;
    uint32_t BitValue;
    uint8_t CRC = 0xFF;

    for (BitMask = 0x80000000; BitMask != 0x80; BitMask >>= 1)
    {
        BitValue = Data & BitMask;
        CRC = CRC8(BitValue, CRC);
    }

    CRC = (uint8_t)~CRC;
    return CRC;
}


static uint8_t CRC8(uint32_t BitValue, uint8_t CRC)
{
    uint8_t Temp = (uint8_t)(CRC & 0x80);
    if (BitValue != 0)
    {
        Temp ^= 0x80;
    }
    CRC <<= 1;
    if (Temp > 0)
    {
        CRC ^= 0x1D;
    }

    return CRC;
}
```

```
void TIMER16_0_IRQHandler(void)
{
    if ( LPC_CT16B0->IR & (0x1 << 0) )
    {
        LPC_CT16B0->IR = 0x1 << 0;                  // clear interrupt flag
        // Read temperature, rate & accelerations
        ReadAndProcessData();
    }
    if ( LPC_CT16B0->IR & (0x1 << 1) )
    {
        LPC_CT16B0->IR = 0x1 << 1;                  // clear interrupt flag
    }
    if ( LPC_CT16B0->IR & (0x1 << 4) )              // Bit4 CR0INT
    {
        LPC_CT16B0->IR = 0x1 << 4;                  // clear interrupt flag
    }
    if ( LPC_CT16B0->IR & (0x1 << 6) )              // Bit6 CR1INT
    {
        LPC_CT16B0->IR = 0x1 << 6;                  // clear interrupt flag
    }
    return;
}


void ReadAndProcessData(void)
{
    // Read temperature, rate & accelerations. Note: interleaved reading due to off-frame protocol
    Response_Temp = SendRequest(REQ_READ_RATE);
#if PRODUCT_TYPE == SCC
    Response_Rate = SendRequest(REQ_READ_ACC_X);
    Response_Acc_X = SendRequest(REQ_READ_ACC_Y);
    Response_Acc_Y = SendRequest(REQ_READ_ACC_Z);
    Response_Acc_Z = SendRequest(REQ_READ_TEMP);
#else
    Response_Rate = SendRequest(REQ_READ_TEMP);
#endif

    // Handle rate data
    Rate = (Response_Rate & DATA_FIELD_MASK) >> 8;
    RSdata = (Response_Rate & RS_FIELD_MASK) >> 24;
    if (RSdata != 1) DataError = true;
    Sum_Rate += Rate;

    // Check CRC if necessary
    // if (CalculateCRC(Response_Rate) != (Response_Rate & CRC_FIELD_MASK)) DataError = true;

#if PRODUCT_TYPE == SCC
    // Handle accelerometer data
    Acc_X = (Response_Acc_X & DATA_FIELD_MASK) >> 8;
    RSdata = (Response_Acc_X & RS_FIELD_MASK) >> 24;
    if (RSdata != 1) DataError = true;
    Sum_X += Acc_X;
    Acc_Y = (Response_Acc_Y & DATA_FIELD_MASK) >> 8;
    RSdata = (Response_Acc_Y & RS_FIELD_MASK) >> 24;
    if (RSdata != 1) DataError = true;
    Sum_Y += Acc_Y;
    Acc_Z = (Response_Acc_Z & DATA_FIELD_MASK) >> 8;
    RSdata = (Response_Acc_Z & RS_FIELD_MASK) >> 24;
    if (RSdata != 1) DataError = true;
    Sum_Z += Acc_Z;
#endif

    // Handle temperature data
    Temp = (Response_Temp & DATA_FIELD_MASK) >> 8;
    RSdata = (Response_Temp & RS_FIELD_MASK) >> 24;
    if (RSdata != 1) DataError = true;
    Sum_Temp += Temp;

    if (LoopCount >= 10)
    {
        Result_Rate = Sum_Rate;
        Sum_Rate = 0;
```

```
#if PRODUCT_TYPE == SCC
        Result_Acc_X = Sum_X;
        Sum_X = 0;
        Result_Acc_Y = Sum_Y;
        Sum_Y = 0;
        Result_Acc_Z = Sum_Z;
        Sum_Z = 0;
#endif
        Result_Temp = Sum_Temp;
        DataReady = true;
        LoopCount = 0;
        Sum_Temp = 0;
    }
    LoopCount++;
}


int main(void)
{
    char Buffer[80];

    uint32_t Response_StatSum;
    uint32_t Response_RateStat1;
    uint32_t Response_RateStat2;
    uint32_t Response_AccStat;
    uint32_t Response_ComStat1;
    bool     StartupOK;

    Wait_ms(100);                               // Wait for power supply to stabilize

    SystemInit();

    // initialize SSP port, share pins with SPI1 on port2(p2.0-3).
    SSP_IOConfig();
    SSP_Init();

    UARTInit(230400);                           // Initialize serial port
    Print_String("\f");

    // Reset sensor
    LPC_GPIO->CLR[1] = PIN_EXTRESN;             // Gyro reset pin low
    Wait_ms(2);
    LPC_GPIO->SET[1] = PIN_EXTRESN;             // Gyro reset pin high

    // Sensor power-up
    //-----------------------------------------------------------------------------------------
    Wait_ms(25);                                // Wait 25 ms until the SCC2000 is accessible via SPI
    SendRequest(REQ_WRITE_FLT_60);              // Set output filter to 60 Hz
    Wait_ms(595);                               // NOTE: wait 595 ms in case the output filter is set
                                                // to 60 Hz, 725 ms if the filter is set to 10 Hz
    // Clear status registers
    //-----------------------------------------------------------------------------------------
    SendRequest(REQ_READ_RATE_STAT1);
    SendRequest(REQ_READ_RATE_STAT2);
    SendRequest(REQ_READ_ACC_STAT);
    SendRequest(REQ_READ_COM_STAT1);
    SendRequest(REQ_READ_STAT_SUM);

    // Check functionality
    //-----------------------------------------------------------------------------------------
#if PRODUCT_TYPE == SCC
    StartupOK = true;
    Response_StatSum = SendRequest(REQ_READ_STAT_SUM);     // Read Status Summary register again to get
    RSdata = (Response_StatSum & RS_FIELD_MASK) >> 24;     // the correct status data from off-frame
                                                           // protocol
    if (RSdata != 1) StartupOK = false;
    if (!StartupOK)
    {
        Print_String("\r\nStart-up RS error\r\n");
        sprintf(Buffer, "Status Summary = 0x%X\r\n", Response_StatSum);
        Print_String(Buffer);

        SendRequest(REQ_READ_RATE_STAT1);
```

```
        Response_RateStat1 = SendRequest(REQ_READ_RATE_STAT2);
        Response_RateStat2 = SendRequest(REQ_READ_ACC_STAT);
        Response_AccStat = SendRequest(REQ_READ_COM_STAT1);
        Response_ComStat1 = SendRequest(REQ_READ_RATE);
        sprintf(Buffer, "Rate Status 1 = 0x%X\r\n", Response_RateStat1);
        Print_String(Buffer);
        sprintf(Buffer, "Rate Status 2 = 0x%X\r\n", Response_RateStat2);
        Print_String(Buffer);
        sprintf(Buffer, "Acceleration Status = 0x%X\r\n", Response_AccStat);
        Print_String(Buffer);
        sprintf(Buffer, "Common Status 1 = 0x%X\r\n", Response_ComStat1);
        Print_String(Buffer);

        while(1);       // Halt execution
    }
    Sum_Rate = 0;
    Sum_X = 0;
    Sum_Y = 0;
    Sum_Z = 0;
    Sum_Temp = 0;

#else
    Response_StatSum =   SendRequest(REQ_READ_RATE_STAT1);
    Response_RateStat1 = SendRequest(REQ_READ_RATE_STAT2);
    Response_RateStat2 = SendRequest(REQ_READ_COM_STAT1);
    Response_ComStat1 =  SendRequest(REQ_READ_COM_STAT1);

    StartupOK = true;
    if (((Response_RateStat1 & 0x00C03F00) != 0x00C03F00) || (Response_RateStat1 == 0xFFFFFFFF))
        StartupOK = false;
    if (((Response_RateStat2 & 0x0001FF00) != 0x0001FF00) || (Response_RateStat2 == 0xFFFFFFFF))
        StartupOK = false;
    if (((Response_ComStat1 & 0x00F87700)  != 0x00F87700) || (Response_ComStat1 == 0xFFFFFFFF))
        StartupOK = false;
    if (((Response_StatSum & 0x00004100)   != 0x00004100) || (Response_StatSum == 0xFFFFFFFF))
        StartupOK = false;

    if (!StartupOK)
    {
        sprintf(Buffer, "\r\nStart-up RS error\r\n");
        Print_String(Buffer);

        sprintf(Buffer, "Status Summary = 0x%X\r\n", Response_StatSum);
        Print_String(Buffer);
        sprintf(Buffer, "Rate Status 1 = 0x%X\r\n",  Response_RateStat1);
        Print_String(Buffer);
        sprintf(Buffer, "Rate Status 2 = 0x%X\r\n",  Response_RateStat2);
        Print_String(Buffer);
        sprintf(Buffer, "Common Status = 0x%X\r\n",  Response_ComStat1);
        Print_String(Buffer);

        while(1);       // Halt execution
    }
    Sum_Rate = 0;
    Sum_Temp = 0;
#endif

    LoopCount = 1;
    DataError = false;
    DataReady = false;

    SendRequest(REQ_READ_TEMP);               // First read temp once to get into desired
                                              // measurement cycle in off-frame protocol
    // Set the 16-bit TIMER0 interrupt routine to read sensor data at 2.3 kHz output data rate
    LPC_SYSCON->SYSAHBCLKCTRL |= (1 << 7);    // Enable AHB clock to the 16-bit counter/timer 0
                                              // (CT16B0)

    // MR0 (Match Register 0) will generate an interrupt at 2300 Hz (MCLK is 48 MHz)
    LPC_CT16B0->MR0 = 20870;
    LPC_CT16B0->MCR = (0x3 << 0);             // Interrupt and Reset on MR0
    NVIC_EnableIRQ(TIMER_16_0_IRQn);          // Enable the TIMER0 Interrupt
    LPC_CT16B0->TCR = 1;                      // Enable TIMER0
```

```
    // Main data processing loop
    //-------------------------------------------------------------------------------------------
    while (1)
    {

        if (DataError)
        {
            // In case of error read status registers
            LPC_CT16B0->TCR = 0;              // Disable TIMER0 to avoid race condition with the
                                              // interrupt routine

            // Clear the data that is coming from previous frame, request status summary
            SendRequest(REQ_READ_STAT_SUM);
            Response_StatSum = SendRequest(REQ_READ_RATE_STAT1);
            Response_RateStat1 = SendRequest(REQ_READ_RATE_STAT2);
            Response_RateStat2 = SendRequest(REQ_READ_COM_STAT1);
#if PRODUCT_TYPE == SCC
            Response_ComStat1 = SendRequest(REQ_READ_ACC_STAT);

            // Request temperature data to get the measurement loop to
            // continue correctly after reading the status registers
            Response_AccStat = SendRequest(REQ_READ_TEMP);

            sprintf(Buffer, "%08X %08X %08X %08X %08X\r\n", Response_StatSum, Response_RateStat1,
                Response_RateStat2, Response_AccStat, Response_ComStat1);
#else
            Response_ComStat1 = SendRequest(REQ_READ_TEMP);
            sprintf(Buffer, "%08X %08X %08X %08X\r\n", Response_StatSum, Response_RateStat1,
                Response_RateStat2, Response_ComStat1);
#endif
            LPC_CT16B0->TCR = 1;              // Enable TIMER0 again

            Print_String(Buffer);
            DataError = false;
        }
        else if (DataReady)
        {
#if PRODUCT_TYPE == SCC
            sprintf(Buffer, "%5d %5d %5d %5d %5d\r\n", Result_Rate / 10, Result_Acc_X / 10,
                Result_Acc_Y / 10, Result_Acc_Z / 10, Result_Temp / 10);
#else
            sprintf(Buffer, "%5d %5d\r\n", Result_Rate / 10, Result_Temp / 10);
#endif
            Print_String(Buffer);
            DataReady = false;
        }
    }
}
```

4 Result Data Output

In this example system the measurement and status data are sent to a PC terminal program (Tera Term) for easier examination. The serial communication parameters are: bps - 230400, Data - 8, Parity - None.

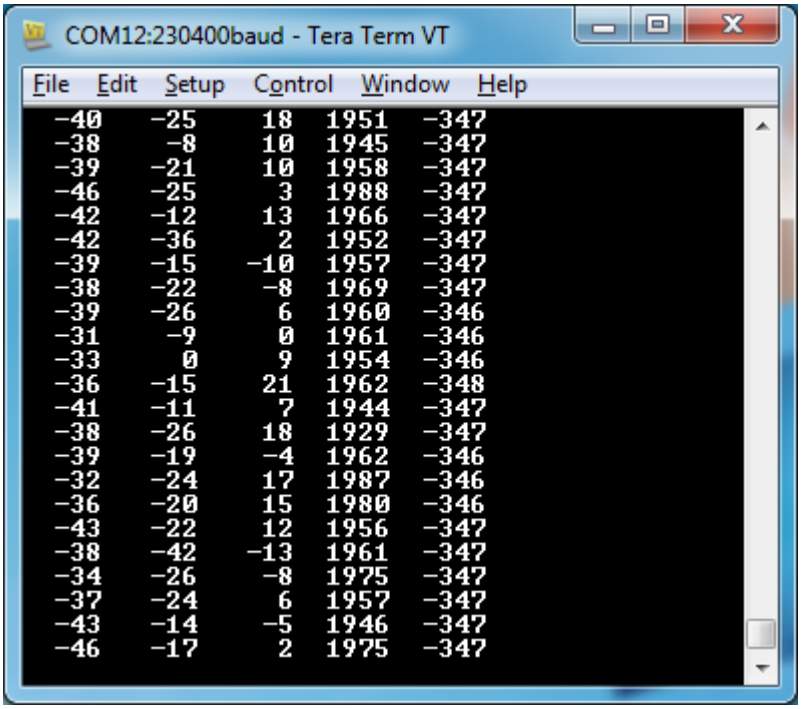

Figure 3. Sample result data captured from PC screen

Logic analyzer waveforms are shown on Figure 4 below. Note the use of off-frame protocol.

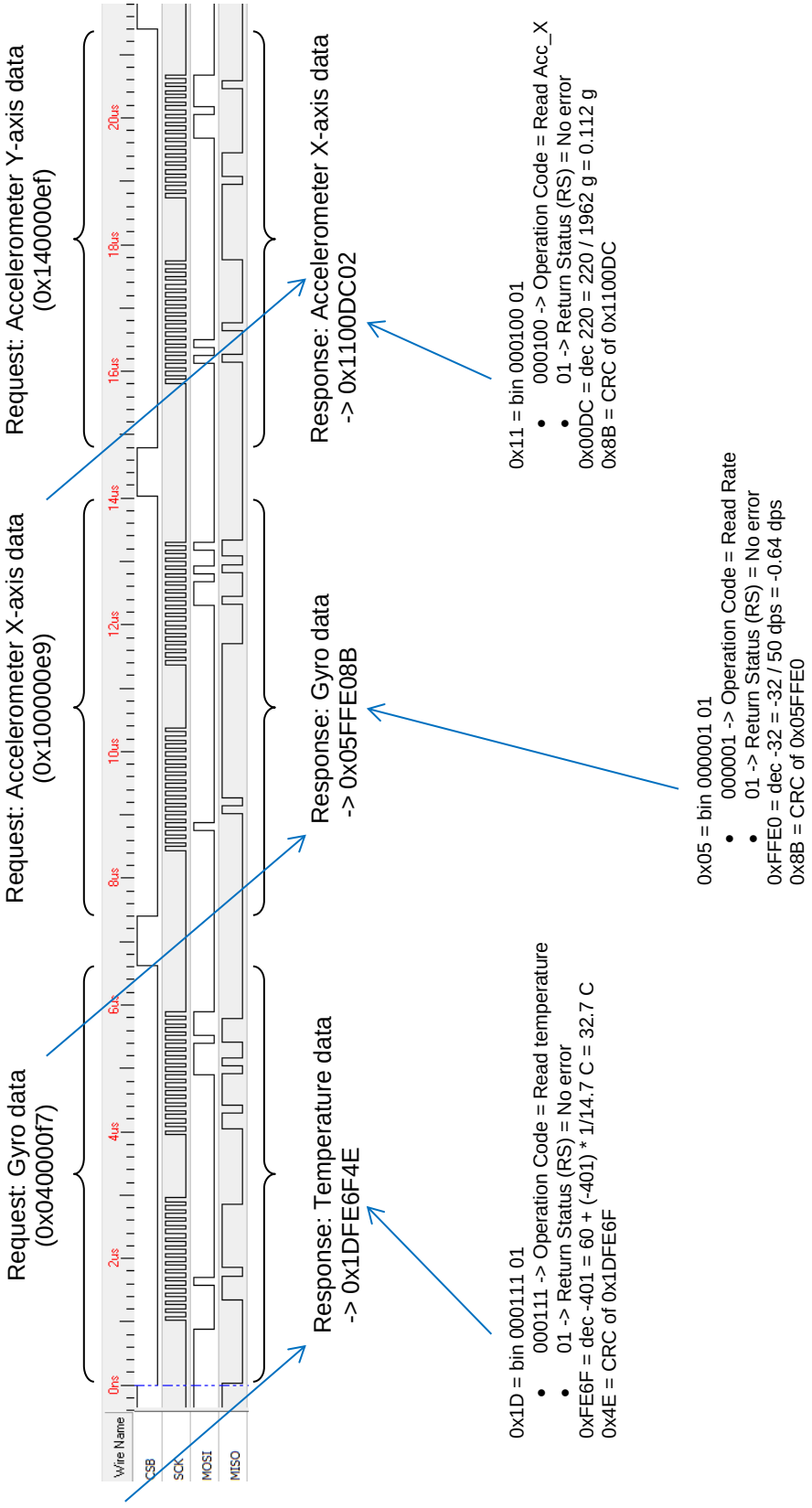

Figure 4. SPI communication waveforms (unit conversions apply for SCC2230-D08).

The complete transmission cycle is shown on Figure 5 below. Ten measurement results are averaged in the interrupt routine and the averaged results are sent out in one serial communication burst in the main loop.

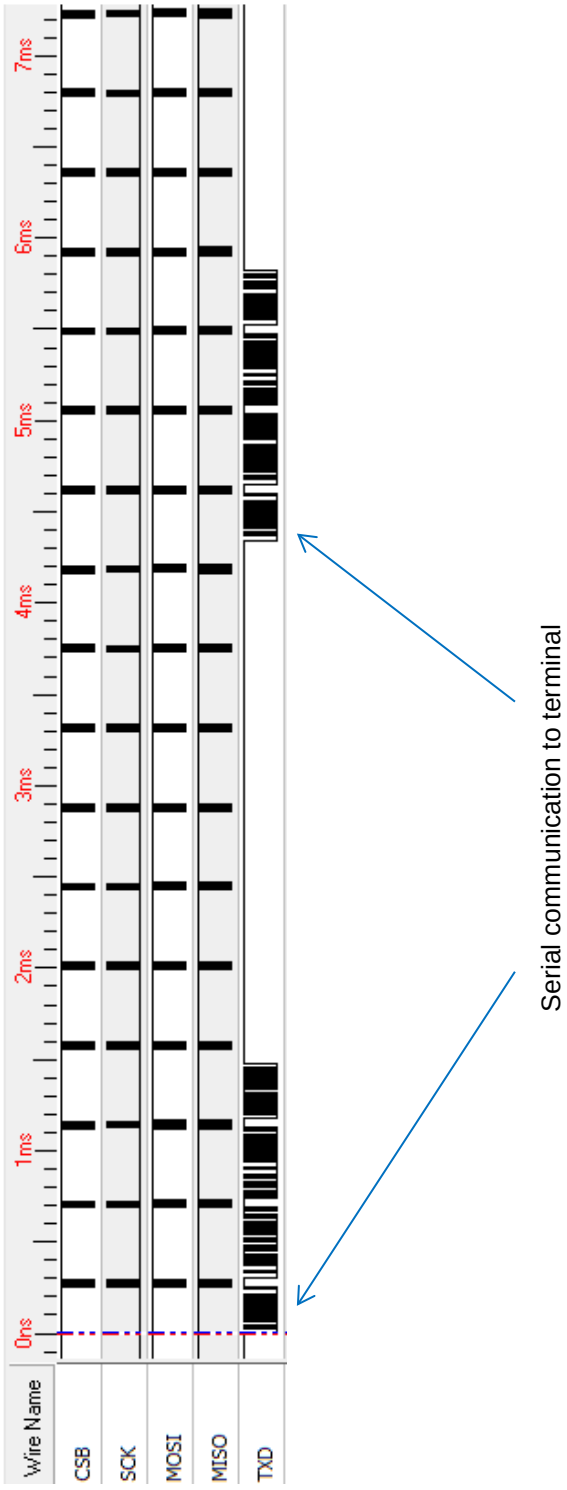

Figure 5. Complete transmission cycle