OBJECTIVES

- Understand general synchronous serial communication concepts.
- Become familiar with the Serial Peripheral Interface (SPI) serial communication protocol and with the SPI system available within the ATmega128A1U.
- Learn how to utilize the SPI protocol to interface with an external Inertial Measurement Unit (IMU) sensor package.
- Stream and plot real-time 3D (XYZ) acceleration data using the SPI and USART systems, as well as the SerialPlot software program.

INTRODUCTION

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 between systems 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 with synchronous serial communication when considering the accuracy, or error, of a transmission. The main reason for this stems from the fact that for asynchronous systems to uphold a decided data transfer rate while also minimizing error, it is necessary to include synchronization bits throughout a data transmission, which ultimately produces major overhead, and in turn, drastically reduces any actual data transfer rate.

One popular synchronous serial communication protocol is the SPI protocol. With SPI, communication is possible between multiple devices, although only one device at a time may act as the controlling (master) device, while all other devices must act as responding (slave) devices.

With the SPI protocol, full-duplex communication is possible, i.e., two devices can talk to each other at the same time, although the master device must always start any communication between devices. To start communication, the master generates the synchronized clock signal. Often, a Slave Select (SS) or Chip Select (CS) signal is connected between the master and slave device(s) which enables or selects the slave(s) with which the master device chooses to communicate. Upon each generated clock signal (a specific type of edge of the signal is the determining factor), a bit of data is shifted into and out of a shift register located within each device via two signals: the Master-Output-Slave-Input (MOSI) signal and the Master-Input-Slave-Output (MISO) signal.

Although the SPI communication protocol is very common, it does not nearly account for all synchronous serial communication protocols; some other popular serial standards are Inter-Integrated Circuit (IIC, or more commonly, I2C), Universal Serial Bus (USB), and Controller Area Network (CAN). Moreover, the synchronous mode within the USART system of the ATmega128A1U (the S of USART) is another example of a synchronous serial communication protocol, and it closely resembles SPI.

LAB STRUCTURE

In this lab, you will explore synchronous serial communication by utilizing the SPI protocol. More specifically, you will learn how to configure and use the SPI system within the ATmega128A1U for the purposes of communicating with an LSM330 Inertial Measurement Unit (IMU) located on the OOTB Robotics Backpack. The LSM330 contains a MEMS (Micro-Electro-Mechanical System) 3D digital accelerometer and a MEMS 3D digital gyroscope.

Finally, 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 USART system.

REQUIRED MATERIALS

- Atmel ATXmega128A1U AU Manual (doc8331)
- LSM330 Datasheet
- Skeleton files: lsm330.c and lsm330.h
- OOTB µPAD v2.0 with USB A/B cable
- OOTB Robotics Backpack
- OOTB Robotics Backpack Schematic
- Digital Analog Discovery (DAD) kit with WaveForms software

SUPPLEMENTAL MATERIALS

- Atmel ATXMEGA128A1U Manual (doc8385)
- Header file: lsm330_registers.h
- SN74LVC1G3157 SPST Analog Switch Datasheet
- Setup and Use of the SPI (doc2585)

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.


PRE-LAB PROCEDURE

REMINDER OF LAB POLICY

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

1. INTRODUCTION TO SPI AND THE LSM330

In this section, you will begin to become familiar with the SPI protocol, the SPI system available within the ATxmega128AU, and the LSM330 inertial measurement unit (IMU). 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 ATxmega128AU was designed to simplify the procedure of emulating the SPI protocol.
- From the perspective of the microcontroller, the LSM330 IMU is an external peripheral.


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 ATxmega128AU 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 slave?
- How will the slave device(s) be enabled? If a slave 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 LSM330 IMU peripheral chip located on the OOTB Robotics Backpack, for the purposes of interfacing with a MEMS 3D digital accelerometer and MEMS 3D digital gyroscope.

Like with the ATxmega128AU, the LSM330 utilizes 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 ATxmega128AU does), the LSM330 utilizes only the SPI (or I²C) protocol.

Although it generally does not matter which physical connections are chosen for SPI communication, it is important to recognize that the LSM330 component located on the OOTB Robotics Backpack was designed to utilize specific I/O pins on the ATxmega128AU. Additionally, to allow the potential for both SPI and I²C 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 LSM330.

Below, you will begin to become familiar with the LSM330 component and understand how it was designed to connect to your microcontroller.

1.2. Read through the LSM330 datasheet. Pay extra attention to §§ 2, 4, 6.2, 7, and 8, as well as to Table 2 (focus on the SPI function of each pin), Table 6, and Table 9.

1.3. Determine which signals from the ATxmega128AU will be utilized to communicate with the LSM330 chip on the OOTB Robotics Backpack. Refer to the appropriate schematic(s) and manual(s).

PRE-LAB EXERCISES

i. Considering SPI communication between the ATxmega128AU and LSM330 in context of the relevant OOTB kit, answer each of the questions within the previously given bulleted list.

2. COMMUNICATING WITH THE LSM330

In this section, you will incrementally write several functions in the C programming language for the purposes of communicating with the LSM330 component, performing tests when appropriate.

First, you will design C functions to utilize the SPI system within the ATxmega128AU. These functions should allow a generic use of the SPI system and should follow the provided spi.h and spi.c header and source files, respectively.

2.1. Within the provided spi.c file, complete the C function, void spi_init(void), to initialize the necessary SPI module within the ATxmega128AU as well as the necessary control signals on the µPAD, for the purposes of communicating with the relevant LSM330.

2.1.1. 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, since the dedicated slave select pin within the ATxmega128AU SPI module will not be utilized by the LSM330, make sure to appropriately handle this pin. For more details, refer
2. Write a second C function, `void spi_write(uint8_t data)`, to transmit a single byte of data from the master device (the Atmega1284U), 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 necessary SPI status register. Interrupts should not be used for this purpose, as it inhibits portable code. (Note that more advanced programming techniques outside the scope of this course could be used to circumvent this portability issue.) After the SPI transmission is complete, your function should end. Do not enable/disable any slave devices within this function, as multiple sequential calls to the function are possible, rendering multiple enables/disables unnecessary and inefficient. This will be handled later, in a different portion of code.

2.3. Write a third C function, `uint8_t spi_read(void)`, to read a single byte of data from a connected slave 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 LSM330, and then after the transmission is complete, return the contents of the data register from the function. 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. 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 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.

In this section of the lab, you will verify communication abilities for the appropriate SPI module with your DAD and write a simple C program. (The following method does not allow you to easily check receive functionality, although a different, more intuitive technique will be used to test this in the following section of this lab.)

NOTE: Pay more attention to the accelerometer functionality as opposed to the gyroscope, since the use of the gyroscope is not required in this course.

2.4. Write a simple C program (`lab6_2.c`) to initialize the relevant SPI module and to continually transmit 0x53. Utilize any appropriate functions created in the previous sections. Also, to simulate the LSM330 slave device being enabled/disabled, with no backpack attached to the top of the µPAD, assert an available pin on the µPAD 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 your SPI module is correctly transmitting 0x53, you will view all appropriate SPI signals with the SPI digital bus analyzer function of the Logic (LSA) program within the 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 Chip Select [CS], or Slave 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 LSM330 datasheet), and Data (the signal used to transmit data to or receive data from a slave device, which can be represented by many things, e.g., MOSI/MISO, Slave Data In [SDI], or Slave Data Out [SDO]).

2.5. Within the Logic program of Waveforms, select SPI from the “Click to Add channel” dropdown menu (shown on the left in Figure 1). You will be prompted with the Add SPI menu (shown on the right in Figure 1). Configure any necessary signals. The chosen DIO pin on the DAD 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.

2.6. Remove any backpack from your µPAD to have access to the relevant I/O port pins. 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 0x53. You may wish to use the slave select signal as a falling-edge trigger source within the LSA. You can also use a “protocol trigger” to trigger the LSA when an SPI transmission starts.

2.7. Take a screenshot of your LSA window transmitting a single full byte of data, including all necessary signals. Make sure that you choose an appropriate time base so that a single transmission is clearly visible within the LSA.

3. RECEIVING WITH SPI & COMMUNICATING WITH THE LSM330

In the previous section, the DAD was used to measure highly precise synchronous signals with a digital bus analyzer function. Although it is possible to measure the MISO signal with the technique used in the previous section of this lab, you will not
only test the read functionality of your SPI system, but also whether or not it is configured to properly interface with the

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_registers.h

LSM330. To accomplish this, you will create a new C program (lab6_3.c) to read from an accelerometer register within the external LSM330 IMU.

Overall, the LSM330 has a plethora of configuration registers for the internal accelerometer (and gyroscope), just like within a microcontroller and other peripherals (e.g., an external UART, an LCD, etc.). However, unlike when accessing a register within the ATxmega128A1U, where a parallel data bus is used to write and read data, SPI (or I2C) must be used to write to or read from a register within the LSM330.

NOTE: Any LSM330 register name ending with “_A” indicates that the register is associated with the built-in accelerometer. Register names ending in “_G” reference registers that are associated with the built-in gyroscope.

Before configuring registers, as mentioned before, the LSM330 is capable of both SPI and FC communication. To simplify circuitry on your µPAD, as well as the PCB design, certain signals were designed to have the ability to select between the SPI and I²C 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 (more specifically, an analog, bidirectional, 2-input multiplexer), located on the OOTB Robotics Backpack. At datasheet for this switch is linked in the “Supplemental Materials” section of this lab document. In other words, there are additional signals located within your OOTB 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.

3.1. Re-read and understand the pertinent sections of the LSM330 datasheet.

3.2. Determine what signals must be controlled by your microcontroller, and why they must be controlled. Furthermore, determine where these predefined signals connect to on the ATxmega128A1U, and how to control them in software. Refer to the OOTB Robotics Backpack schematic, as well as the LSM330 datasheet.

Before reading from an accelerometer register within the LSM330, you must first design two functions to allow you to write to or read from any of the accelerometer registers within the LSM330:

void accel_write(uint8_t reg_addr, uint8_t data) – Writes a single byte of data (data) to a specific accelerometer register (reg_addr) within the LSM330.

uint8_t accel_read(uint8_t reg_addr) – Returns a single byte of data that is read from a specific accelerometer register (reg_addr) within the LSM330.

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_registers.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 include file for the ATxmega128A1U. The intention is for you to refer to easily identifiable names, rather than “magic numbers”, or 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 OOTB 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).

3.3. Create C functions to be able to write to any of the available accelerometer registers within the LSM330 (void accel_write(uint8_t reg_addr, uint8_t data)) and to be able to read from any of the available accelerometer registers (uint8_t 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. See the Appendix for more information about these files.

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 § 8.1 in the LSM330 datasheet), returns the default value of 0x40. Your program (lab6_3.c) should verify that this value is received upon reading from the WHO_AM_I_A register. This will tell you whether your interface with the LSM330 is configured properly or not.

3.4. Create a C program (lab6_3.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.

If you do not receive the expected data, use the LSA on your DAD board to debug the system. Compare the timing diagram you see using your LSA to the SPI timing diagrams in the LSM330’s datasheet.
4. CONFIGURING THE LSM330 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 \texttt{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, requirements, and questions to pose to yourself:

- The LSM330 is designed to be able to generate an interrupt signal upon the accelerometer completing an acceleration measurement, signaling that data is ready to be read from the accelerometer. In this lab, this interrupt signal must be used to generate an external interrupt on your microcontroller, 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 \texttt{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 accelerations 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 \texttt{CTRL\_REG5\_A}.)

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

i. Configure an external interrupt within your microcontroller to trigger upon the accelerometer completing a measurement.

ii. Configure \texttt{CTRL\_REG4\_A} and \texttt{CTRL\_REG5\_A} within the LSM330.

4.1. Create a C function, \texttt{void accel_init(void)}, to initialize the LSM330’s accelerometer, 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 planes, X, Y, and Z.

4.2. Determine how to access accelerometer data from the LSM330. Refer to the LSM330 datasheet.
5. PLOTTING REAL-TIME ACCELEROMETER DATA

In this section of the lab, we will create a C program (lab6_5.c) to plot accelerometer data for each of the coordinate planes (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 use 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 LSM330.

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](image)

SerialPlot allows for a few different ways input data can be formatted. The first, and simplest 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 formatted data.

For the example in Figure 3, assuming you were using three channels of signed 16-bit data using little-endian formatting, the three values would be the following, in order: 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:

- 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.
- For the Data Format tab, choose Simple Binary, three channels, int16 number type, and little-endian endianness.
- For the Plot tab, make sure 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, you should 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.

5.2. 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 you have already created, such as usartd0_out_char(). Be careful if you decide to use usartd0_out_string(), because it expects a string to be null-terminated, and your data may not be! It is advised to write a new function that is similar to usartd0_out_string(), that outputs a specific number of bytes instead of checking for a null character.

Now, recall that the LSM330 will be configured to interrupt your microcontroller, upon completing an acceleration measurement. The program 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.

5.3. Create a main routine within your C file (lab6_5.c) to plot accelerometer data for all three coordinate planes (X, Y, and Z) by communicating with the Data Streamer, only when new accelerometer data becomes available, as described above. For the purposes of this lab, as mentioned earlier, configure the appropriate USART module with the highest baud rate possible for your system clock frequency.

After your program is running, click Open in the Plot tab or at the top of the SerialPlot window after you have selected the appropriate COM port that corresponds to your μPAD.

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.

5.4. 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. What are some examples of useful macro functions, in the context of this lab?
iii. What is the highest speed SPI clock that the IMU can handle?
iv. Why is it a better idea to modify global flag variables inside of ISRs instead of doing everything inside of them?
v. To output two unsigned 32-bit values (0x12345678 [CH1] and 0x9A34F21104 [CH2]) to SerialPlot, list all the bytes in the order you would send them via UART.
vi. 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 LSM330 IMU.
3) In §§ 2 and 3, test the SPI functions that you wrote in § 1 (with programs lab6_2.c and lab6_3.c, respectively), and create functions to communicate with the LSM330 IMU. Take a screenshot of your LSA, as described in § 2.4.
4) In § 4, create a C function that initializes the built-in accelerometer, within the LSM330.
5) In § 5, create a C program (lab6_5.c) to plot accelerometer data for all three coordinate planes (X, Y, and Z), using SerialPlot. Take a screenshot of your plot, as described in § 5.4.
APPENDIX

A. USING THE PROVIDED LSM330 C FILES

Three files have been provided for you: `lsm330.c`, `lsm330.h`, and `lsm330_registers.h`.

The `lsm330_registers.h` file contains very useful definitions which give labels to the addresses and bitfields of the LSM330 internal registers, as shown in Figure 2.

The `lsm330.c` file is blank and may be used as a place to define the LSM330-related functions as described in § 3.

The `lsm330.h` file is more interesting. First and foremost, it will serve as a place to put the declarations of the functions you defined in `lsm330.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 LSM330. The following typedefs are provided:

- `lsm330_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.

- `lsm330_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.

- `lsm330_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.

- `lsm330_data_t` – This type is a union that will allow you to access the data read from the LSM330 in a much easier way. You will need to declare an instantiation of this union in your code, like so:

  ```
  lsm330_data_t lsm_data;
  ```

  You are creating a variable of the `lsm330_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 LSM330 data directly into this union by accessing the “`lsm330_data_raw_t`” member. Here is an example:

  ```
  lsm330_data.byte.accel_x_low = /*lsm330_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:

```
  lsm330_data.word.accel_y
  ```

Notice the difference between this and the previous snippet. You’re accessing the same union (`lsm330_data`), but a different one of the structs that are defined within it (byte vs. word). 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.