TECHNICAL NOTE

SPI COMMUNICATION WITH SCA21X0 AND SCA3100

1 Introduction

The purpose of this document is to show how to set up SPI communication between Murata SCA21X0 / SCA3100 digital accelerometers and an NXP LPC11U14 Cortex-M0 microcontroller. The code example contains following operations:

- LPC11U14 MCU configuration.
- Accelerometer start-up.
- Accelerometer Continuous Self Test (STC) is activated.
- Measurement data is read and sent to serial port.

2 Development Hardware

In this example a Murata prototype board “ADP Demo PCB” is connected to 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 PCB 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 is used (TTL-232R-3V3-WE).

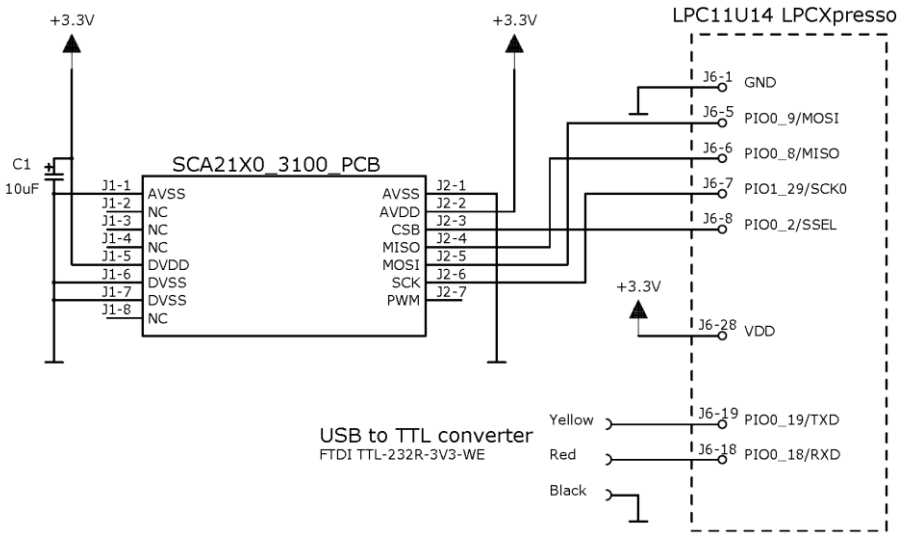

Figure 1. System schematic

3 C-Code Example

The example code was created for NXP LPCXpresso Development Board (LPC11U14) 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 the SCA21X0/3100 using SPI0 block of the LPC11U14 MCU. 48 MHz internal system clock is used and the SPI clock frequency is set to 8 MHz.

3.1 Code Flowchart

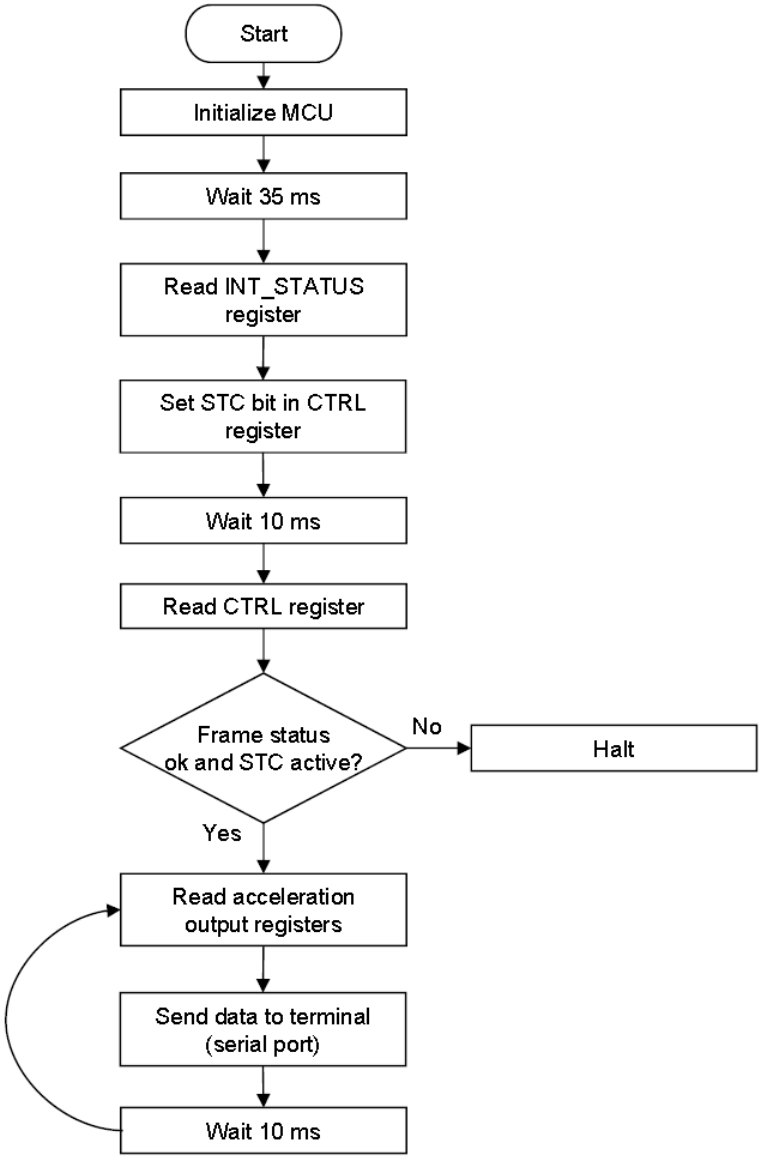

Figure 2. Example Code Flowchart

3.2 C-Code Listing

```
//******************************************************************************
// SCA21X0, SCA3100 Demo - SPI Interface to digital accelerometer sensor
//
// Uses NXP LPCXpresso Development Platform LPC11U14 (Cortex-M0). Measurement results
// sent to PC terminal software thru UART.
//
//
// This software is released under the BSD license as follows.
// Copyright (c) 2019, 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 type definitions
#define SCA3100_D04           0         // 900 counts/g
#define SCA3100_D07           1         // 650 counts/g
#define SCA2100_D02           4         // 900 counts/g
#define SCA2110_D04           3         // 900 counts/g
#define SCA2120_D06           2         // 900 counts/g

// Select the product type here
#define PRODUCT_TYPE  SCA3100_D04

// Accelerometer registers
#define REVID                 0x00
#define CTRL                  0x01
#define STATUS                0x02
#define RESET                 0x03
#define X_LSB                 0x04
#define X_MSB                 0x05
#define Y_LSB                 0x06
#define Y_MSB                 0x07
#define Z_LSB                 0x08
#define Z_MSB                 0x09
#define TEMP_LSB              0x12
#define TEMP_MSB              0x13
#define INT_STATUS            0x16
#define ID                    0x27
```

```
//Ctrl-register bits
#define BIT_PORST             0x40
#define BIT_PDOW              0x20
#define BIT_ST                0x08
#define BIT_MST               0x04
#define BIT_ST_CFG            0x02

// Pin definitions
#define PIN_CSB               (1 << 2)

// 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);
void Wait_us(uint16_t us);
void Wait_ms(uint16_t ms);
void UARTInit(uint32_t baudrate);
void Print_String(char *str_ptr);
uint8_t CalcParity(uint16_t Data);
uint8_t ReadRegister(uint8_t Address, uint8_t *Data);
uint8_t WriteRegister(uint8_t Address, uint8_t Data);
uint8_t ReadXYZ(int16_t *Xacc, int16_t *Yacc, int16_t *Zacc);
uint8_t ReadXY(int16_t *Xacc, int16_t *Yacc);
uint8_t ReadYZ(int16_t *Yacc, int16_t *Zacc);


// 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);
}


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
  Fdiv = ((48000000UL/LPC_SYSCON->UARTCLKDIV)/16)/baudrate ;// Set baud rate, System AHB freq = 48 MHz
  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;
}


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 8-bit, Frame format SPI, CPOL = 0, CPHA = 0 and SCR = 2
   LPC_SSP0->CR0 = 0x0207;

   // 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;

   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
  LPC_SYSCON->SYSPLLCTRL    = 0x0023;          // 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->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 stablized, 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;
}

uint8_t CalcParity(uint16_t Data)
{
   // svicent http://www.edaboard.com/thread46732.html
   uint8_t NoOfOnes = 0;

   while(Data != 0)
   {
      NoOfOnes++;
      Data &= (Data - 1); // Loop will execute once for each bit of Data set
   }
   // if NoOfOnes is odd, least significant bit will be 1
   return (~NoOfOnes & 1);
}


uint8_t ReadRegister(uint8_t Address, uint8_t *Data)
{
   uint8_t Status;

   Address <<= 2;                           // Address to be shifted left by 2 and RW bit to be reset
   Address += CalcParity(Address);          // Add parity bit

   LPC_GPIO->CLR[PORT0] = PIN_CSB;          // Select sensor

   Status = 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 = Address;                  // Write address to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Status = LPC_SSP0->DR;                   // Read RX buffer (status byte)

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   *Data = LPC_SSP0->DR;                    // Read RX buffer (data byte)

   LPC_GPIO->SET[PORT0] = PIN_CSB;          // Deselect sensor

   return Status;
}

uint8_t WriteRegister(uint8_t Address, uint8_t Data)
{
   uint8_t Status;
```

```
    Address <<= 2;                          // Address to be shifted left by 2
    Address |= 2;                           // RW bit to be set
    Address += CalcParity(Address);         // Add parity bit

    LPC_GPIO->CLR[PORT0] = PIN_CSB;         // Select sensor

    Status = 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 = Address;                 // Write address to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Status = LPC_SSP0->DR;                  // Read RX buffer (status byte)

    LPC_SSP0->DR = Data;                    // Write data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Data = LPC_SSP0->DR;                    // Read RX buffer (dummy)

    LPC_GPIO->SET[PORT0] = PIN_CSB;         // Deselect sensor

    return Status;
}

// Read Z-, Y- and X-acceleration data registers using the decremented reading feature
// and convert to 16-bit signed values
uint8_t ReadXYZ(int16_t *Xacc, int16_t *Yacc, int16_t *Zacc)
{
    uint8_t Status;
    uint8_t Result;
    uint8_t Address = Z_MSB;

    Address <<= 2;                          // Address shifted left by 2 and RW bit to be reset
    Address += CalcParity(Address);         // Add parity bit

    LPC_GPIO->CLR[PORT0] = PIN_CSB;         // Select sensor

    Result = 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 = Address;                 // Write address to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Status = LPC_SSP0->DR;                  // Read RX buffer (status byte)

    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Result = LPC_SSP0->DR;                  // Read RX buffer (data byte)
    *Zacc = (int16_t)(Result << 8);         // Store Z MSByte

    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Result = LPC_SSP0->DR;                  // Read RX buffer (data byte)
    *Zacc |= Result;                        // Store Z LSByte

    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Result = LPC_SSP0->DR;                  // Read RX buffer (data byte)
    *Yacc = (int16_t)(Result << 8);         // Store Y MSByte

    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Result = LPC_SSP0->DR;                  // Read RX buffer (data byte)
    *Yacc |= Result;                        // Store Y LSByte

    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Result = LPC_SSP0->DR;                  // Read RX buffer (data byte)
    *Xacc = (int16_t)(Result << 8);         // Store X MSByte

    LPC_SSP0->DR = 0;                       // Write dummy data to TX buffer
    while (LPC_SSP0->SR & SSPSR_BSY);       // Wait until the Busy bit is cleared.
    Result = LPC_SSP0->DR;                  // Read RX buffer (data byte)
```

```
   *Xacc |= Result;                          // Store X LSByte

   LPC_GPIO->SET[PORT0] = PIN_CSB;           // Deselect sensor

   return Status;
}

// Read Y- and X-acceleration data registers using the decremented reading feature
// and convert to 16-bit signed values
uint8_t ReadXY(int16_t *Xacc, int16_t *Yacc)
{
   uint8_t Status;
   uint8_t Result;
   uint8_t Address = Y_MSB;

   Address <<= 2;                            // Address shifted left by 2 and RW bit to be reset
   Address += CalcParity(Address);           // Add parity bit

   LPC_GPIO->CLR[PORT0] = PIN_CSB;           // Select sensor

   Result = 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 = Address;                   // Write address to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);         // Wait until the Busy bit is cleared.
   Status = LPC_SSP0->DR;                    // Read RX buffer (status byte)

   LPC_SSP0->DR = 0;                         // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);         // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                    // Read RX buffer (data byte)
   *Yacc = (int16_t)(Result << 8);           // Store Y MSByte

   LPC_SSP0->DR = 0;                         // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);         // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                    // Read RX buffer (data byte)
   *Yacc |= Result;                          // Store Y LSByte

   LPC_SSP0->DR = 0;                         // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);         // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                    // Read RX buffer (data byte)
   *Xacc = (int16_t)(Result << 8);           // Store X MSByte

   LPC_SSP0->DR = 0;                         // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);         // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                    // Read RX buffer (data byte)
   *Xacc |= Result;                          // Store X LSByte

   LPC_GPIO->SET[PORT0] = PIN_CSB;           // Deselect sensor

   return Status;
}

// Read Y- and Z-acceleration data registers using the decremented reading feature
// and convert to 16-bit signed values
uint8_t ReadYZ(int16_t *Yacc, int16_t *Zacc)
{
   uint8_t Status;
   uint8_t Result;
   uint8_t Address = Z_MSB;

   Address <<= 2;                            // Address shifted left by 2 and RW bit to be reset
   Address += CalcParity(Address);           // Add parity bit

   LPC_GPIO->CLR[PORT0] = PIN_CSB;           // Select sensor

   Result = 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 = Address;                   // Write address to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);         // Wait until the Busy bit is cleared.
   Status = LPC_SSP0->DR;                    // Read RX buffer (status byte)
```

```
   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Zacc = (int16_t)(Result << 8);          // Store Z MSByte

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Zacc |= Result;                         // Store Z LSByte

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Yacc = (int16_t)(Result << 8);          // Store Y MSByte

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Yacc |= Result;                         // Store Y LSByte

   LPC_GPIO->SET[PORT0] = PIN_CSB;          // Deselect sensor

   return Status;
}

// Read X- and Z-acceleration data registers using the decremented reading feature
// and convert to 16-bit signed values
uint8_t ReadXZ(int16_t *Xacc, int16_t *Zacc)
{
   uint8_t Status;
   uint8_t Result;
   uint8_t Address = X_MSB;

   Address <<= 2;                           // Address shifted left by 2 and RW bit to be reset
   Address += CalcParity(Address);          // Add parity bit

   LPC_GPIO->CLR[PORT0] = PIN_CSB;          // Select sensor

   Result = 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 = Address;                  // Write address to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Status = LPC_SSP0->DR;                   // Read RX buffer (status byte)

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Xacc = (int16_t)(Result << 8);          // Store X MSByte

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Xacc |= Result;                         // Store X LSByte

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Zacc = (int16_t)(Result << 8);          // Store Z MSByte

   LPC_SSP0->DR = 0;                        // Write dummy data to TX buffer
   while (LPC_SSP0->SR & SSPSR_BSY);        // Wait until the Busy bit is cleared.
   Result = LPC_SSP0->DR;                   // Read RX buffer (data byte)
   *Zacc |= Result;                         // Store Z LSByte

   LPC_GPIO->SET[PORT0] = PIN_CSB;          // Deselect sensor

   return Status;
}
```

```
int main(void)
{
   char Buffer[80];
   uint8_t Frame_Status;
   uint8_t Int_Status;
   uint8_t Reg_Ctrl;
   int16_t Acc_X;
   int16_t Acc_Y;
   int16_t Acc_Z;

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

   SystemInit();
   SSP_IOConfig();                      // initialize SSP port, share pins with SPI1 on port2(p2.0-3).
   SSP_Init();
   UARTInit(230400);                    // Initialize serial port
   Print_String("\f");

   // Accelerometer start-up, SCA21X0 and SCA3100
   //-----------------------------------------------------------------------------------------------
   Wait_ms(35);            // Wait 35 ms. Memory reading and self-diagnostic. Settling of signal path.

   // Acknowledge for possible saturation (SAT-bit)
   Frame_Status = ReadRegister(INT_STATUS, &Int_Status);

   Frame_Status = WriteRegister(CTRL, BIT_ST); // Set PORST = 0, Start STC
   Wait_ms(10);                                // Wait 10 ms. STS calculation
   //-----------------------------------------------------------------------------------------------

   // Read CTRL-register to check accelerometer operation
   Frame_Status = ReadRegister(CTRL, &Reg_Ctrl);

   // If frame status bits are ok and the STC is active then start reading the outputs.
   if(((Frame_Status & 0x7E) == 2) && ((Reg_Ctrl & BIT_ST) == BIT_ST))
   {
      while(1)
      {
 #if (PRODUCT_TYPE == SCA3100_D04) || (PRODUCT_TYPE == SCA3100_D07)
         Frame_Status = ReadXYZ(&Acc_X, &Acc_Y, &Acc_Z);
         Acc_X >>= 2; Acc_Y >>= 2;  Acc_Z >>= 2;            // Scale the data (drop unused data bits)
         sprintf(Buffer, "X:%5d  Y:%5d  Z:%5d  S:%X\r\n", Acc_X, Acc_Y, Acc_Z, Frame_Status);
 #endif
 #if PRODUCT_TYPE == SCA2100_D02
         Frame_Status = ReadXY(&Acc_X, &Acc_Y);
         Acc_X >>= 2; Acc_Y >>= 2;
         sprintf(Buffer, "X:%5d  Y:%5d  FS:%X\r\n", Acc_X, Acc_Y, Frame_Status);
  #endif
  #if PRODUCT_TYPE == SCA2110_D04
         Frame_Status = ReadXZ(&Acc_X, &Acc_Z);
         Acc_X >>= 2; Acc_Z >>= 2;
         sprintf(Buffer, "X:%5d  Z:%5d  FS:%X\r\n", Acc_X, Acc_Z, Frame_Status);
  #endif
  #if PRODUCT_TYPE == SCA2120_D06
         Frame_Status = ReadYZ(&Acc_Y, &Acc_Z);
         Acc_Y >>= 2; Acc_Z >>= 2;
         sprintf(Buffer, "Y:%5d  Z:%5d  FS:%X\r\n", Acc_Y, Acc_Z, Frame_Status);
  #endif
         Print_String(Buffer);
         Wait_ms(10);
      }
   }
   else
   {
      // Start-up error, show status and stop operation
      sprintf(Buffer, "Accelerometer fail: Frame_Status=%X: Ctrl=%X\r\n", Frame_Status, Reg_Ctrl);
      Print_String(Buffer);
      while(1);
   }
```

4 SPI Waveforms

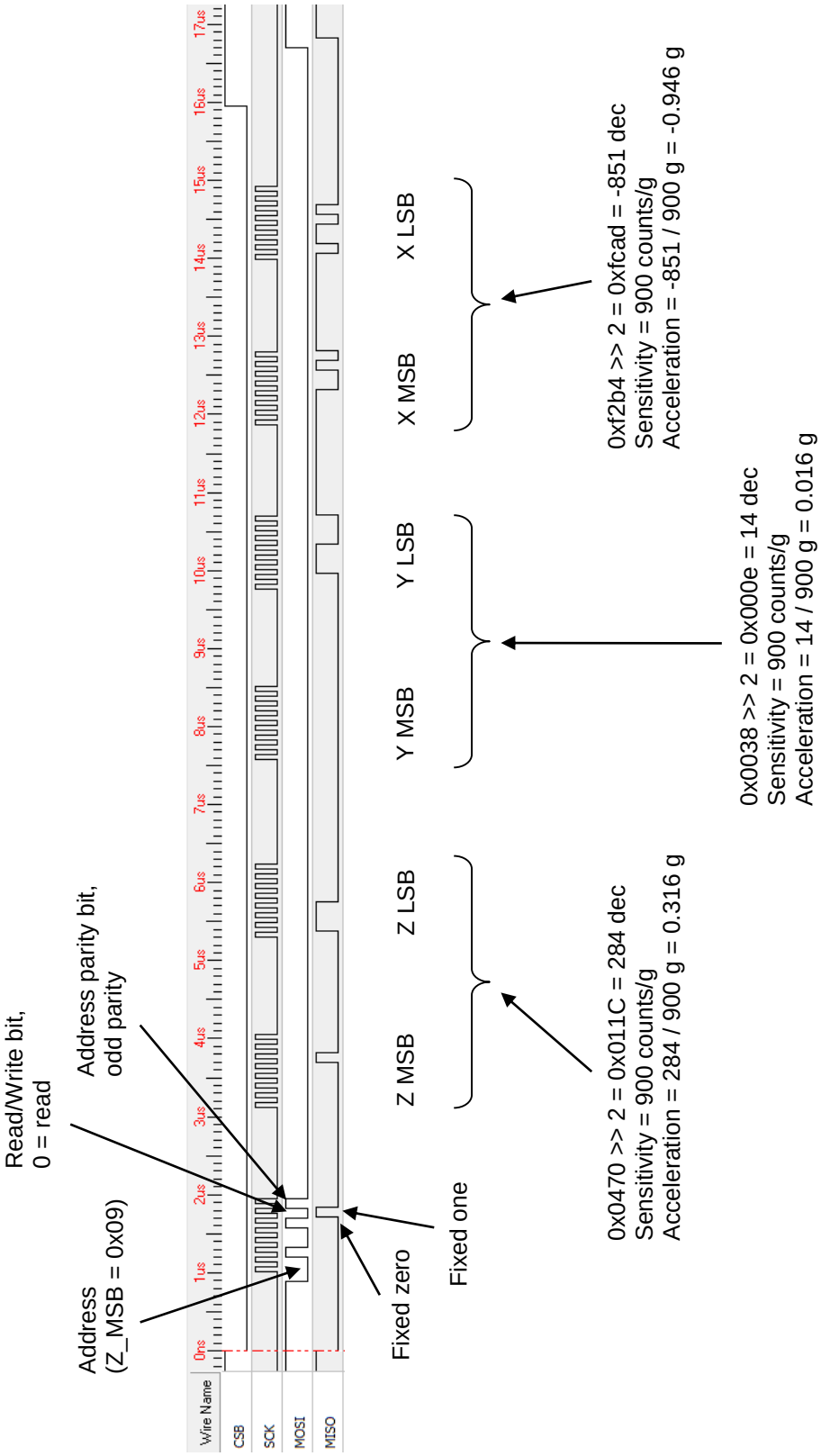

Figure 2. SPI communication waveforms (unit conversions apply for SCA3100-D04)