K64F Serial Communication Project

By Kevin Wu & Micheal Dominguez

Abstract

This project explores embedded communication techniques using the K64F microcontroller in conjunction with an FXOS8700CQ accelerometer/magnetometer sensor. We implemented and tested three fundamental serial communication protocols—UART, SPI, and I²C—by transmitting sensor data to both a PC terminal and an Arduino Uno. To assess transmission timing, we used a digital storage oscilloscope to measure end-to-end latency. Our analysis verified accurate sensor interfacing and effective communication integration. The results confirmed a measured SPI communication latency of 13 microseconds, consistent with the calculated bit-time from a 374.49 kHz clock rate. This project highlights key skills in embedded systems design, protocol implementation, and signal analysis.

Contents

1 Overview

This project was initiated to enhance our understanding of microcontroller communication protocols through the use of tools provided by the EE department at the University of California, Riverside. Using the K64F microcontroller development board, we explored the implementation of UART, SPI, and I^2C communication protocols. The goal was to interface with an onboard FXOS8700CQ accelerometer/magnetometer sensor, relay sensor data to external devices (Arduino Uno, PC), and evaluate communication reliability using electronic test tools.

2 Accomplishments

- Interfaced the FXOS8700CQ sensor with the K64F board using I^2C
- Transmitted sensor data to a PC terminal using UART and to an Arduino Uno via SPI
- Used an oscilloscope to measure signal behavior and confirm accurate data transfer
- Demonstrated complete system-level integration across three different communication protocols

3 Hardware Design

3.1 Circuit Setup

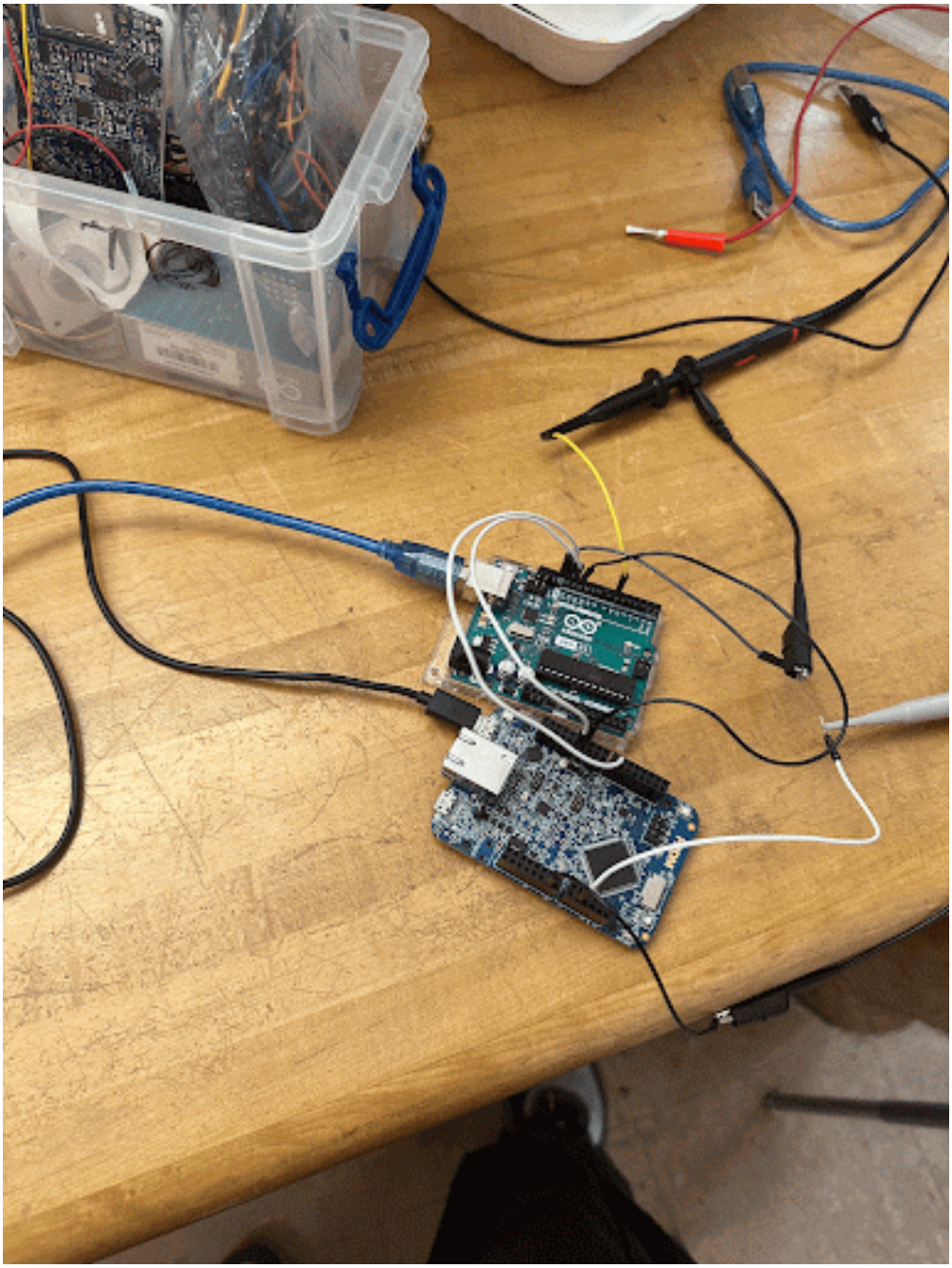

Figure 1

- **Figure 1:** Circuit configuration for Arduino Uno and K64F sensor interface

3.2 Schematic Diagram

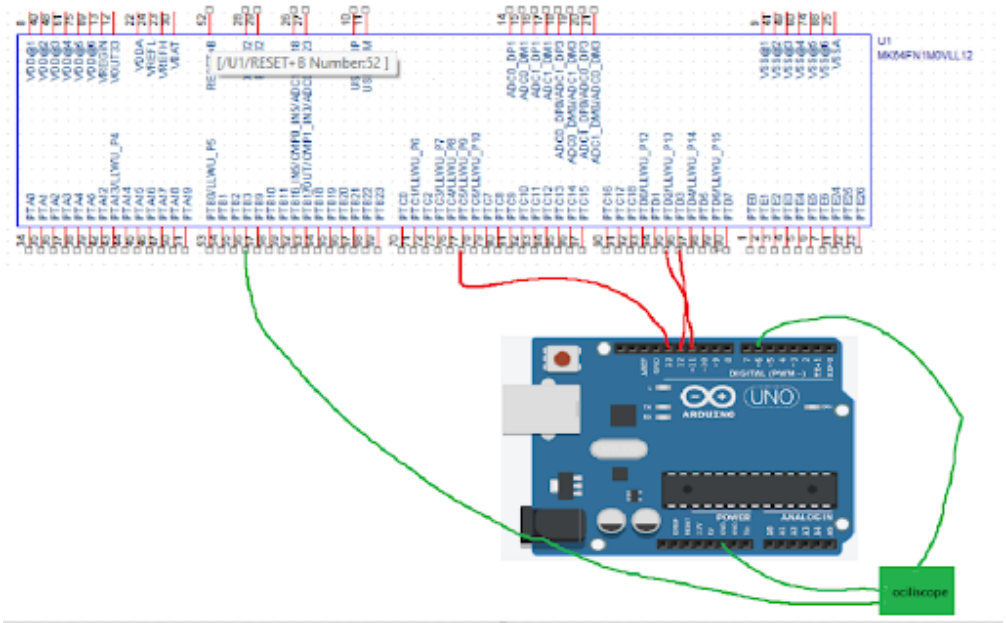

Figure 2

- **Figure 2:** Schematic showing SPI connections on K64F and Arduino Uno

4 Software Design

We used Processor Expert in Kinetis Design Studio (KDS) to configure the drivers and peripherals. The software workflow included:

- **I²C:** Initialized and read data from the FXOS8700CQ sensor
- **UART:** Sent real-time sensor readings to PC
- **SPI:** Transmitted sensor data to an Arduino Uno, visualized via oscilliscope

Software design began with the I2C programming using the KDS Processor Expert. After ensuring the serial port configuration, and the accelerometer/magnetometer sensor were connected to the correct ports (PTE and PTB), the sensors readings were being relayed to the terminal via the UART. Then we sent the data from K64F to the Arduino serial terminal using the SPI. For 3 of the SPI lines, we connected the Arduino to the corresponding pins on the K64F (11 to PTD2, 12 to PTD3, 13 to PTC5).

4.1 Code Implementation: K64F main.c

The K64F's 'main.c' manages the full embedded communication pipeline. It initializes the FXOS8700CQ sensor over I^2C and performs periodic reads of its accelerometer, magnetometer, and temperature data. This information is formatted and transmitted over both UART (to a serial terminal) and SPI (to the Arduino Uno). A GPIO pin on Port B is used to mark the start of SPI transmission by being set HIGH before 'SM1_SendBlock()' is called and LOW afterward. This enables timing synchronization for oscilloscope-based latency measurement. SPI data is transmitted as formatted strings containing sensor values, including "Who Am I" identifier, temperature, and XYZ readings.

4.2 Code Implementation: Arduino main.c

The Arduino Uno was configured as a SPI slave using the built-in SPI library. It handled data reception inside the 'SPI_STC_vect' ISR (interrupt service routine). Each received byte was buffered until a newline character was detected, indicating the end of a transmission. Once the message was complete, it was printed to the serial monitor for verification. A GPIO pin (pin 6) was pulsed high for 10 ms after each complete transmission, allowing us to mark the end of reception for oscilloscope measurements. This paired with the K64F's GPIO start signal provided a method to measure transmission latency accurately.

5 Experimental Results

5.1 Oscilloscope Analysis

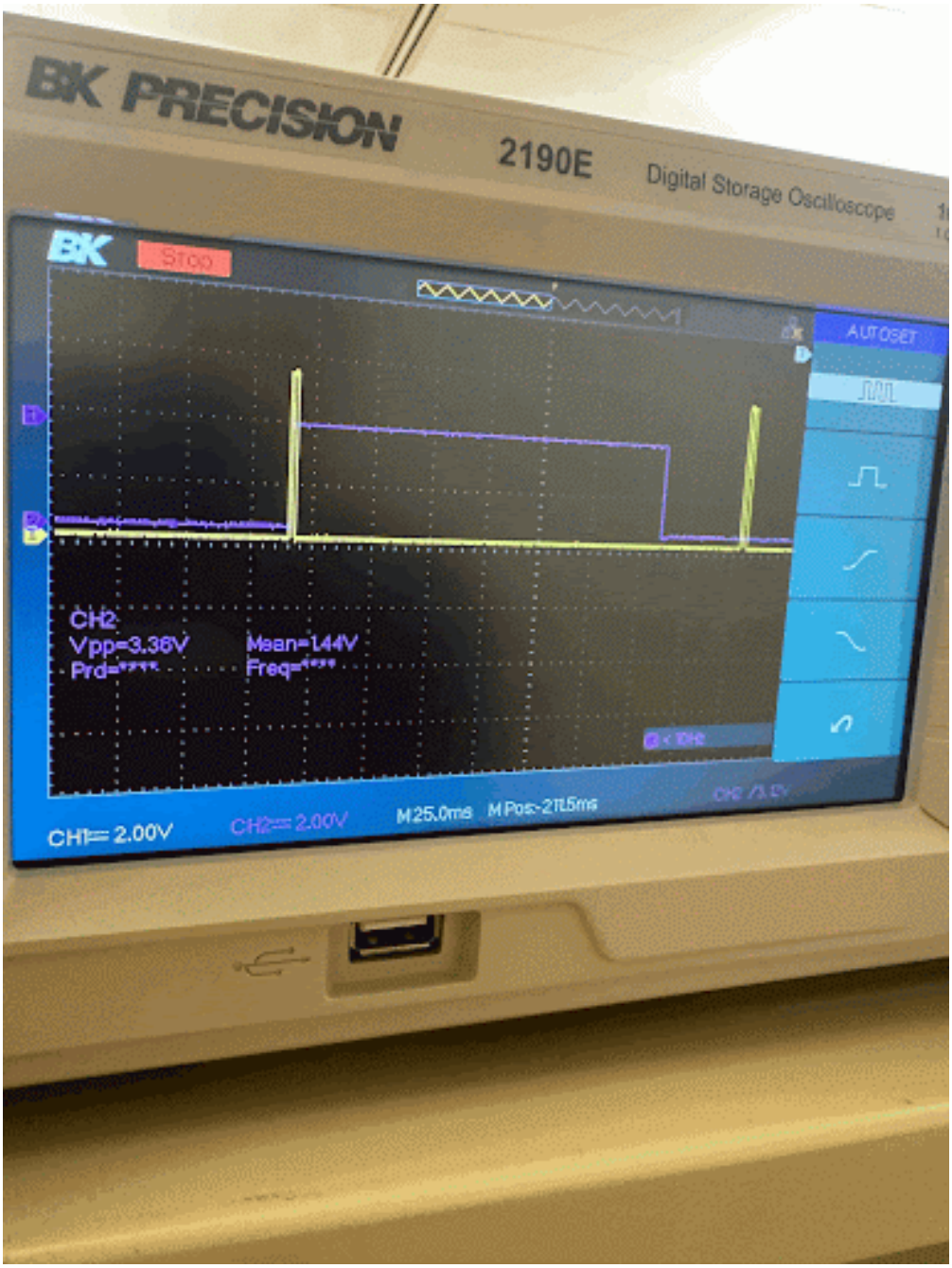

Figure 3

- **Figure 3:** Oscilloscope capture of SPI transmission. Yellow marks the beginning(rising edge) of the string, as purple marks the continuation of the transmitted string "Who Am I" over SPI

5.2 Measured Latency

- **End-to-End Latency:** 13 µs (latency of transmitting one string from K64F to Arduino. The end-to-end latency is the difference between the two pulses)

6 Conclusion

Conclusion

This project successfully demonstrated the use of UART, SPI, and I^2C communication protocols in an embedded system built around the K64F microcontroller, Arduino Uno, and FXOS8700CQ accelerometer/magnetometer sensor. Sensor data was transmitted to both a PC and an Arduino Uno, showing the system's ability to communicate across multiple platforms. We used an oscilloscope to verify that each communication channel was operating correctly and consistently.

One of the key results was measuring the end-to-end latency for SPI communication. We recorded a total delay of approximately 13 microseconds between initiating data transfer on the K64F and receiving the signal on the Arduino. With the SPI clock rate measured at 374.49 kHz, this results in a clock period of about 2.67 microseconds per bit. Given that several bytes were transmitted, the expected total time for the message falls within the range of 10–16 microseconds. Our measured value of 13 microseconds falls well within that range, confirming the validity of the result.

In summary, this project provided valuable experience in building and validating a multi-protocol embedded system. It reinforced practical skills in sensor integration, data communication, and signal analysis, all of which are essential for developing reliable embedded applications.

Kevin W. led the development of the software and SPI implementation. Micheal D. focused on hardware setup, oscilloscope configuration, and circuit validation.

Appendices

Appendix A: K64F main.c

```
/* ###################################################################
**     Filename    : main.c
**     Project     : Lab6_Part2
**     Processor   : MK64FN1M0VLL12
**     Version     : Driver 01.01
**     Compiler    : GNU C Compiler
**     Date/Time   : 2019-11-03, 17:50, # CodeGen: 0
**     Abstract    :
**         Main module.
**         This module contains user's application code.
**     Settings    :
**     Contents    :
```

```
**         No public methods
**
** ###################################################################*/
/*!
 ** @file main.c
 ** @version 01.01
 ** @brief
 **         Main module.
 **         This module contains user's application code.
 */
/*!
 **  @addtogroup main_module main module documentation
 **  @{
 */
/* MODULE main */


/* Including needed modules to compile this module/procedure */
#include "Cpu.h"
#include "MK64F12.h"
#include "Events.h"
#include "Pins1.h"
#include "FX1.h"
#include "GI2C1.h"
#include "WAIT1.h"
#include "CI2C1.h"
#include "CsIO1.h"
#include "IO1.h"
#include "MCUC1.h"
#include "SM1.h"
/* Including shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "PDD_Includes.h"
#include "Init_Config.h"
/* User includes (#include below this line is not maintained by Processor Expert) */

/*lint -save  -e970 Disable MISRA rule (6.3) checking. */

unsigned char write[512];
int main(void)
/*lint -restore Enable MISRA rule (6.3) checking. */
{
        /* Write your local variable definition here */

        /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
        PE_low_level_init();
        /*** End of Processor Expert internal initialization.                    ***/
        /* Write your code here */
        uint32_t delay;
        uint8_t ret, who;
        int8_t temp;
```

```
int16_t accX, accY, accZ;
int16_t magX, magY, magZ;

//config pin b
SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;
PORTB_GPCLR = 0x000C0100;
GPIOB_PDDR = 0xF;

int len;
LDD_TDeviceData *SM1_DeviceData;
SM1_DeviceData = SM1_Init(NULL);

printf("Hello\n");

FX1_Init();

for(;;) {
        // get WHO AM I values
        if (FX1_WhoAmI(&who)!=ERR_OK) {
                return ERR_FAILED;
        }
        printf("Who␣Am␣I␣value␣in␣decimal␣\t:␣%4d\n",who);
        len = sprintf(write, "Who␣Am␣I␣␣value␣in␣decimal␣\t:␣%4d\n",who);
        GPIOB_PDOR = 0x8;
        SM1_SendBlock(SM1_DeviceData, &write, len);
        for(delay = 0; delay < 300000; delay++); //delay
        GPIOB_PDOR = 0x0;
        // get raw temperature values
        if (FX1_GetTemperature(&temp)!=ERR_OK) {
                return ERR_FAILED;
        }
        printf("RAW␣Temperature␣value␣in␣decimal␣\t:␣%4d\n",temp);
        len = sprintf(write, "RAW␣Temperature␣value␣in␣decimal␣\t:␣%4d\n",temp);
        SM1_SendBlock(SM1_DeviceData, &write, len);
        for(delay = 0; delay < 300000; delay++); //delay

        // Set up registers for accelerometer and magnetometer values
        if (FX1_WriteReg8(FX1_CTRL_REG_1, 0x00) != ERR_OK) {
                return ERR_FAILED;
        }
        if (FX1_WriteReg8(FX1_M_CTRL_REG_1, 0x1F) != ERR_OK) {
                return ERR_FAILED;
        }
        if (FX1_WriteReg8(FX1_M_CTRL_REG_2, 0x20) != ERR_OK) {
                return ERR_FAILED;
        }
        if (FX1_WriteReg8(FX1_XYZ_DATA_CFG, 0x00) != ERR_OK) {
                return ERR_FAILED;
        }
        if (FX1_WriteReg8(FX1_CTRL_REG_1, 0x0D) != ERR_OK) {
                return ERR_FAILED;
        }

        // Get the X Y Z accelerometer values
```

```
                accX = FX1_GetX();
                accY = FX1_GetY();
                accZ = FX1_GetZ();
                printf("Accelerometer value \tX: %4d\t Y: %4d\t Z: %4d\n", accX, accY,
    accZ);
                len = sprintf(write, "Accelerometer value \tX: %4d\t Y: %4d\t Z: %4d\n",
    accX, accY, accZ);
            SM1_SendBlock(SM1_DeviceData, &write, len);
                for(delay = 0; delay < 300000; delay++); //delay

                // Get the X Y Z magnetometer values
                if (FX1_GetMagX(&magX)!=ERR_OK) {
                        return ERR_OK;
                }
                if (FX1_GetMagY(&magY)!=ERR_OK) {
                        return ERR_OK;
                }
                if (FX1_GetMagZ(&magZ)!=ERR_OK) {
                        return ERR_OK;
                }
                printf("Magnetometer value \tX: %4d\t Y: %4d\t Z: %4d\n", magX, magY,
    magZ);
                len = sprintf(write, "Magnetometer value \tX: %4d\t Y: %4d\t Z: %4d\n",
    magX, magY, magZ);
            SM1_SendBlock(SM1_DeviceData, &write, len);
                for(delay = 0; delay < 300000; delay++); //delay
            }


            /* For example: for(;;) { } */

            /*** Don't write any code pass this line, or it will be deleted during code
    generation. ***/
  /*** RTOS startup code. Macro PEX_RTOS_START is defined by the RTOS component. DON'T
    MODIFY THIS CODE!!! ***/
  #ifdef PEX_RTOS_START
    PEX_RTOS_START();                  /* Startup of the selected RTOS. Macro is defined
    by the RTOS component. */
  #endif
  /*** End of RTOS startup code.  ***/
  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/

/* END main */
/*!
 ** @}
 */
/*
 ** ###################################################################
 **
 **     This file was created by Processor Expert 10.4 [05.11]
 **     for the Freescale Kinetis series of microcontrollers.
```

```
 **
 ** ###################################################################
```

Listing 1: K64F main.c Code

Appendix B: Arduino SPI Receiver Code

```
#include <SPI.h>
char buff [255];
volatile byte indx;
volatile boolean process;

void setup (void) {
   Serial.begin (115200);
   pinMode(MISO, OUTPUT); // have to send on master in so it set as output
   pinMode(6, OUTPUT);
   SPCR |= _BV(SPE); // turn on SPI in slave mode
   indx = 0; // buffer empty
   process = false;
   SPI.attachInterrupt(); // turn on interrupt
}

ISR (SPI_STC_vect) // SPI interrupt routine
{
   byte c = SPDR; // read byte from SPI Data Register

   if (indx < sizeof(buff)) {
      buff[indx++] = c; // save data in the next index in the array buff
      if (c == '\n') {
        buff[indx - 1] = 0; // replace newline ('\n') with end of string (0)
        process = true;
      }
      digitalWrite(6, HIGH);
      delay(10);
      digitalWrite(6, LOW);
   }
}

void loop (void) {
   if (process) {
      process = false; //reset the process
      Serial.println (buff); //print the array on serial monitor
      indx= 0; //reset button to zero
   }
}‘‘‘
```

Listing 2: Arduino Uno main.c SPI code