OBJECTIVES

- Understand general concepts regarding synchronous serial communication.
- Become familiar with the *Serial Peripheral Interface* (*SPI*) communication protocol and how to apply it with the *ATxmega128A1U*.
- Interface with an external *inertial measurement unit* (*IMU*) sensor package, by way of the SPI communication protocol.
- Stream and plot real-time 3D (XYZ) acceleration data using the *SPI* and *USART* systems of the *ATxmega128A1U*.

INTRODUCTION

Note that the manuals use a different word for ***student***, as in *master/**student***. I have chosen to use ***student*** instead throughout this document, but have no control over the terminology used in the manuals.

In general, systems that use *synchronous serial communication* utilize a common clock signal to determine when to send, receive, or sample data. In fact, all synchronous communication is dependent on such a signal. In contrast, for devices communicating with an asynchronous serial communication protocol, there exists no synchronization signal. Instead, a common transfer rate must be upheld by the systems, or else, data received could be interpreted incorrectly, or even entirely missed.

Synchronous serial communication is normally preferred over asynchronous serial communication because higher data transfer rates can generally be achieved. The main reason for this is that asynchronous serial systems must normally include within all data transmissions some bits for synchronization and "error-checking", which effectively reduces any possible rate of *data transfer*.

One popular synchronous serial communication protocol is *Serial Peripheral Interface* (*SPI*). With SPI, full-duplex communication is possible between two or more devices, although only one device may act as a controlling *master* – every other connected device must act as a responding *student*. Although there may be only one master, it is possible that the *role of master* be assigned dynamically. However, for this to be done properly, some form of "handshaking" is required.

When communication is to occur between some master and student(s), *the master device is always responsible for initiating the communication*. Normally, before some communication may occur, a student must be enabled by way of a *chip select* (*CS*), or student *select* (*SS*), signal. In general, such a signal is either controlled by the master or is always driven *true*. To start communication once some set of students is enabled, the master generates a clock signal, often referred to as the *serial clock* (*SCK, SPC*). Upon some edge of each generated clock pulse, either the rising edge or falling edge, each of the master and student(s) will shift out a single bit of data from an internal shift register, where each has their own. Furthermore, each device is to "sample" some bit of data on the opposite edge of the same clock pulse. Any data shifted into a student from some master is transmitted via a signal most commonly referred to as *Master-Output-Student-Input* (*MOSI*), and any data shifted into a master from some student is transmitted via a signal most commonly referred to as *Master-Input-Student-Output* (*MISO*). In general, either or both of some MOSI/MISO signals can be used; in other words, it is possible for [1] a master to transmit to, but not receive from, some student(s), [2] a master to receive from, but not transmit to, some student(s), and [3] a master to both transmit to and receive from some student(s). For both [2] and [3], it must be ensured that no two student devices share a chip select signal, for else there would be bus contention.

Although SPI is a very common synchronous serial communication protocol, there exist many others; some others include *Inter-Integrated Circuit* (*IIC*, or more commonly, *I^2C*), *Universal Serial Bus* (*USB*), and *Controller Area Network* (*CAN*). Moreover, the synchronous mode within the USART system of the *ATxmega128A1U*, the 'S' of "USART", provides access to another synchronous serial communication protocol, which is very similar to SPI.

LAB STRUCTURE

In this lab, you will utilize synchronous serial communication by way of the SPI protocol. More specifically, you will learn how to configure and use the *SPI* system of the *ATxmega128A1U*, for the purpose of communicating with an *LSMDS3TR* or *LSM6DSL inertial measurement unit* (*IMU*) chip, located on the *OOTB Robotics Backpack*. The *LSM6* devices contains a Micro-Electro-Mechanical System (MEMS) 3D digital accelerometer and a MEMS 3D digital gyroscope[1]. Once SPI communication with the IMU is properly established, you will generate real-time plots of linear acceleration on your computer, by streaming IMU sensor data to a data visualization program, *SerialPlot*, via the relevant USART module.

REQUIRED MATERIALS

- Atmel *ATxmega128A1U* AU Manual (doc8331)
- *LSM6DSL Datasheet* (most of you will have this)
- *LSM6DS3TR DataSheet*
- Relevant skeleton code files (.c and .h): lab6_files.zip
- *OOTB µPAD*, with USB A/B cable
- *OOTB Robotics Backpack*, with accompanying schematic
- **D**igital **A**nalog **D**iscovery (**DAD**) kit, with *WaveForms*

[1] 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 axes. A *gyroscope* is a device that measures angular velocity and uses gravity to help determine orientation.

SUPPLEMENTAL MATERIALS

- Atmel *ATxmega128A1U* Manual (doc8385)
- *Setup and Use of the SPI* (doc2585)
- *SerialPlot* source website
  - *SerialPlot* for 64- and 32-bit Windows

PRE-LAB PROCEDURE

REMINDER OF LAB POLICY

You must re-read the *Lab Rules and Policies* before submitting any pre-lab assignment and before attending any lab.

1. INTRODUCTION TO SPI AND THE *LSM6 DEVICES*

In this section, you will begin to become familiar with the SPI protocol, the SPI system available within the *ATxmega128A1U*, and the *LSM6DSL* inertial measurement unit (IMU). (A few of you have a *LSMDS3TR* IMU; in all of our uses except one, they will be identical. The single exception will be discussed in the first green part of section 3. All notes about the *LSMDS3TR* are in green.) When proceeding, keep the following in mind:

- The SPI serial communication protocol, like any other digital communication protocol, is just a predefined system of rules for performing data transfer between multiple components via a set of digital signals.
- The *SPI* system within the *ATxmega128A1U* was designed to simplify the procedure of emulating the SPI protocol.
- From the perspective of the microcontroller, the IMU is an external peripheral.

1.1. Carefully read § 22 (*SPI – Serial Peripheral Interface*) of the 8331 manual.

Generally, for design simplicity, many devices (especially peripherals) are built to only support a subset of possible SPI configurations. Fortunately, the SPI system within the *ATxmega128A1U* was built to support most of the possible SPI configurations, and thus, is able to communicate with many types of components. However, because of this flexibility, there are several considerations to make when utilizing this SPI system (and often when using the SPI protocol in general):

- Which device(s) should be given the role of master and which device(s) should be given the role of student?
- How will the student device(s) be enabled? If a student select is utilized, rather than just have the device(s) be permanently enabled, which pin(s) will be used?
- What is the order of data transmission? Is the MSb or LSb transmitted first?
- In regard to the relevant clock signal, should data be latched on a rising edge or on a falling edge?
- What is the maximum serial clock frequency that can be utilized by the relevant devices?

Throughout this lab, you will utilize the SPI protocol to communicate with an IMU peripheral chip located on the *OOTB Robotics Backpack*, for the purposes of interfacing with a MEMS 3D digital accelerometer and, optionally, a MEMS 3D digital gyroscope.

Like with the *ATxmega128A1U*, the *LSM6* devices utilize memory-mapped registers to store data relevant to its internal components. However, instead of utilizing a parallel address bus and parallel data bus to access these memory-mapped registers, like the *ATxmega128A1U*, the IMU utilizes the SPI (or I^2C) protocol.

It is important to recognize that the IMU component located on the *OOTB Robotics Backpack* was designed to utilize specific I/O pins on the *ATxmega128A1U*. Additionally, to allow the potential for both SPI and I^2C communication, certain signals were designed to be multiplexed with a digital switch (more specifically, an analog, bidirectional, 2-input multiplexer) on the *OOTB Robotics Backpack*. Thus, there are additional signals within the *OOTB Robotics Backpack* that must be directly controlled by the microcontroller (with I/O port assignments, just like in previous labs), before attempting to configure the *LSM6* device.

Below, you will begin to become familiar with the *LSM6* component and understand how it was designed to connect to your microcontroller.

1.2. Read through the *LSM6DSL* Datasheet (or the *LSM6DS3TR* Datasheet). Pay extra attention to §§ 2, 4, 6.4 (or 6.2), 7, and 8, as well as to Table 2 (focus on the SPI function of each pin), Table 6, and Table 10 (or Table 9).

1.3. Determine which signals from the *ATxmega128A1U* will be utilized to communicate with the *LSM6DSL* IMU (or the *LSMDS3TR* IMU) chip on the *OOTB Robotics Backpack*. Refer to the appropriate schematic(s) and manual(s).

PRE-LAB EXERCISES

i. In regard to SPI communication that is to exist between the relevant *ATxmega128A1U* and IMU chips, answer each of the questions within the previously given bulleted list.

2. COMMUNICATING WITH SPI

Throughout the next few sections, you will incrementally create several routines with the "C" programming language, for the purpose of communicating with the IMU device. Additionally, you will perform experiments when appropriate, to verify your work.

First and foremost, you will design "C" functions to utilize the SPI system within the *ATxmega128A1U*. The functions that you will create should allow for a somewhat generic use of the SPI system and should follow the provided ***spi.h*** and ***spi.c*** files.

2.1. Within the provided ***spi.c*** file, complete the "C" function, ***void spi_init(void)***, to initialize the appropriate SPI module within the *ATxmega128A1U* as well as the appropriate control signals on the *µPAD*, for the purpose of communicating with the relevant *IMU*.

2.1.1. Make sure that you select the bit transmission order (MSb or LSb) expected by the IMU, and that you do not choose a SPI clock frequency that is too fast for the IMU. For more details, refer to § 22.3 (*Master Mode*) of the 8331 manual, § 33.2 (*Alternate Pin Functions*) of the 8385 manual, and any relevant schematics.

2.2. Write a second "C" function, ***void spi_write(uint8_t data)***, to transmit a single byte of data from the master device (the *ATxmega128A1U*), and then wait for the SPI transmission to be complete.

2.2.1. To wait for the transmission to be complete, you should poll a specific flag in the relevant SPI status register. Interrupts should not be used for this purpose, as it inhibits portable code. (Note that more advanced programming techniques, which are outside the scope of this course, could be used to circumvent such a portability issue.) After the SPI transmission is complete, your function should terminate.

IMPORTANT NOTE: Do not enable or disable any student devices within this function, as multiple sequential calls to the function are possible, rendering the use of multiple enables/disables inefficient. Such enabling and disabling should be handled within a separate routine, which is discussed later on.

2.3. Write a third "C" function, ***uint8_t spi_read(void)***, to read a single byte of data from a connected student device.

2.3.1. This function should write some arbitrary byte of data to the SPI data register, to trigger an exchange of data between your microcontroller and the IMU, and then after the transmission is complete, have the function return the contents of the data register. Remember that, with SPI communication, data is shifted in from a student 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. The byte of data that you choose to transmit in order to accomplish this task is arbitrary, e.g., it could be 0xFF, 0x37, etc. (The student device will know to not save any data being received from the master during this time.) Just as with the ***spi_write*** routine above, do *not* enable or disable any student devices within this function.

Now, you will create a "C" program and utilize your DAD to verify *transmit* functionality for the relevant SPI module.

NOTE: The following method could also be used to verify *receive* functionality, although a different, more intuitive technique will be used to test such functionality, in the following section of this lab.

2.4. Write a simple "C" program, **`lab6_2.c`**, to initialize the relevant SPI module and to continually transmit **0xCE** (decimal **206**) from this same module. Utilize the relevant "C" functions previously written by yourself. Also, to simulate the IMU student device being enabled/disabled, assert an available I/O port pin *low* before each transmission (i.e., simulate the process of enabling the device), and de-assert the same pin after each transmission (i.e., set the relevant pin *high*, or simulate the process of disabling the device).

To verify that the SPI module is correctly transmitting **0xCE**, you will view all appropriate SPI signals with the *SPI* digital bus analyzer function of the *Logic* (LSA) feature within *Waveforms*. See the left image of Figure 1 to determine where this feature may be accessed from within the *Logic* feature.

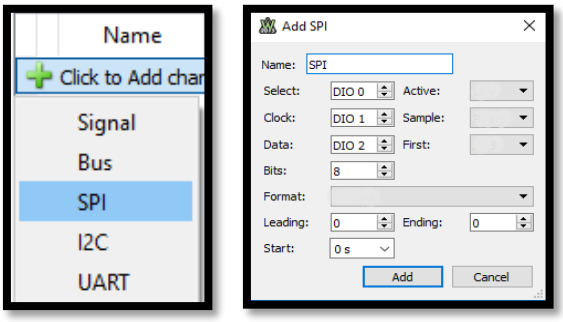

Figure 1. (left) Accessing the SPI function within the *Logic* feature; (right) the *Add SPI* menu.

As shown in the right image of Figure 1, the SPI function can be used to view the three relevant SPI signals: ***Select*** (the signal used to enable a student device, usually denoted as *chip select* [*CS*] or *student select* [*SS*]), ***Clock*** (the clock signal used to synchronize any SPI connected devices, usually denoted as *serial clock* [*SCK*], or something similar, such as *serial port clock* [*SPC*] in the IMU datasheet), and ***Data*** (the signal used to transmit data to or receive data from a student device, which can represented by several common names, e.g., *Master-Out-Student-In* [*MOSI*], *Master-In-Student-Out* [*MISO*], *Student Data In* [*SDI*], and *Student Data Out* [*SDO*]).

2.5. Within the *Logic* feature of *Waveforms*, select *SPI* from the "*Click to Add channel*" dropdown menu, depicted in the left image of Figure 1. You will be prompted with the *Add SPI* menu, an example of which is depicted in the right image of Figure 1. Configure the relevant settings. Refer to the relevant schematics, if necessary.

2.6. Use the program created above, **`lab6_2.c`**, along with the SPI digital bus analyzer function of your DAD, to verify that your microcontroller is correctly transmitting **0xCE**. To be able to probe the relevant pins, you may either [1] remove any backpack connected to the *µPAD* or [2] utilize the *J2* header of the *OOTB Robotics Backpack*. The chosen DIO pin on the DAD for SPI data should connect to the appropriate MOSI signal. You may wish to use the

student select signal as a falling-edge trigger source within the *Logic* feature. You can also use a "protocol trigger" to trigger the LSA when an SPI transmission starts.

2.7. Take a screenshot of the LSA measuring a single, full byte of data being transmitted. Include all relevant signals. Make sure that you choose an appropriate time base such that a single transmission is clearly visible.

2.8. Students should complete up to this point by **Monday, March 25th** and submit a pre-lab report (using the regular lab submission template) to the Lab 6 assignment for all work through this part of the lab. The final submission will require all of the work done for this lab including portions submitted at the first due date. The penalty for submitting the first portion of the lab late is **20%** (i.e., **20** out of the 100 points available in Lab 6).

3. COMMUNICATING WITH THE IMU

In this section, you will verify *receive* functionality by interfacing with the external IMU. To accomplish this, you will create a new "C" program, **`lab6_3.c`**, to read from an accelerometer register within the IMU. However, before writing this program, you will first design two "C" functions to allow you to read from or write to *any* register within the IMU.

Important Note: Most, if not all of you will have the *LSM6DSL* IMU on your robotics backpack, however, some of you might have the *LSMDS3TR* IMU instead. There are 2 differences. First, is that the *LSM6DSL* IMU has more available features (none applicable to this lab) and therefore some of the configuration registers will look slightly different (but will be located at the same address as its counterpart on the *LSMDS3TR*). Second, and most important is that when you read the WHO_AM_I register on the *LSMDSL* you will receive a value of 0x6A and when you read the register on the *LSMDS3TR* IMU you will receive a value of 0x69. Note that this is how you will determine which device you have and which manual you should reference when configuring it. Please let me know if you (by email) if you have an *LSMDS3TR IMU.*

Overall, the IMU has a plethora of configuration registers for its internal accelerometer (and gyroscope), just like our microcontroller does for its internal peripherals. However, unlike when accessing a register within the *ATxmega128A1U*, where parallel busses are utilized to communicate address and data values, SPI (or I^2C) must be used when reading from or writing to some register within the IMU.

3.1. Re-read and understand the sections of the IMU datasheet that concern configuration registers and how they are to be accessed.

NOTE: Any IMU register that has a name suffixed with "_XL" is associated with the built-in accelerometer. Any register that has a name suffixed with "_G" is associated with the built-in gyroscope.

As stated above, you will design two "C" functions to allow you to read from or write to *any* register within the IMU. These functions are described below.

Note that the functions and files in the template are named after the *LSM6DS3TR* and not the *LSM6DSL*. This will not make a difference to your device when you program it, but you can change the name of the functions and files if your device is the *LSM6DSL*.

void LSM_write(uint8_t reg_addr, uint8_t data) – Writes a single byte of data, `*data*`, to the address `*reg_addr*`, which is meant to be associated with an *LSM* register.

uint8_t LSM_read(uint8_t reg_addr) – Returns a single byte of data from an *LSM* register associated with the address `*reg_addr*`.

When writing code to configure or utilize the IMU, it is highly recommended that you download and utilize the relevant skeleton files provided for this lab. The given `*lsm6ds3tr.h*` header file is primarily meant to provide *declarations* for the above two functions (as well as another that we will be written in the following section of this lab), whereas the given `*lsm6ds3tr.c*` file is primarily meant to provide *definitions* for these two functions (and the third that has yet to be discussed). Separately, the `*lsm6ds3tr_registers.h*` header file is primarily meant to provide some useful symbols for address values associated with memory-mapped registers located within the *LSM6DS3TR,* somewhat similar to the AVR include file `*ATxmega128A1Udef.inc*`. See Figure 2 for a few examples of the symbols provided within this file. Recall that you can define your own set of constants, or condense certain aspects of "C" code, with macros. (In "C", a macro is defined with the `*#define*` preprocessor directive.)

```
INT1_CTRL       = 0x0D,
INT2_CTRL       = 0x0E,

WHO_AM_I        = 0x0F,

CTRL1_XL        = 0x10,
```

Figure 2. Some code segment taken from `*lsm6dsl_registers.h*`

Note that the *lsm6ds3tr*_registers.h will also work for the *LSM6DSL* IMU with the exception that there are some registers (none relevant to this lab) that do not exist on the *LSM6DS3TR* IMU so you should not try to access these.

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

- Enable the IMU via the relevant chip select signal. (Refer to how the *OOTB Robotics Backpack* is designed.)
- Send the appropriate amount of data to the IMU, based on the timing diagram given in the IMU datasheet.
- Disable the IMU via the relevant chip select signal.

3.2. Create a "C" function to be able to write to any of the registers within the IMU, ***void LSM_write(uint8_t reg_addr, uint8_t data)***, as well as another "C" function to be able to read from any of the available registers, ***uint8_t LSM_read(uint8_t reg_addr)***. Whenever appropriate, utilize other "C" functions previously written by yourself.

Now, we will verify *receive* functionality of the relevant SPI module. Within the IMU, there exists a predefined register to identify the device ID. This register, denoted by `WHO_AM_I` (refer to § 9.11 in the *LSM6DSL* datasheet), returns the default value of **0x6A**. (The *LSMDS3TR's* WHO_AM_I register, in § 9.12, will return a value of 0x69 – if this is the value that you read please use the *LSMDS3TR's* datasheet when configuring the IMU.) If you are able to create a program that successfully reads from the `WHO_AM_I` register, you should be able to reason that your SPI interface is appropriately designed.

3.3. Create a "C" program, **`lab6_3.c`**, to read from the `WHO_AM_I` register within the IMU and store the read value into some temporary variable, so that the read value

may be verified through the relevant debug window(s) of *Atmel Studio*.

3.4. Connect the *OOTB Robotics Backpack* to the *µPAD*, if necessary. Test the *receive* functionality of your system by debugging your program within *Atmel Studio*. Determine whether or not the expected value was read by your SPI system by any appropriate means, e.g., by way of a *Watch* window within *Atmel Studio*, etc. If you do not receive the expected data, use the LSA on your DAD board to debug the system. Probe the relevant pins via the *J2* header of the *OOTB Robotics* Backpack, and compare measurements received from the LSA to the relevant timing diagrams provided within the IMU datasheet.

4. CONFIGURING THE IMU ACCELEROMETER

Now that you have a set of functions that allow you to configure accelerometer registers within the IMU, you need to actually configure the accelerometer! However, before attempting to do so, read through the following comments, questions, and requirements:

- Before the accelerometer is configured or utilized at all, it is recommended that you first perform a software reset of the entire *LSM6* device. Refer to information regarding the *CTRL3_C* register.
- For this lab, you must have the accelerometer acquire acceleration forces from each of the X, Y, and Z coordinate directions simultaneously. How do you specify that the accelerometer should do so? Refer to information regarding the *CTRL9_XL* register.
- What should the *full-scale selection* be for the accelerometer? Overall, this criterion dictates the limits of acceleration force that the accelerometer can measure, and, in turn, dictates how precise an accelerometer measurement can be. As an example, of those full-scale settings provided by the IMU, "±2g" would provide the lowest limits, but would, in turn, provide the highest precision. A setting of "±2g" would likely not work well for fighter jets, but it likely would for the contexts of our course. Refer to information regarding the *CTRL1_XL* register.
- What should the *output data rate* be for the accelerometer? Overall, this criterion dictates how fast the accelerometer will output all of the relevant data. When deciding this rate, the relevant serial communication rate(s) and microcontroller system clock frequency should be taken into account. For our purposes, 208 Hz would likely be good, but some other rate, either faster or slower, might be more appropriate. Refer to information regarding the *CTRL1_XL* register.
- The IMU is designed to be able to generate an interrupt signal upon the accelerometer completing an acceleration measurement, which can be used to signal that data is ready to be read from the accelerometer. *In this lab, such an interrupt signal must be used to trigger an I/O interrupt on your microcontroller, instead of this signal being manually polled.* Refer to information regarding the *INT1_CTRL* register, as well as the relevant schematics. Which *input sense configuration* should be utilized by the relevant I/O pin of your microcontroller?

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

i. Perform a software reset of the IMU, via the *CTRL3_C* register.
ii. Configure the *CTRL1_XL*, *CTRL9_XL*, and *INT1_CTRL* registers, in some appropriate manner.
iii. Configure an I/O port interrupt to trigger upon the accelerometer completing a measurement.

4.1. Create a "C" function, ***void LSM_init(void)***, to initialize the IMU as described above.

In the next section 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 directions X, Y, and Z.

4.2. Determine how to access accelerometer data from the *LSM6* device. Refer to the *LSM6DSL* or *LSM6D3TR* datasheet.

NOTE: If the interrupt does not get triggered (with everything properly configured), a solution is to read from the accelerometer at least once (e.g., registers OUTX_L_XL and OUTX_H_XL) beforehand to effectively "wake up" the system to start sampling.

5. PLOTTING REAL-TIME ACCELEROMETER DATA

In this section of the lab, you will create a "C" program, **`lab6_5.c`**, to plot accelerometer data for each of the three coordinate directions X, Y, and Z in real-time, using *SerialPlot*.

SerialPlot is a very simple open source tool that can be used to visualize serial data. In this course, we will utilize the USART system of the XMEGA, which is automatically translated to the USB protocol, to send data to the *SerialPlot* application on our computer. This will allow us to visualize many different "channels" of data very easily. For the purposes of this lab, each channel will represent one of the X, Y, or Z axes data from the IMU.

So, to simply communicate with *SerialPlot*, UART must be used to allow communication between your computer and your microcontroller, via the USB connection on your *µPAD*. Then, to plot accelerometer data from your IMU, all you must do is output the data via UART in the correct sequence of "frames" for it to be interpreted and displayed properly by *SerialPlot*.

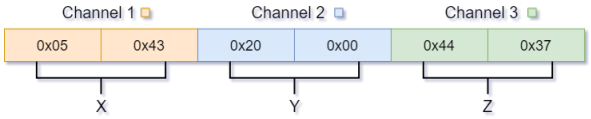

Figure 3. Example *SerialPlot* Simple Binary Data Format

SerialPlot allows for a few different ways input data can be formatted. The first, most simple format is depicted in Figure 3. This format is known as "Simple Binary" in *SerialPlot*. There are a few things to note about this figure, and about the Simple Binary" format in general:

- *SerialPlot* can plot up to 32 channels, so you must specify how many to use for your application. In Figure 3, three channels are used. This means in one "frame" of data, you must output three channels worth of data for it to work correctly.
- *SerialPlot* can interpret most simple data types such as 8, 16, and 32-bit signed and unsigned integers, as well as floats. Figure 3 is an example of what 16-bit data would look like in this format. You can choose whether it is plotted as signed or unsigned.
- *SerialPlot* can also support both little-endian and big-endian data formats.

For the same example given in Figure 3, assuming that three channels of signed 16-bit data in the little-endian data format were expected, the following three values should be supplied to *SerialPlot* in the same order given: 0x4305, 0x0020, and 0x3744. When outputting this data with UART, you would output the data in the order shown in Figure 3: 0x05, 0x43, 0x20, etc. After all six bytes have been sent to *SerialPlot* via USB, the plot will update with the three new values corresponding to each channel.

If you were using five channels of 8-bit unsigned data, you would only need to output five bytes, starting with the data that should correspond to channel one and ending with the data for channel five.

5.1. Download and install *SerialPlot.* Once installed, open the program and initialize each tab as follows:

5.2. For the *Port* tab, make sure your *µPAD* is connected to your computer and select the corresponding COM port. It should have "EDBG" in the name. Click the refresh button next to the "Port" field if it does not show up automatically. Choose the highest baud rate you can use for your given system clock frequency. Leave everything else as default, e.g., no parity, 8-bit data, 1 stop bit, no flow control. Wait to click *Open* until your program is running.

5.3. For the *Data Format* tab, choose Simple Binary, three channels, int16 number type, and little-endian endianness.

5.4. For the *Plot* tab, make sure that all three channels are visible, choose 1000 for the buffer size and plot width, make sure both check boxes are selected for "Index as X Axis" and "Auto Scale Y Axis", and choose "Signed 16 bits" for the range preset.

Each of the three channels will be used to represent one of the X, Y, and Z axes measured by the accelerometer. For the purposes of this lab, **make channel one display the X axis data, channel two display the Y axis data, and channel three display the Z axis data.**

NOTE: Don't forget to send the data in little-endian format (like you should have configured *SerialPlot* to accept), and that the data you are working with (from the accelerometer) is in a signed 16-bit format. This means that for every channel/axis, you need to output two bytes.

If you ever need to use the accelerometer data for arithmetic or logical comparisons, you should store it into *int16_t* variables, one per axis! This may be necessary for lab quizzes or hardware exams, so make sure you understand how to accomplish this.

For consistent results, the corresponding port interrupt should be initialized before initializing the IMU. Otherwise, configuring the registers CTRL9_XL and INT1_CTRL is initially not sufficient to get the IMU to generate interrupts. In this case, reading the XYZ registers before the implementation loop is necessary to "wake up" the IMU.

Clarifications/Suggestions:
The issue in the previous paragraph would not be an issue if the following is done. Make the IMU's interrupt active-low (which should be done in any case since the *Robotics Backpack* has a pull-up resistor on INT). **Also make the *ATxmega128A1U*'s pin interrupt trigger on a low level.** Since the first read of data from the IMU will clear the INT flag (make it high), then it can go low again pretty soon, after reading the first acceleration (or gyro) value. In the associated *ATxmega128A1U*'s ISR, the interrupt should be disabled and the ***accel_flag*** should be set. Once all of the accelerometer (and/or gyro) data has been read, then the *ATxmega128A1U*'s interrupt should be re-enabled.

5.5. Create a "C" file, **`lab6_5.c`**. Write a "C" function to transmit a stream of sensor data via your USB Serial Port, in the correct order according to the *Simple Binary* format, following the pattern in Figure 3 and as described above. Make sure that your function transmits the correct number of bytes, in the right order. Your function should also utilize other functions (or macros) that are available in lab6_files.zip, provide to you, such as ***usartd0_out_char()***. Be careful to *not* use ***usartd0_out_string()***, since this routine expects a null-terminated string, which your data will likely not be! Instead, it is advised that you create a function, similar to ***usartd0_out_string()***, that outputs *a specific number of bytes* via the relevant USART module, rather than continually output until a null character is encountered.

Now, recall that the *LSM6* device will be configured to interrupt your microcontroller, upon completing an acceleration measurement. **The routine that you will create below must only output data for *SerialPlot* when new data from the accelerometer becomes available, since we only care to plot new data.** (There is no point in outputting the same data repeatedly.)

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* output any data to *SerialPlot* within an ISR. Instead, a global flag (e.g., ***volatile uint8_t accel_flag***) should be asserted to alert your main program that new accelerometer data is ready to be output.

5.6. Create a main routine within your "C" file, **`lab6_5.c`**, to plot accelerometer data for all three coordinate directions (X, Y, and Z) via the *Data Streamer*, as described above. *Remember to only transmit new accelerometer data.* As mentioned previously, configure the relevant USART module with the highest baud rate possible for your system clock frequency.

After your program is running, and after you have selected the appropriate COM port that corresponds to your *µPAD* within *SerialPlot*, click *Open* either [1] in the *Plot* tab or [2] at the top of the *SerialPlot* window.

NOTE: Recognize 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.

5.7. Using *SerialPlot*, plot all three axes of accelerometer data, in real-time. Take a screenshot of your graph, including all three waveforms.

PRE-LAB EXERCISES

ii. Why is it a better idea to modify global flag variables inside of ISRs instead of doing everything inside of them?
iii. To output two unsigned 32-bit values **0xEC47A3B1 [CH1] and 0x110AFCE8 [CH2]** to *SerialPlot*, list all the bytes in the order you would send them via UART.
iv. What is the most positive value that can be received from the accelerometer (in decimal)? What about the most negative?

EXTRA CREDIT

For 10% extra credit, implement the same functionality for the gyroscope as you did for the accelerometer. There will be no PI help for extra credit.

PRE-LAB PROCEDURE SUMMARY

1) Answer all pre-lab exercises, when appropriate.
2) In § 1, create "C" functions to configure/utilize the SPI module that will connect to the IMU.
3) In §§ 2 and 3, test the SPI "C" functions that you wrote in § 1, and create "C" functions to communicate with the IMU. Take a relevant screenshot of the LSA, as described in § 2.7.
4) In § 4, create a "C" function that initializes the built-in accelerometer within the IMU.
5) In § 5, create a "C" program to plot accelerometer data for all three coordinate directions (X, Y, and Z), using *SerialPlot*. Take a screenshot of your plot, as described in § 5.7.

APPENDIXES

A. USING THE PROVIDED *LSM6DS3TR* AND *LSM6DSL* C FILES

Three files have been provided for you: *lsm6dsl.c, lsm6dsl.h, and lsm6dsl_registers.h* for the *LSM6DSL*. (*Similarly, the LSMDS3TR registers have been provide for you: lsm6ds3tr.c, lsm6ds3tr.h, and lsm6ds3tr_registers.h.)*

The ***LSM6DSL_registers.h*** file contains **very** useful definitions which give labels to the addresses and bitfields of the *LSM6DSL* internal registers, as shown in Figure 2.

The ***LSM6DSL.c*** file is blank and may be used as a place to *define* the *LSM6DSL*-related functions as described in § 3.

The ***LSM6DSL.h*** file is more interesting. First and foremost, it will serve as a place to put the *declarations* of the functions you *defined* in ***LSM6DSL.c***. Additionally, it contains several type definitions which are provided for you to use. You are **not** required to use any of these type definitions; however, they will likely make your life a lot easier when it comes to managing the inertial data from the *LSM6DSL*. The following typedefs are provided:

LSM6DSL_module_t – An enumeration that can be used to specify a device (accelerometer or gyroscope) if you ever need to. An example would be a function that could act on either the accelerometer or the gyro. This enumerated type could be used as one of the argument types to the function.

LSM6DSL_data_raw_t – This type is a structure that can be used to store the data when read from the IMU. It supports both the accelerometer and the gyro, but you can use it exclusively for the accelerometer if you don't choose to get the gyroscope working.

LSM6DSL_data_full_t – This type is a structure that can be used to access the full X, Y, or Z axis data in a signed 16-bit format. This is used in the following typedef.

LSM6DSL_data_t – This type is a union that will allow you to access the data read from the *LSM6DSL* in a much easier way. You will need to declare an instantiation of this union in your code, like so:

```
LSM6DSL_data_t lsm_data;
```

You are creating a variable of the *LSM6DSL_data_t* type, which is a union. You can choose the name of the variable; it doesn't necessarily have to be *`lsm_data`*. Now, you should be able to store the *LSM6DSL* data directly into this union by accessing the "*LSM6DSL_data_raw_t"* member. Here is an example:

```
lsm_data.byte.accel_x_low = /* `LSM6DSL_read` call*/
```

After you read the rest of the accelerometer data in this fashion, you then have it all in a *contiguous* section of memory. Now, you can access *either* the bytes individually, *or* the full words that correspond to each axis' data. For example, if you wanted to access the accelerometer's Y-axis data, you would type the following:

```
lsm_data.word.accel_y
```

Notice the difference between this and the previous snippet. The same union (*`lsm_data`*), but different structure, is accessed. Because it is a union, these two structs share the same memory. When you loaded the individual bytes with all the IMU data, you filled the section of memory that corresponds to the union. This gives you the ability to either access the bytes individually, or the entire words!

Understanding how exactly unions and structures work isn't necessarily in the scope of this class, especially if this is the first time you have used "C". If you are interested in learning more, there is plenty of information online about these topics that you can learn more from.

Again, you do *not* have to use these constructs. They are just provided to introduce you to some more advanced functionality of "C".

B. TROUBLESHOOTING SERIAL PLOT

The goal of this section is to help you with some of the issues students typically have with the serial plot software.

Problem #1 (Flipped Low & High Bytes)

- Many students run into an issue where their data seems to fluctuate rapidly through a large range of values. This typically happens when the data from your *µPAD* is flipped. *SerialPlot* is taking in your low byte as a high byte and your high byte as the low byte. The simplest way to fix this is to restart you program and ensure that Serial Plot is running before you start the program on your *µPAD*.

Problem #2 (Baud Rate)

- If you are getting junk data from your *µPAD* on *SerialPlot*, this usually indicates that the Baud Rate is incorrectly input into *SerialPlot*. Check both your code and *SerialPlot* values to ensure that they match.

Problem #3 (PuTTY)

- If you are getting no data from your *µPAD*, ensure that you do not have PuTTY running. Only one program can have control over a serial connection of your *µPAD*, and PuTTY and *SerialPlot* don't get along, so only use one at a time.

C. OPTIMIZATION LEVELS

When using the "C" programming language within our course, you are required, unless told otherwise, to turn off the compiler optimization tool utilized by *Atmel Studio*. To do so, perform the following within *Atmel Studio*.

1. Navigate to *"Project | <project_name> Properties"*.
2. Select *"Toolchain"*.
3. Select the "*Optimization"* listing, located under `*AVR/GNU "C" Compiler*` section.
4. Set "*Optimization Level*" to "***None (-O0)***".

For more details, read the last page of the document called *Create, Simulate, and Emulate a Projectz* on the Software/Docs page of our class website.

D. POSSIBLE SOURCES OF ERRORS

- SPI not enabled properly on the correct port
- *LSM6* device not enabled properly (including interrupts)
- Pin interrupts not configured correctly on master
- Pin directions not configured properly
- SS line not being pulled low during a SPI transaction
- Software flag not declared as volatile
- Software flag not checked properly in main
- Software flag not cleared immediately upon entering the conditional where it's checked