LAB 5: SERIAL COMMUNICATION (SCI/SPI) AND C

OBJECTIVES

- Understand the basic operation and structure of asynchronous and synchronous serial communication, i.e., SCI (UART) and SPI.
- Utilize C programming constructs to create functions to interact with the XMEGA's USART and SPI systems.
- Learn how to use SPI communication to interact with an external IMU sensor package.
- Stream and plot real-time 3D (XYZ) acceleration data using the XMEGA's USART system and *Atmel Studio's Data Visualizer*.

REQUIRED MATERIALS

- µPAD v2.0 Development Board
- Robotics Backpack
- Digilent/NI Analog Discovery (DAD/NAD) kit
- *LSM330.h* header file
- Documentation
  - XMEGA_AU_Manual (doc8331)
  - ATxmega128A1U_Manual (doc8385)
  - Robotics Backpack Schematic
  - LSM330 Datasheet
  - Analog switch: SN74LVC1G3157
  - avr151_spiAVR151: SPI Application Note
  - Data Visualizer Software User's Guide
  - Data Visualizer GIF
  - data_stream_protocol.txt

YOU WILL NOT BE ALLOWED INTO YOUR LAB SECTION WITHOUT THE REQUIRED PRE-LAB.

Note: The amount of information given in this lab is much less than in earlier labs. Do **NOT** procrastinate!

INTRODUCTION

In this lab, you will reinforce your understanding of two types of important serial communication systems: Universal Asynchronous Receiver and Transmitter (UART, also known as SCI=Serial Communications Interface) and Serial Peripheral Interface (SPI). These are **NOT** the only asynchronous/synchronous communication protocols; some other popular serial standards are I^2C, USB, and CAN. The XMEGA also has a synchronous mode of the UART system [the S of USART].

The main difference between UART (SCI) and SPI is that UART is an **asynchronous** communication protocol, whereas SPI is a **synchronous** communication protocol. Systems that use synchronous communication utilize a common clock signal to determine when to send/receive/sample data; all synchronous communication between the systems is dependent on this signal. For devices communicating with an asynchronous protocol, there exists no synchronization signal. Instead, a common transfer rate must be upheld by the systems, or else, data received could either be wrongly interpreted, or entirely missed.

As discovered in Homework 4, the asynchronous UART protocol utilizes a transfer rate (denoted as the *baud rate*), as well as two physical connections for each device to communicate data, i.e., a pin for **R**eceiving data (Rx), and a pin for **T**ransmitting data (Tx). The only way that this protocol functions correctly is if both devices agree on a common baud rate.

SPI, on the other hand, utilizes a shared clock signal connected between (at least) two devices (usually denoted as SCK), where only one device is known as the "master" device, and where the other devices are "slaves." Full-duplex communication is possible with SPI (i.e., two devices can talk to each other at the same time), although the master always starts *any* communication between the devices. When the master wants to transmit/receive data to/from the slave(s), it generates the synchronized **S**erial **C**loc**k** signal (**SCK**). Upon each generated clock signal (a specific edge of the signal is the determining factor), a bit of data is shifted into (or out from) a shift register located within each device via two signals, one denoted as the **M**aster-**O**utput-**S**lave-**I**nput (**MOSI**) signal, and the other as the **M**aster-**I**nput-**S**lave-**O**utput (**MISO**) signal.

Synchronous communication is usually preferred over asynchronous communication because data transfer rates can be faster for synchronous communication. For asynchronous systems to uphold a decided data transfer rate *while maintaining low error*, it is necessary to include synchronization bits throughout the data transmission, which ultimately produces major overhead, and in turn, drastically reduces the actual data transfer rate.

Just as with UART (SCI), SPI can be done with timers and GPIO in with software by "bit-banging," although this is not allowed in our labs.

In this lab, you will first learn how to use the C programming language to configure your XMEGA's USART systems for asynchronous communication. Then, you will demonstrate your understanding of how to utilize USART with two simple C programs. (If you have not yet completed the previously assigned homework, do so **BEFORE** continuing with this lab.)

After this, you will learn how to configure and use your processor's SPI system, by setting up communication with an *LSM330 Inertial Measurement Unit* (IMU) chip, located on your *Robotics Backpack*, that contains a MEMS 3D digital accelerometer and a MEMS 3D digital gyroscope.

NOTE: An **accelerometer** is a device used to measure acceleration, e.g., static accelerations like gravity, or dynamic accelerations, such as vibrations or movements in the X, Y, or Z, coordinate planes. A **gyroscope** is a device that measures angular velocity and uses gravity to help determine orientation. You will **NOT** use a gyroscope in this lab nor during this semester of the course.

LAB 5: SERIAL COMMUNICATION (SCI/SPI) AND C

An important thing to note is that the LSM330 IMU is an external chip that needs to also be configured; a predefined method of communication for the device is as an SPI slave. (I^2C is also available with device, although we will not be using I^2C during any labs this semester.) You will therefore need to configure the IMU using SPI and also use SPI to retrieve the necessary sensor data ~~using SPI~~.

Finally, once SPI communication with the IMU is properly established, you will use your USART system to stream sensor data read from the IMU to Atmel Studio's *Data Visualizer* (via your PC's USB COM port), to generate real-time plots of linear acceleration force data.

PRELAB REQUIREMENTS

GETTING STARTED

NOTE: All software in this lab should be written in C. If you cannot get your solutions to function in C, you may write them in assembly language for ***partial*** credit.

NOTE: Although the C language has a multitude of built-in functions, you are **NOT** permitted to use any of them in EEL 3744C. For example, you are **NOT** allowed to use the `_delay_ms` or `_delay_us` functions. Also, do not use `sprintf`, `printf`, or any similar functions.

NOTE: Convert the 32 MHz clock configuration code that you previously used in assembly language to C. In other words, you **must** set the clock frequency to 32 MHz as done before, but now in C. Also, either keep your compiler's optimization at the default (which is –O1), or at *Optimize for size* (which is –OS). This is to assure that the assembly code generated for the setting of the clock speed will execute quickly enough (which is likely not to happen if optimization is turned off). An alternative is to use in-line assembly for setting the clock speed.

NOTE: You should **ALWAYS** create a flowchart or pseudo-code of a desired algorithm **BEFORE** writing any code. In order to help you practice this habit, the flowchart(s) or pseudo-code for this and all subsequent labs in our course are due **72 hours** prior to the start of your lab.

PART A – UTILIZING USART (SCI) WITH C

You have already been introduced to the XMEGA's USART system in a previously assigned homework. (If you have not yet completed this homework, do so **BEFORE** continuing with this lab.) In this part of the lab, you will rewrite the necessary USART assembly subroutines into C functions, and then test them by writing two simple C programs (`lab5a1.c`/`lab5a2.c`). **The baud rate required for this part of the lab is 57,600 Hz.**

The first program will just continuously output the ASCII character `U`. After verifying that your PuTTY terminal correctly receives the ASCII character `U`, you will use your DAD/NAD to verify your baud rate and to view the transmission frame used when sending the ASCII character `U`. Unfortunately, on the µPAD, there is no easy access to the XMEGA's PORTD transmit and receive pins. Therefore, you will need to configure another USART system.

NOTE: You will **NOT** be able to communicate with your computer's terminal with this other USART system, as it is not connected to the necessary FTDI chip.

The second C program written will be used to toggle on/off ~~each of your~~ the available ~~the~~ red LED within your RGB LED package, upon receiving a predefined string (which will be sent from your computer's terminal program), specifying which LED to turn on or off.

At the start of your program, all three LEDs within your on-board RGB package should be turned off, and your USART system should be configured to receive strings of input. **Upon your processor receiving the input string `red`, `RED`, `Red`**, etc. (i.e., all possible cases), the RED_PWM RGB LED should t**oggle on/off** (i.e., upon receiving `red`, if the RED_PWM RGB LED is currently off, the LED should toggle on, and if currently on, the LED should toggle off). Your program should be able to toggle the red RGB LED infinitely many times.

NOTE: The labels **RED_PWM**, **GREEN_PWM**, and **BLUE_PWM**, defined in the µPAD schematic, are just names! You should **NOT** use PWM for the purposes of this part of the lab.

There are multiple methods that can be used to receive a string of input, e.g., creating another useful USART C function, ***char* usart_d0_in_string(void),*** using USART receive-driven interrupts, etc. However, it is highly recommended that you develop a habit of trying to initially choose an efficient method to code, as your coding techniques will be graded on exams.

1. Write C functions to utilize the necessary USART system to transmit a single byte of data (e.g., ***void usart_d0_out_char(char c)***), to receive a single byte of data (e.g., ***char usart_d0_in_char(void)***), and to transmit a finite string of characters (e.g., ***void usart_d0_out_string(char * string)***). It would also help to create a function to perform any necessary initializations for the USART system (e.g., ***void usart_d0_init(void)***).

   NOTE: You may want to create a USART library (e.g., *usart.h*).

2. Write a C program (`lab5a1.c`) to continuously output the ASCII character `U`.

LAB 5: SERIAL COMMUNICATION (SCI/SPI) AND C

3. Modify your USART initializations (e.g., create another initialization function), to configure an appropriate USART system for which you have access to the transmit pin. Use the *Scope* feature in *Waveforms*, along with your DAD/NAD, to measure the width of both a single data bit and a single character transmission frame; verify that these waveforms match the **57,600 Hz** baud transmission rate, with one start bit, one parity bit, and one stop bit. Include **screenshots** in your prelab submission of both a single data bit as well as a single character transmission frame. Make sure to use the appropriate scaling in the *Scope* window. Label any necessary components of the screenshots.
4. Write a C program (`lab5a2.c`) to *toggle* on/off the red LED of the RGB LED package, as specified above.

NOTE: As mentioned before, make sure that you configure your XMEGA clock to operate at 32 MHz, now with a C function.

PART B: CONFIGURING SPI

In this part, you will set up an SPI module of the XMEGA to enable communication with the IMU located on your *Robotics Backpack*.

1. Carefully read sections 22.2, 22.3, 22.5 and 22.7 of the doc8331 manual, as well as the provided LSM330 datasheet. Some important sections to read in the LSM330 datasheet are Section 4, Section 6.2, Section 7, and Section 8. Some other important items of interest in the datasheet are Table 2 (focus on the SPI function of each pin), Table 6, and Table 9.

NOTE: Pay more attention to the accelerometer functionality as opposed to the gyroscope, as we will not be utilizing the gyroscope this semester.

For this lab, you will need to configure an SPI system within the XMEGA to interface with the LSM330 device. Therefore, you will need to have the necessary SPI module on the XMEGA fully functional before you can even begin to configure the IMU.

Here are some of the main things to consider when configuring a SPI module:

- What is the order of data transmission (MSB or LSB transmitted first)?
- Is the microprocessor the master device, or a slave device?
- What SPI transfer mode should be selected? (The transfer mode determines when the data is latched, and usually refers to the "phase" or the "polarity" of the clock signal.)
- What is the serial clock speed, or the speed at which data is transferred?

NOTE: The serial clock is denoted by SCK in Atmel documentation, and SPC in the LSM330 datasheet.

After reading (at least) all the documentation listed above (and after absorbing as much of it as possible), you will write several SPI functions in C. The first will initialize the necessary XMEGA SPI module to communicate with the LSM330. This function must also initialize the correct control signals and SPI signals used by the µPAD. Refer to the *Robotics Backpac*k schematic for specific ports and pins.

2. Determine which SPI system will connect to the LSM330 IMU, located on your *Robotics Backpack*.
3. Write a C function (e.g., ***void spi_init(void)***) to initialize the necessary SPI module within the XMEGA, as described above. Make sure that you select the bit transmission order (MSB or LSB) expected by the LSM330, and that you do **NOT** choose a SPI clock frequency that is too fast for the LSM330. **Also make sure that the master device does not compete for the SPI data bus (refer to Section 22.3 of the 8331 manual).**

NOTE: You may want to create a C library for any relevant SPI functions (e.g., *spi.h*)

Next, you will write C functions to transmit data as well as to receive data, via the XMEGA's SPI module.

4. Write a second C function (e.g., ***uint8_t spi_write(uint8_t data)***) to transmit a single byte of data from the master device (the XMEGA), and then wait for the SPI transmission to be complete. To wait for the transmission to be complete, you should poll a specific flag in the necessary SPI status register~~, and then once that flag is set, clear it~~. Interrupts should not be used for this purpose, as it prevents modular code. After the SPI transmission is complete, your function should return whatever new data is shifted into the SPI data register. (Remember that when you transmit data via SPI, data gets shifted out from the master and in from the slave at the **same time**.) Do **not** enable/disable any slave devices in this function, as multiple sequential calls to the function are possible, rendering multiple enables/disables unnecessary and inefficient.
5. Write a third C function (e.g., ***uint8_t spi_read(void)***), to read a single byte of data from a connected slave device. Once more, remember that with SPI communication, data is shifted in from a slave device at the **same time** that data is shifted out from the master device. To be able to read in a byte of data, you must send out a byte of data. Thus, to read a byte of data from a slave device, we can simply return the result of an SPI write transmission, e.g., ***return spi_write(0xFF)***. The byte of data that you choose to transmit in order to accomplish this task is arbitrary, i.e., the parameter of 0xFF shown above can be anything (0x37, for example). The reason for this is that when the master reads from the slave device, the

slave device will know to not save any data being received from the master during this time.

PART C: TRANSMITTING WITH SPI

In this part of the lab you will write a simple C program that will be used to verify the transmitting ability of your SPI module. You will transmit data to your DAD/NAD, and **NOT** yet to the LSM330. (The following method does not allow you to check read functionality, although a different, more intuitive technique will be used to test this in the following part of this lab.)

6. Write a simple C program **`(lab5c.c)`** to first initialize your SPI module, and then to continuously transmit **0x53**. Utilize any appropriate functions created in the previous steps. To also simulate a slave device being enabled/disabled, before each transmission, assert an **available** pin on the µPAD **low** (enable the "device"), and after each transmission, de-assert the same pin (set it to **high**, or disable the "device").

To verify that your SPI module is correctly transmitting **0x53**, you will view all necessary SPI signals with the *SPI* digital bus analyzer function of the *Logic* (LSA) program, within the DAD/NAD's *Waveforms* software (see the left image of Figure 1).

As shown in the right image of Figure 1, the SPI function can be used to view all three necessary SPI signals: ***Select*** (the signal used to enable a slave device, usually denoted as **C**hip **S**elect [**CS**], or **S**lave **S**elect [**SS**]), ***Clock*** (the clock signal used to synchronize any SPI connected devices, usually denoted as **S**erial **C**lo**c**k [**SCK**], or something similar, such as **S**erial **P**ort **C**lock [**SPC**] in the LSM330 datasheet), and ***Data*** (the signal used to transmit data to or receive data from a slave device, which is denoted by many things, e.g., **MOSI**/**MISO**, **S**lave **D**ata **I**n [**SDI**], or **S**lave **D**ata **O**ut [**SDO**]).

7. Within the *Logic* program of *Waveforms*, select *SPI* from the "*Click to Add channel*" dropdown menu (shown in Figure 1). You will be prompted with the *Add SPI* menu (also shown in Figure 1). Configure any necessary signals. The chosen DIO pin on the DAD/NAD for SPI data should connect to the necessary MOSI signal (to measure transmitted data). **Remember to refer to any of the given schematics, if necessary.**
8. Remove all backpacks from your µPAD. Use the program created above **`(lab5c.c)`**, along with the SPI digital bus analyzer function of your DAD/NAD to verify that your XMEGA is correctly transmitting **0x53**. You may want to use the SCK signal as a trigger source within the LSA.
9. Take a **screenshot** of your LSA window transmitting a full byte of data, including **all** necessary signals.

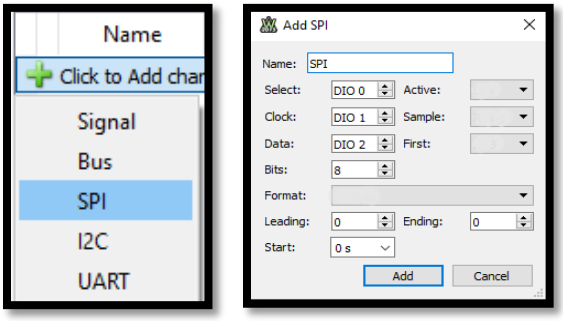

Figure 1: Adding an SPI bus within the *Logic* program (left), along with the *Add SPI* menu (right)

Include this screenshot in your report's appendix (as always).

PART D: RECEIVING WITH SPI & COMMUNICATING WITH THE LSM330

In the previous part, the DAD/NAD was used to measure highly precise synchronous signals with a digital bus analyzer function. To be able to transmit these same signals, you could utilize the protocol analyzer function (*Protocol*) within *Waveforms*. However, this will not be used for this lab.

NOTE: Although it is possible to measure the MISO signal with the technique used in the previous part of this lab, it would be cumbersome to do so with your very limited access to the necessary pins, as the *Robotics Backpack* must be plugged into the µPAD in order to communicate with the LSM330.

Instead, you will create a new C program **`(lab5d.c)`** to read from a predefined register within the external LSM330 IMU. Before reading from this predefined register, you will first create two functions (e.g., ***void accel_write(uint8_t reg_addr, uint8_t data)*** and ***uint8_t accel_read(uint8_t reg_addr)***) that allow you to communicate with *any* of the accelerometer registers within the LSM330.

NOTE: This part requires that you **understand** the LSM330 datasheet.

The LSM330 has a plethora of configuration registers, just like the XMEGA, other microprocessors, and other peripherals (e.g., an external UART, an external DAC, an LCD, etc.). The main difference with accessing a register within your XMEGA and accessing a register within the LSM330 is that you must use SPI (or I^2C) to write to or read from a register within the LSM330. However, another important difference is that the LSM330 contains two individual entities that must separately be controlled, namely the built-in accelerometer and built-in gyroscope. This means that, in general, when configuring the accelerometer, the accelerometer must be independently

LAB 5: SERIAL COMMUNICATION (SCI/SPI) AND C

enabled from the gyroscope. Again, you will **not** be using the gyroscope this semester.

NOTE: Any LSM330 register name ending with "_A" indicates that the register modifies/inspects the built-in accelerometer. Register names ending in "_G" reference registers that modify/inspect the built-in gyroscope.

Separately, as mentioned before, the LSM is capable of both SPI and I^2C communication. To simplify circuitry on your µPAD, as well as PCB design, certain signals were designed to have the ability to be used for both the SPI and I^2C connections on the LSM330, rather than have two sets of signals (since both communication protocols use the same signals, albeit in a different manner). To accomplish this design strategy, the necessary signals are multiplexed with a digital switch (i.e., an analog, bidirectional, 2-input multiplexer), located on the *Robotics Backpack* (see the digital switches on page 3 of the *Robotics Backpack* schematic). In other words, there are additional signals located within your *Robotics Backpack* that must be directly controlled by your processor (with I/O port assignments, just like you have done in previous labs), before attempting to configure the LSM330.

1. Determine what signals must be controlled by your XMEGA, and why they must be controlled. Furthermore, determine where these predefined signals connect to on the XMEGA, and how to control them in software. Refer to the *Robotics Backpack* schematic, as well as the LSM330 datasheet.

Before writing code to configure or utilize the LSM330, it is highly recommended that you download and use (with the ***#include*** C preprocessor directive) the given ***LSM330.h*** header file available on the course website. This header file contains useful definitions for register addresses within the LSM330 (see Figure 2), similar to the XMEGA's include file. The intention is for you to refer to easily identifiable names, rather than "magic numbers" (unique values with unexplained meaning). Recall that you can define your own set of constants, or condense certain aspects of C code, with macros (by using the ***#define*** preprocessor directive).

Following this, to create the necessary C functions to write to or read from *any* accelerometer register within the LSM330, here are the order of events that need to occur:

- Enable the appropriate device with the necessary signal(s). (Remember to refer to how the *Robotics Backpack* is designed.)
- Send the appropriate amount of data to the LSM330, based on the timing diagram given in the LSM330 datasheet.
- Disable the device with the appropriate signal(s).

2. Create C functions to be able to write to **any** of the available accelerometer registers within the LSM330 (e.g., ***void accel_write(uint8_t reg_addr, uint8_t data)***) and to be able to read from **any** of the available accelerometer registers (e.g., ***void accel_read(uint8_t reg_addr)***). Be sure to utilize any previously created functions, when appropriate.

NOTE: You may want to add *declarations* for the above functions within the given ***LSM330.h*** header file, as well as create a source file (e.g., ***LSM330.c***) to contain any *definitions* for these functions.

Now, we will verify receive functionality of our SPI system. Within the LSM330, there exist two predefined registers to identify both the built-in accelerometer device ID, as well as the built-in gyroscope device ID. The former, denoted by **`WHO_AM_I_A`** (refer to Section 8.1 in the LSM330 datasheet), returns the default value of **0x40**. Your program **`(lab5d.c)`** must verify that this value is received upon reading from the **`WHO_AM_I_A`** register.

3. Create a C program **`(lab5d.c)`** to read the **`WHO_AM_I_A`** register, as described above. Use any technique to verify that the expected value is received, e.g., using a *Watch* window within Atmel Studio, etc.

PART E: UTILIZING THE ACCELEROMETER

Now that you have a set of functions that allow you to configure the LSM330's accelerometer registers, you need to actually configure the accelerometer!

NOTE: Before using any system, it is always recommended to first perform a software reset, if possible. To perform a reset of the LSM330 accelerometer in software, you can set a certain bit within the **`CTRL_REG4_A`** register.

Before initializing the accelerometer, we must identify necessary components of the device, and we should also identify useful components. Below are a few helpful comments, as well as questions to pose to yourself:

- The LSM330 is designed to **be able to** generate an interrupt signal upon the accelerometer completing an acceleration force measurement, signaling that data is ready to be read from the accelerometer. In this lab, this

```
CTRL_REG4_A     =   0x23,
CTRL_REG5_A     =   0x20,
CTRL_REG6_A     =   0x24,
CTRL_REG7_A     =   0x25,
STATUS_REG_A    =   0x27,
```

Figure 2: Code segment given in *LSM330.h*

LAB 5: SERIAL COMMUNICATION (SCI/SPI) AND C

interrupt signal **must** be used to generate an external interrupt on your microprocessor, instead of wasting time by continuously waiting for the accelerometer to perform a measurement. To utilize this interrupt signal, you must route an internal accelerometer signal to an available LSM330 interrupt pin (see **CTRL_REG4_A**, as well as the necessary schematics). What edge-level should be configured for this external interrupt?

- The accelerometer within the LSM330 has the ability to measure acceleration forces in the X, Y, and Z coordinate planes. In this lab, you will use the accelerometer to measure all three dimensions simultaneously. How will you enable the accelerometer to do so? At what rate will measurements be taken? (See **CTRL_REG5_A**.)

Overall, for this lab, you must do the following when initializing the LSM330 accelerometer:

i. Configure an external interrupt within your XMEGA to trigger upon the accelerometer completing a measurement.
ii. Configure CTRL_REG4_A and CTRL_REG5_A within the LSM330.

1. Create a C function (e.g., ***void accel_init(void)***) to initialize the LSM330's accelerometer, as described above.

In the next part of this lab, we will begin to utilize measurements being made by the accelerometer. To do so, you must first understand how to access accelerometer data for each of the three coordinate planes, X, Y, and Z.

2. Determine how to access accelerometer data from the LSM330. Refer to the LSM330 datasheet.

PART F: REAL-TIME DATA PLOTTING

In this part of the lab, we will create a C program **(lab5f.c)** to plot accelerometer data for each of the coordinate planes (X, Y, and Z) in real-time, with the *Data Visualizer* extension program of Atmel Studio.

Because an apparent bug in the Data Visualizer, when communicating with the Data Visualizer, configure the necessary USART system to use NO parity.

As stated in the *Data Visualizer Software User's Guide*, "the *Data Visualizer* is a program to process and visualize data." There are multiple ways to communicate with the *Data Visualizer*, one of which is through a serial COM port (read: UART). In other words, there exist built-in terminal programs within the *Data Visualizer* that allow external devices to transmit data to the *Data Visualizer*. The data received by a *Data Visualizer* terminal can be used to plot a waveform with a graphing utility also built into the *Data Visualizer*. However, not all data received should be expected to be plotted within a graphing utility. Thus, to be able to only plot data received *when desired*, the *Data Visualizer* requires that another communication protocol be upheld on top of UART (or any other allowable) communication.

There are two communication protocols that the *Data Visualizer* can interpret: Data Stream Protocol and **A**tmel **D**ata **P**rotocol (**ADP**). Data Stream Protocol is user-defined and more flexible than ADP, although there are still certain criteria that must be met. Data Stream Protocol must be used for this lab.

So, for this part of the lab, to simply communicate with the *Data Visualizer*, UART must be used to allow communication between your computer and your XMEGA, via the USB connection on your µPAD. **(The required baud rate for this part of the lab is 1 MHz.)** Then, to plot accelerometer data from your IMU to the graphing utility within the *Data Visualizer*, Data Stream Protocol must be used.

An example Data Stream Protocol transmission is shown in Figure 3. There are a few things to note about this figure, and about the Data Stream Protocol in general:

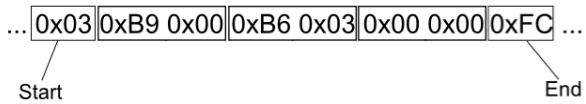

Figure 3: Example Data Stream Protocol

- The data located in between the **required** "start" and "end" bytes (which will be used to synchronize the data streamer) are just arbitrary example values. The start/end bytes are also arbitrary, although they must be inverses of each other (and the start byte must specifically **NOT** be 0x5F).
- According to Section 5.1.2 of the above documentation (*Data Visualizer Software User's Guide*), "all data must be given as little-endian values, meaning that the lowest byte must be sent first."

1. Refer to the *Data Visualizer Software User's Guide* for any more information on the Data Stream Protocol.

To define the Data Stream Protocol, a data stream configuration text file must be used.

2. Download the ***data_stream_protocol.txt*** file from the course website. This protocol defines a Data Stream Protocol for transmitting accelerometer data for all

three coordinate planes (X, Y, and Z). The contents of this file are shown in Figure 4.

```
-D,1,1,ACCEL_X
-D,2,1,ACCEL_Y
-D,3,1,ACCEL_Z
```

Figure 4: Example Data Stream Protocol configuration file, *data_stream_protocol.txt*

Each row (or line) of a data stream configuration file defines a specific stream of data to be interpreted by the *Data Visualizer*, i.e., a waveform to be plotted within the graphing utility of the *Data Visualizer*. Be aware that the set of data streams specified within a Data Stream Protocol configuration file defines the set of data that **must** be sent to the *Data Visualizer* within a **single** transmission, e.g., in the above configuration file, the data corresponding to ACCEL_X, ACCEL_Y, and ACCEL_Z must be sent sequentially, within a single Data Stream Protocol transmission.

NOTE: When transmitting data to the *Data Visualizer*, you **MUST** transmit the data in the **same order** as it appears in this configuration file. In other words, if using the given configuration file within the context of this lab, X-axis accelerometer data should be transmitted first, then Y-axis data, and then finally, Z-axis data.

The first column of each data stream determines the type of data used for that particular stream. Some of the available types are shown in Figure 5. The second and third columns of each data stream are used to define the visual organization of the data streams within the *Data Stream Control Panel* tab of the *Data Visualizer* (shown in Figure 6). Lastly, the final column simply defines a name for the data stream, which is also referenced within the *Data Stream Control Panel*.

Type	Size	Tag
Unsigned byte	1	B
Signed byte	1	-B
Unsigned short	2	D
Signed short	2	-D
Unsigned word	4	W
Signed word	4	-W

Figure 5: Several of the available data stream data types

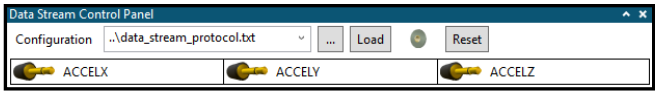

Figure 6: Data streamer control panel upon loading *data_stream_protocol.txt*

3. Write a C function (e.g., within your **`lab5f.c`** program) to transmit a stream of sensor data via your USB Serial Port (USARTD0), with the Data Stream Protocol defined in *data_stream_protocol.txt*. Make sure that your function transmits the correct number of bytes. Your function should also utilize other functions (or macros) that you have already created, such as ***usartd0_out_char()*** and ***usartd0_out_string()***. Again, **the required baud rate for this part of the lab is 1 MHz**.

Now, recall that the LSM330 will be configured to interrupt your XMEGA, upon completing an acceleration force measurement. Your program **`(lab5f.c)`** must **only** communicate with the *Data Visualizer* when **new** data from the accelerometer becomes available, since we only care to plot new data within the *Data Visualizer* graphing utility.

Moreover, as always, your program should spend as little time as possible within the respective interrupt service routine. In other words, your program must **NOT** communicate with the *Data Visualizer* within an ISR. Instead, a global flag (e.g., ***volatile uint8_t accel_flag***) should alert your main program that new accelerometer data is ready to be output.

4. Create a program **`(lab5f.c)`** to plot accelerometer data for all three coordinate planes (X, Y, Z) by communicating with the *Data Streamer*, **only when new accelerometer data becomes available**, as described above.

To plot the waveforms with the *Data Visualizer* extension program, you must first open the *Data Visualizer* by navigating to **Tools → Data Visualizer**, within Atmel Studio.

Next, a single *Graph* window must be added by navigating within the *Data Visualizer* to **Configuration → Visualization → Graph**, located on the left side of the *Data Visualizer* window. This is the graphing utility that will be used to plot accelerometer data.

Following this, you will need to open a *Data Stream Control Panel* within the Data Visualizer. This can be done from **Configuration → Protocols → Data Streamer.** A Data Stream Protocol configuration file must be loaded by clicking the **"…"** button within the *Data Stream Control Panel*. After loading the given configuration file, you should have three available data streams that can be plotted with the graphing utility. The *Data Stream Control Panel* should look like what is shown in Figure 6.

Finally, open a *Serial Port Control Panel* by navigating to **Configuration → External Connection → Serial Port**. This is what will open the serial terminal used to provide accelerometer data from the µPAD to the *Data Visualizer*.

LAB 5: SERIAL COMMUNICATION (SCI/SPI) AND C

To see how to connect these components together within the *Data Visualizer*, see the Data Visualizer GIF on our website. For the purposes of this lab, the source of all data from which to plot within the *Data Visualizer* is the serial port on your XMEGA. To connect this serial port to the *Data Visualizer*, drag the plug from the *Serial Port Control Panel* to the jack within the *Data Stream Control Panel*. This is to make the serial stream of accelerometer data act as the source to the data streamer. Then, to get the data to be displayed within the *Graph*, you must connect each of the data stream plugs within the *Data Stream Control Panel* to the ***new plot*** jack of the *Graph*. Start by connecting the "ACCELX" plug, followed by "ACCELY", and finally "ACCELZ".

After all of this, your graphing utility should plot three waveforms representing the data for each of the accelerometer coordinate planes (X, Y, and Z), in real-time!

NOTE: Remember that Earth's gravitational force vector (perpendicular to the ground) will continuously act on the accelerometer axes that are partially aligned with the gravity vector.

5. Use a graphing utility within the *Data Visualizer* to plot all three axes of accelerometer data, in real-time. **Take a screenshot** of your graph, including all three waveforms. Include this screenshot in your prelab report's appendix.

Because an apparent bug in the Data Visualizer, when communicating with the Data Visualizer, configure the necessary USART system to use NO parity.

PRE-LAB QUESTIONS

1. What are some examples of useful macro ***functions***, in the context of this lab?
2. Why do we use UART to communicate with the *Atmel Studio Data Visualizer* extension program?
3. What is the highest speed of communication that the IMU can handle?
4. Why is it a better idea to modify global flag variables inside of ISRs instead of doing everything inside of them?
5. Why is the "-D" tag used within the Data Stream Protocol defined in ***data_stream_protocol.txt***?
6. What is the most positive value that can be received from the accelerometer (in decimal)? What about the most negative?

PRE-LAB PROCEDURE SUMMARY

1. Answer all pre-lab questions.
2. In Part A, create two C programs (**`lab5a1.c/lab5a2.c`**) to demonstrate your understanding of USART. Use your DAD/NAD to verify the baud rate generated, as well as to view a character transmission frame. Take a screenshot of both of these measurements.
3. In Part B, create C functions to configure/utilize the SPI module that will connect to the LSM330 IMU.
4. In Parts C and D, test the SPI functions that you wrote in Part B (with programs **`lab5c.c`** and **`lab5d.c`**, respectively).
5. In Part D, create functions to communicate with the LSM330 IMU. Make sure to utilize the SPI functions created in Part B.
6. In Part E, create a C function that initializes the built-in accelerometer, within the LSM330.
7. In Part F, create a C program **`(lab5f.c)`** to plot accelerometer data for all three coordinate planes (X, Y, and Z), within the built-in graphing utility of the *Data Visualizer*.

IN-LAB PROCEDURE

1. Demonstrate your USART program **`(lab5a2.c)`** from Part A executing on your board.
2. Demonstrate IMU accelerometer data being plotted within the graphing utility of Atmel Studio's *Data Visualizer*, as required in Part G.
3. Answer any questions that your TA may ask you about your hardware or software.

NOTE: If you cannot get the *Data Visualizer* graphing utility to function properly, output accelerometer data to a terminal window for partial credit. If Part G does not work at all, be prepared to demonstrate earlier parts of the lab.

REMINDER OF LAB POLICY

Please re-read the *Lab Rules & Policies* so that you are sure of the deliverables that are due prior to the start of your lab. You must adhere to the *Lab Rules and Policies* document for **every** lab.