# **Communication Network**

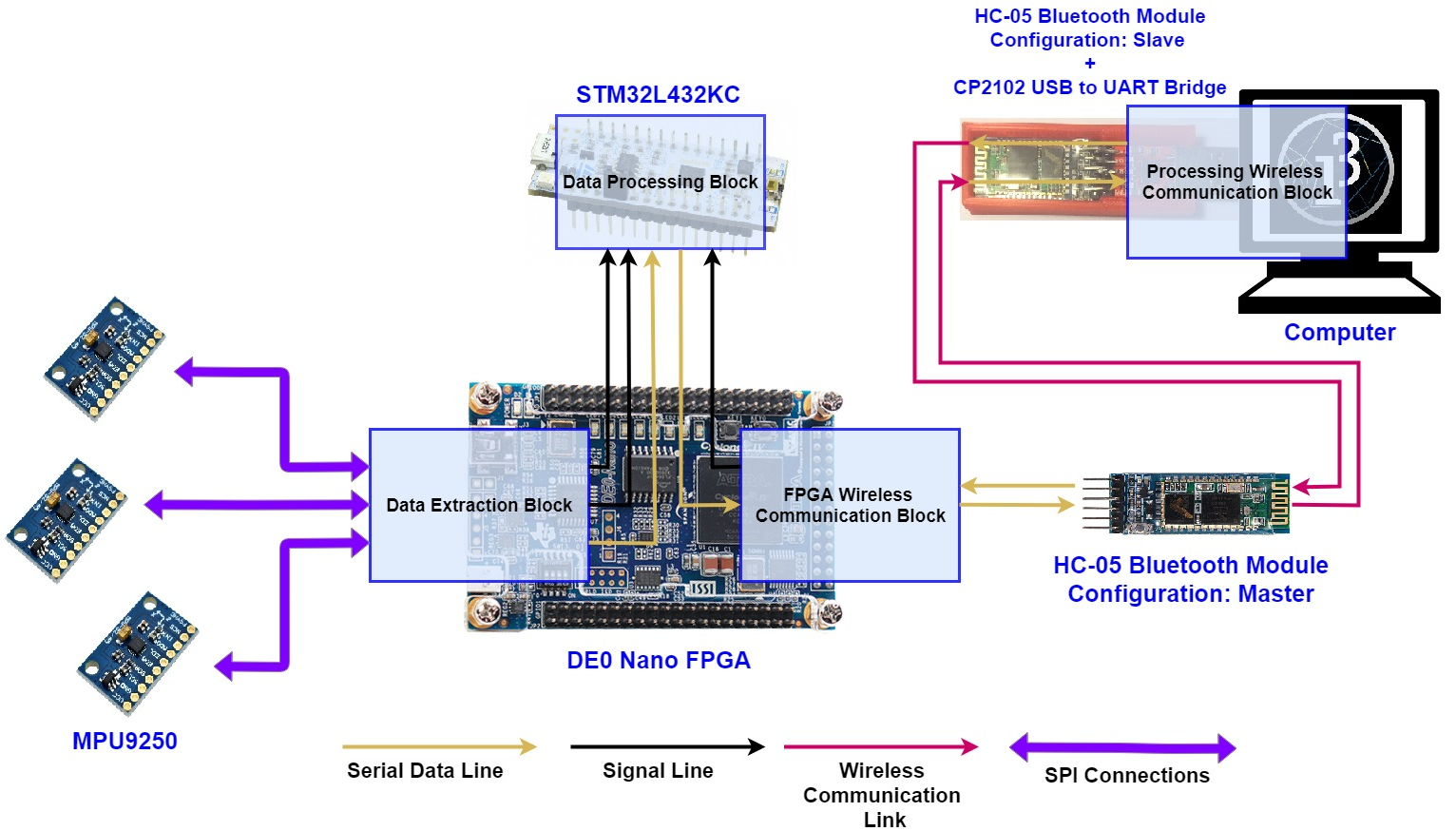
The purpose of this communication network is to provide a means of moving data from three internal measurement units (IMUs), through the system, to the destination which is the processing simulation. This communication network consists of 4 distinct sections; the data extraction block, data processing block, FPGA wireless communication block and the processing communication block. Figure 1 provides an overview of how the data moves across hardware before it arrives in the processing simulation. This part of the design section is concerned with the design of the network and how it enables the data to move before it reaches the processing simulation. The Blocks inside of the FPGA function at a clock speed of 1MHz and each component is synchronised through the same clock. Because each component in the system is connected to this clock, the connections to the clock have not been shown to maintain clarity in the upcoming diagrams.

Figure 1: Diagram depicting the flow of data across hardware.

## **Data Extraction Block**

The ‘Data Extraction Block’, as the name suggests, is responsible for the extraction of data from the internal measurement units. The data is extracted using the serial peripheral interface and then is stored in a buffer where it waits to be transmitted to the ‘Data Processing Block ‘using the serial peripheral interface once again. To achieve parallel data extraction, three IMU extraction components are present, each consisting of the above mentioned serial peripheral interface component, buffer, and also an address loader, data extraction controller and data transmission controller. The address loader is used to load addresses to the SPI such that it can request data from registers inside the MPU9250 which contain the gyroscope and accelerometer data. The two controllers are used to synchronise the extraction and transmission process.

Because the there are three IMU extraction blocks within the data extraction block but there is only one serial peripheral interface component that communicates with the data processing block inside of the STM32L432KC board, the ‘FPGA-STM32L432 SPI Access Controller’ is implemented to control the access such that data can be sent without any conflict. Figures 2 and 3 show all the components that can be found inside the data extraction block and how they are connected.

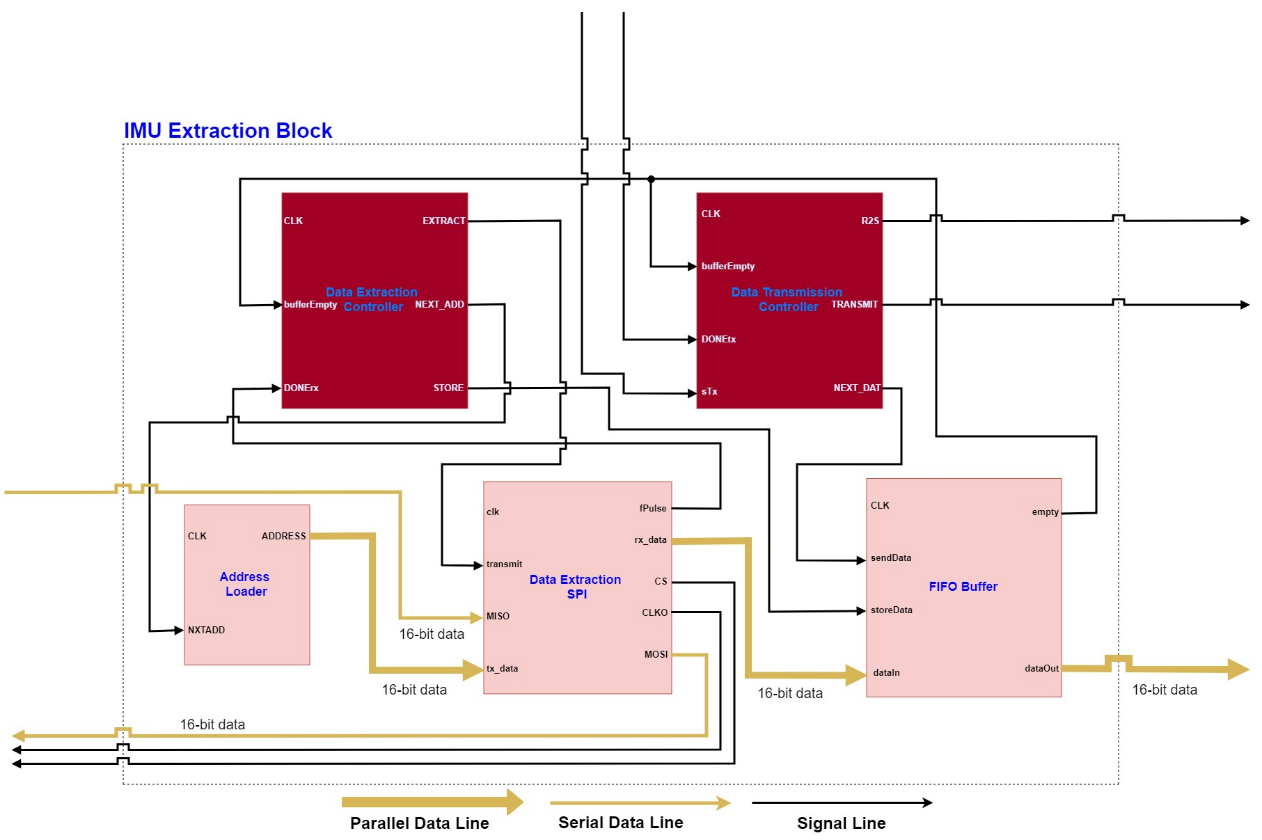
Machine generated alternative text:
Data Extraction 
Block 
CLK 
r2slMU1 
txlMU1 
datalMU1 
16-bit data 
dataOut 
TRANSMIT 
FPGA-STM32 
clk 
MISO 
cs 
CLKO 
FPGA-STM32rx data 
r2slMU2 
SPI 
Access 
txlMU2 
Controller 
datalMU2 
r2slMU3 
txlMU3 
datalMU3 
DONEtx 
IMU Extraction 
strtTx3 
strtTx2 
strtTx1 
R2S 
SPI 
tx data 
transmit 
DONEtx 
MOSI 
fPulse 
R2S 
DONEtx 
16-bit data 
R2S 
Text 
Blockl TRANSMI 
IMIJ Extraction 
Block2 TRANSMIT 
MISO 
IMLJ Extraction 
Block3 TRANSMIT 
dataOut 
16-bit data 
MISO 
16-bit data 
O 
x 
dataOut 
o 
16-bit data 
MISO 
16-bit data 
16-bit data 
Serial Data Line 
O 
x 
dataOut 
o 
16-bit data 
O 
x 
o 
16-bit data 
16-bit data 
Parallel Data Line 
16-bit data 
Signal Line 

Figure 2: Diagram depicting the component composition and connections between the inside the Data Extraction Block.

Figure 3: Diagram depicting the component composition and connections inside the IMU Extraction Block that is part of the Data Extraction Block.

### **Accessing MPU9250 registers**

MPU9250 internal measurement unit uses the serial peripheral interface to communicate with the master device. The first bit (bit 15) sent to the device indicates a read (1) or a write (0). The next 7 bits indicate the actual address of the register to be accessed. The last 8 bits represent a dummy byte which is used to enable the device to write the 8-bit data from the accessed register back to the master device. Table 1 depicts the data format that is used to write to the device. A complete data sample is made up of 16-bits contained in two registers. For example, accelerometer data for the x-axis is contained in two registers, one register contains the most significant byte and the second contains the least significant byte. By combining these two bytes a complete x-axis accelerometer data sample is formed.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit** | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Description** | Read (1)/Write (0) | Address bit 6 | Address bit 5 | Address bit 4 | Address bit 3 | Address bit 2 | Address bit 1 | Address bit 0 | Dummy Byte Bit | Dummy Byte Bit | Dummy Byte Bit | Dummy Byte Bit | Dummy Byte Bit | Dummy Byte Bit | Dummy Byte Bit | Dummy Byte Bit |

Table : Data format used to write to MPU9250 registers.

### **Data Extraction Controller**

The extraction controller is responsible for the extraction and storage of the IMU data that it extracts. The controller is a state machine that consist of several states which it transitions between based on the input in order to accomplish its task. Table 2 describes the functions that the inputs and outputs of the extraction controller serve to drive the state transitions. The following descriptions of the states explain how the controller operates.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin Name** | **Input/Output** | **Usage** | **Signal Type** |
| CLK | Input | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock |
| bufferEmpty | Input | The FIFO buffer can signal to the controller through this pin that it is either full (0) or empty (1). | High/Low |
| DONErx | Input | The SPI can signal the controller that a new sample has been extracted and can be stored. | One clock cycle pulse |
| EXTRACT | Output | This pin is used to command the SPI component to start the extraction of another sample. | One clock cycle pulse |
| NEXT\_ADD | Output | This pin is used to command the address block to load the next address to be transmitted to the IMU through the | One clock cycle pulse |
| STORE | Output | This pin is used to command the FIFO buffer to store the newly extracted data piece. | One clock cycle pulse |

Table2: Description of inputs and outputs of the Data Extraction Controller Component.

* When in the ‘IDLE’ state, the component waits for the bufferEmpty signal to go high in order to start the data extraction process. The controller transitions to the ‘NXTADDR’ state when the bufferEmpty signal goes high.
* When in the ‘NXRADDR’ state, the controller sends the ‘NEXT\_ADD’ signal to command the Address Loader component to load the new address on its output for the SPI master component to use. The controller then automatically transitions to the ‘STRTrx’ state.
* When in the ‘STRTrx’ state the controller sends a signal using the ‘EXTRACT’ pin to command the SPI component to start the transaction. The controller then automatically transitions to the ‘WAITINGrx’ state.
* When in the ‘WAITINGrx’ state, the controller waits for the ‘DONErx’ input to go high. This signal through this input notifies the controller that an SPI transaction has been completed. When this signal is received, the controller transitions to the ‘STOREstate’.
* When in the ‘STOREstate’, the controller sends a one clock cycle pulse using the ‘STORE’ output pin to command the FIFO buffer to store the data that the SPI extracted in the latest transaction. The controller then increments counter which tracks the number of transactions; this way the controller knows how many more data samples need to be extracted to have a complete set of gyroscope and accelerometer data samples (see Address Loader component). If the counter value is less than 11 then the state transitions back ‘NXTADDR’ state, otherwise the controller resets the counter to zero and transitions to the ‘W4BFFRrx’ state.
* When in the ‘W4BFFRrx’ state, the controller waits for the bufferEmpty signal to go low. This state is used to let the FIFO buffer finish its internal processes before it sends the signal as the buffer is designed to only holds as many samples as there are in a complete set of gyroscope and accelerometer data before it becomes full. If the controller transitioned to the IDLE state before the bufferEmpty signal transitioned to logic 0, the controller would continue initiating new sets of transactions causing data corruption in the later stages of operation such as data transmission. Adding this state between the IDLE state and the ‘STOREstate’ prevents this from happening. When the bufferEmpty input transitions to logic 0, the controller transitions to the IDLE state.

### **Address Loader**

The Address Loader component stores the addresses of the MPU9250 registers which contain the accelerometer and gyroscope data (See Appendix A). When commanded by the Data Extraction Controller this component loads these addresses on its output for the SPI component to use to write to MPU9250 in order to extract accelerometer and gyroscope data. Table 3 describes how each input and output of this component is used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin Name** | **Input/Output** | **Usage** | **Signal Type** |
| CLK | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock |
| NXTADD | INPUT | This input is used by the extraction controller to command the addressBlock to assert next address on its output. | One clock cycle pulse |
| ADDRESS | OUTPUT | This output is connected SPI's tx\_data input such that the SPI can use the addresses to extract data from the IMU. | Data |

Table 3: Description of inputs and outputs of the Address Loader component.

The Address Loader component has an internal counter which tracks how many addresses have been loaded during the extraction process. This counter is reset to at the beginning of each new set of transactions and stops at the last address at the end of each set of transactions. This way the data is always extracted in the correct order. When the ‘NXTADD’ signal goes high, the component increments the counter and thereby asserts a new address on the ‘ADDRESS’ output for the SPI component to use.

### **FIFO Buffer**

This First In First Out Buffer design serves two purposes; the first is to store the data that was extracted from the internal measurement unit before it is sent to the ‘Data Processing Block’ and the second is to append a data identification byte the each 8-bit data piece extracted from the IMU. The reason for appending this bit is the fact that when the data is sent to the STM32L432 development board, the device needs to know what kind of data it is receiving such that it knows how to process and sue this data. This byte is also used by the STML432 for error checking to determine whether the batch of data was sent correctly. The identification byte conveys the following information (See Appendix C for more detail):

**IMU ID** – The IMU identification lets the STM32L432 determine which IMU the data came from so that it can apply appropriate offsets to the data that are unique to each internal measurement unit. Additionally, when the STM32L432 device sends the data to the FPGA for wireless transmission it needs to know what identification byte to append to the data such that the processing simulation also know how the data should be used.

**X, Y or Z axis** – Specifies whether the data byte conveys x, y or z axis information.

**Accel, Gyro or Mag** – Specifies whether the data byte comes from the accelerometer, gyroscope or magnetometer. The STM32L432 needs to know this information in order to process the data correctly as gyroscope data undergoes different processes to that of the accelerometer to obtain useful orientation information.

**Upper or Lower Byte** – Specifies whether the data byte is the most significant byte or the lest significant byte. This information is needed such that the two halves of the sample can be concatenated in the correct order.

The FIFO Buffer is a state machine which uses three states to store and send the data for transmission based on the input signals. Table 4 describes how each input and output of this component is used and the following description of each state the buffer can transition to explain the operation of the buffer.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin Name** | **Input/Output** | **Usage** | **Signal Type** |
| CLK | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock |
| storeData | INPUT | Using this pin the extraction controller can command the FIFO buffer to store the newly extracted sample. | One clock cycle pulse |
| sendData | INPUT | Using this pin the Transmission Controller can command the FIFO buffer to assert the oldest sample on the output such that it can be loaded to the FPGA-STM32L432 SPI master for transmission. | One clock cycle pulse |
| dataIn | INPUT | This pin is used to load the newly extracted sample from the SPI to the FIFO buffer. Data is only loaded when the extraction controller sends a pulse signal to the storeData pin. | Data |
| dataOut | OUTPUT | This pin is used to output the data to the FPGA-STM32L432 SPI master. This pin is connected to the accessControl block such that only the IMU\_Extraction\_Block which has access permission granted by the accessControl block can pass data to the SPI. New data is asserted on the output only when the transmission controller sends a pulse to the sendData pin. | Data |
| empty | OUTPUT | This pin is used by the FIFO buffer to signal to the extraction and transmission controllers that it is either full or empty. Extraction controller starts the process when the buffer signals that it is empty, and the Transmission controller starts the transmission process when the buffer signals that it is full. The state of this pin is high when empty and low when full. | High/Low |

Table 4: Description of inputs and outputs of the FIFO Buffer component.

* When in the ‘IDLE’ state, the buffer waits for the ‘storeData’ or ‘sendData’ input to go high. The buffer will transition to the ‘STORE’ state if the ‘storeData’ input is high and will transition to the ‘SEND’ state if the ‘sendData’ input goes high. The system was designed such that the two signals are never sent simultaneously from the two controllers and therefore there is no risk of race conditions occurring.
* When in the ‘STORE’ state, the buffer reads in the data into the next free space in the buffer and then increments the ‘Newest Sample’ (NS) and the ‘Next Free Space’ (FS) pointers in order to prepare to store the next sample. If the ‘Newest Sample’ pointer reaches a value of 11, the buffer will be considered full and the ‘empty’ output will go low in order to notify the controllers about the state of the buffer. After the buffer increments these pointers, it returns to the IDLE state. Each data read into the buffer is appended a unique identification based on the type of data and its origin (See Appendices A and B).
* When in the ‘SEND’ state, the buffer increments the ‘Oldest Sample’ (OS) pointer to assert a new piece of data on the output for transmission to the ‘Data Processing Block’ before returning to the ‘IDLE’ state. It is worth noting that the SEND signal is sent by the transmission controller after each transaction so that new data is loaded on the output before the next transaction. This happens because the ‘Oldest Sample’ always points to the oldest sample which is automatically asserted on the output by default and so the ‘Data Transmission Controller’ can start the first transaction before it signals to the buffer to assert the next piece of data on its output. When the buffer is commanded to assert a new piece of data on its output after the last data piece has been transmitted, the buffer resets the three pointers (FS, NS and OS) and sets the ‘empty’ output high to notify the extraction and transmission controllers that the buffer is now empty before returning to the ‘IDLE’ state.

### **Data Transmission Controller**

The ‘Data Transmission Controller’, as the name suggest, is responsible for controlling the transmission of data from the FPGA to the STM32L432 development board. This controller is also a state machine which transitions between states based on the input signals. Table 2 describes the functions that the inputs and outputs of this controller serve to drive the state transitions and the following descriptions of each state describe the operation of the controller.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin Name** | **Input/Output** | **Usage** | **Signal Type** |
| CLK | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock |
| bufferEmpty | INPUT | bufferEmpty - The FIFO buffer can signal to the controller through this pin that it is either full (0) or empty (1).   * If the buffer is empty the controller will stop the transmission process. * If the buffer is full the controller will start the transmission process. | High/Low |
| DONEtx | INPUT | Using this pin the FPGA-STM32L432 SPI master can signal the transmission controller that a transaction has been completed. | One clock cycle pulse |
| sTx | INPUT | Using this pin the FPGA-STM32 SPI Access Controller component can notify the transmission controller for a particular IMU that it has granted it access to transmit data to the STM32l432. | One clock cycle pulse |
| R2S | OUTPUT | Using this output pin the transmission controller can notify the FPGA-STM32 SPI Access Controller block that the FIFO buffer it is responsible for has been filled and therefore data is ready for transmission. Only when this pin is HIGH can the FPGA-STM32 SPI Access Controller block grant permission to the controller to transmit its data to the stm32l432. | High/Low |
| NEXT\_DAT | OUTPUT | This pin is used by the controller to command the FIFO buffer to assert the next sample to be transmitted on its output. | One clock cycle pulse |
| TRANSMIT | OUTPUT | This pin is used by the controller to command the FPGA-STM32L432 SPI master to start a transaction. This signal is passed through the FPGA-STM32 SPI Access Controller block and therefore can only reach the SPI when the controller has been granted access. | One clock cycle pulse |

Table 5: Description of inputs and outputs of the Data Transmission Controller component.

* When in the IDLE state, this component waits for the bufferEmpty signal to go low in order to start the data transmission process where it transitions to the RDY2SND state.
* When in the RDY2SND state, the controller asserts the R2S signal high to notify the FPGA-STM32 SPI Access Controller block that it is ready to transmit data. It then waits for the FPGA-STM32 SPI Access Controller block to grant it permission to transmit the data. The controller block does this by asserting the sTx input signal high for one clock cycle. When this happens, the controller transitions to the STRTtx state.
* When in the STRTtx state, the controller sends a signal using the ‘TRANSMIT’ output pin to the FPGA-STM32L432 SPI to command it to start a transaction. The controller then transitions to the WAITINGtx state.
* When in the WAITINGtx state, the controller waits for the DONEtx input to go high which the FPGA-STM32 SPI component uses to notify the controller to the transaction has been completed. When this signal is received, the controller transitions to the NXTtx state.
* When in the NXTtx state, the controller sends a signal to the FIFO buffer to assert the next data piece to be transmitted on its output and increments its internal counter to track how many samples have been transmitted. If 12 data samples have not yet been transmitted, the controller trnasitions back to the STRTtx state to transmit the next sample. If 12 data samples have been transmitted however, the controller transitions to the W4BFFRtx state.
* When it the W4BBFRtx state, the controller waits for until the buffer finishes its internal processes. When the bufferEmpty signal goes HIGH, the controller transitions to the IDLE state, otherwise it remains in the W4BFFRtx state. In this state the R2S signal is asserted low to notify the FPGA-STM32 SPI Access Controller block that there is no more data to transmit.

### **FPGA-STM32 SPI Access Controller**

This component is responsible for controlling which IMU Extraction Block is granted permission to transmit the content of its buffer. This component is necessary because the FPGA-STM32 SPI component can only transmit data from one source at a time. Figure represents the r2SIMU and the txIMU signals as being separate inputs; in reality there is one parallel data input for the r2SIMU and the txIMU and each of the IMU Extraction Blocks has its outputs connected to one of the parallel lines on the r2Simu and txIMU input. The connections in figure 2 have been represented as being single inputs such that the figure is easier to understand. This component is a state machine which transitions between states based on the input signals. Table 6 describes the functions that the inputs and outputs of this controller serve to drive the state transitions and the following descriptions of each state describe the operation of this component.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin Name** | **Input/Output** | **Usage** | **Signal Type** |
| CLK | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock |
| R2sIMU | INPUT | This input is used by the controllers to signal to the FPGA-STM32 SPI Access Controller block that their FIFO is full and that they are ready to send the data. It is a std\_logic\_vector and so each controller is connected to a separate line on the this input. | High/Low |
| txIMU | INPUT | Signals from the controllers commanding the FPGA-STM SPI to start a transaction are send to this pin. Again it is a std\_logic\_vector and so each controller connects to a separate line on this input. | One clock cycle pulse |
| dataIMUn | INPUT | Data from the buffers is sent to this pin for transmission. Each extraction block has its own connection. At this moment in time, a maximum of 6 IMUs can be connected to the FPGA-STM32 SPI Access Controller block. | Data |
| dataOut | OUTPUT | Data from one of the dataIMU inputs is asserted on this output when access is granted by the FPGA-STM32 SPI Access Controller block. This output is then fed to the FPGA-STM SPI. | Data |
| TRANSMIT | OUTPUT | Using this output, the Transmission controller of one of the extraction blocks can command the FPGA-STM SPI to start a transaction. Only the Transmission controller with the permission of the FPGA-STM32 SPI Access Controller block can assert signals on this output. | Once clock cycle pulse |
| strtTx | OUTPUT | This pin is used by the FPGA-STM32 SPI Access Controller block to send a signal that grants permission to a particular extraction block. Since the output is a std\_logic\_vector, each extraction block is sent a signal through a different channel. | Once clock cycle pulse |

Table 6: Description of inputs and outputs of the Data Transmission Controller component.

* When in the SEARCH state the FPGA-STM32 SPI Access Controller block checks each line of the r2sIMU input in a round robin fashion to see if any of the IMU Data Extraction Blocks are ready to send data. If one of the blocks is ready to send data, the FPGA-STM32 SPI Access Controller block transitions to the STRT state. There is a number that is assigned to each of the data extraction blocks inside of the FPGA-STM32 SPI Access Controller block. These numbers are used to determine which data extraction block should be granted the permission.
* When in the STRT state the FPGA-STM32 SPI Access Controller block sends a signal granting permission to the extraction block that caused this component to go into this state. It does this by sending a 1 clock cycle signal pulse to the specific data extraction block transmission controller using the strtTX(n) pin where n denotes the signal line in the parallel connector to which the Data Transmission Controller is connected to. The FPGA-STM32 SPI Access Controller block then transitions to the WAITING state. When this signal is sent, this component allows the data and the signal commanding the FPGA-STM32 SPI to transmit data that belongs to the Data Extraction Block that has been granted access, to pass through this component to reach the FPGA-STM32 SPI.
* When in the WAITING state, the FPGA-STM32 SPI Access Controller block waits for the extraction block to send all the data in its buffer. This happens when the r2sIMU(n) signal goes LOW, where n represents the signal line in the parallel connector to which the IMU Extraction Block which has been granted permission is connected to. The FPGA-STM32 SPI Access Controller block polls the signal to see if the signal has gone LOW. After this the transmission of data is finished and the signal finally goes LOW, the FPGA-STM32 SPI Access Controller block returns to the SEARCH state and continues polling the r2sIMU input from where it left.

## **Data Processing Block**

The communication side of this block is responsible for receiving data from the Data Extraction Block over SPI, converting the processed orientation data into a format that can be sent over SPI and then transmitting the data to the FPGA Wireless Communication Block using SPI. In this case the STM32L432 development board acts as the slave device to the FPGA which controls the process of transmission to the board. ‘Direct Memory Access’ (DMA) has been implemented with the Serial Peripheral Interface in order to increase the rate of transmissions. The programming language used to implement this block is C++.

### **Serial Peripheral Interface Slave Setup**

Before transmission between the two boards can occur, the Serial Peripheral Interface needs to be set-up on the STM32L432 board. To do this a number of registers have to be written to in order to do the following:

1. Enable the clock for ports containing the pins that can be configured to use SPI and also enable the SPI clock.
2. Enable the pins that can be configured to be used as SPI pins. They have to be enabled as ‘Alternate Function’ pins to allow the pins to be configured as SPI pins.
3. Configure the pins to their corresponding alternate function. Two different serial peripheral interfaces can be used on the STM32L432, SPI1 and SPI3. SPI1 is used for this project. Each pin enabled serves a different purpose for the SPI (see Appendix B).
4. Specify the frequency that the SPI is to operate at. This frequency must be the same as the frequency of the master and therefore it is set to 1MHz.
5. Specify the clock polarity (CPOL) and clock phase (CPHA). The clock polarity and clock phase must be identical to the clock polarity and phase of the master device in order to receive data correctly. Since the SPI master in VHDL is set to CPOL = 1 and CPHA = 1, the SPI in the STM32L432 will be set to this clock polarity and phase.
6. Specify the data width that the SPI is going to transmit.
7. Enable SPI1.

The following code extract demonstrates how the above steps were implemented in software. Step 7 is performed is performed after the configurating the SPI slave to work in DMA mode (see ‘Direct Memory Access SPI Setup’ section).

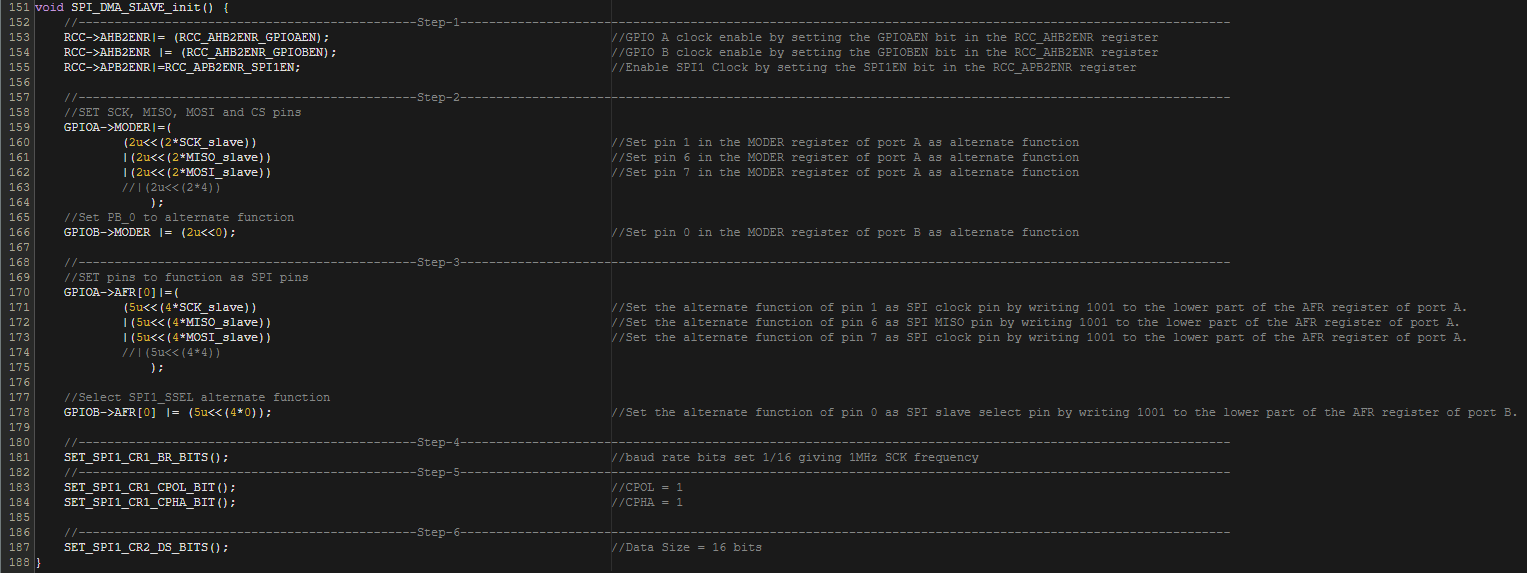


Figure 4: C++ code used to configure STM32L432 registers to set-up SPI slave on the board.

### **Direct Memory Access SPI Setup**

The decision to implement the ‘Direct Memory Access’ (DMA) method in the serial peripheral interface was made due to the ability of allowing faster transfer rates than non-DMA SPI. Achieving the highest transfer rates possible in this communication network is important for the following reasons:

* Obtaining accurate orientation angle from the gyroscope is dependant on the integration process. The accuracy of integration depends on the number of samples that are integrated per unit time; the larger the number of samples, the more accurate the orientation calculations are. The number of samples available for integration per unit time is in turn dependant on the rate of transfer.
* Data is extracted from three internal measurement units simultaneously. This data needs to be transferred to the ‘Data Processing Block’ as fast as possible to maintain a high transfer rate to maintain a good degree of accuracy in the orientation calculations for all three sets of orientation data.
* A higher transfer rate translates to a higher update rate in orientation calculations which causes the delay between the time of extracting the sample and the time the updated orientation data reaches the simulation to decrease. The smaller this delay the more real-time system becomes.

DMA achieves faster transfer rates by bypassing the central processing unit (CPU) and allowing the hardware of the microcontroller to operate the transaction processes of the SPI. This reduces the amount of processing as the process is no longer handled by software, thereby reducing the processing time of each transaction. The transaction process includes transmitting, receiving, loading received data to memory and loading data from memory for transmission. Because the CPU is no longer occupied with the transaction process, it can continue with other tasks such as orientation calculations in parallel to the SPI transactions.

To set-up DMA with SPI the following steps need to be taken:

1. Enable the DMA clock. There are two DMA controllers available on the STM32L432, DMA1 and DMA2, each containing 7 channels. DMA1 can be implemented with SPI1 and there the clock of DMA1 controller is enabled.
2. Direct memory access can be implemented on SPI1 on channels 2 and 3. Channel 2 maintains the process of receiving data and channel 3 maintains the transmission of data. These channels need to be initially disabled and the receiving and transmitting functions need to be selected for channels 2 and 3 respectively.
3. DMA channels 2 and 3 need to be configured so that they function in the desired manner.
4. Interrupts for the receiver and transmitter channels need to be enabled in order for the hardware to notify the software when the transaction has been completed. New data can be loaded for transmission and new data can be unloaded from DMA receiver.
5. The DMA SPI is to be enabled by a setting a set of registers in a sequence.

The following extract demonstrates how these steps were implemented in software.

### **Receiving Data**

Figure 5: C++ code used to configure STM32L432 registers to set-up DMA SPI on the board.

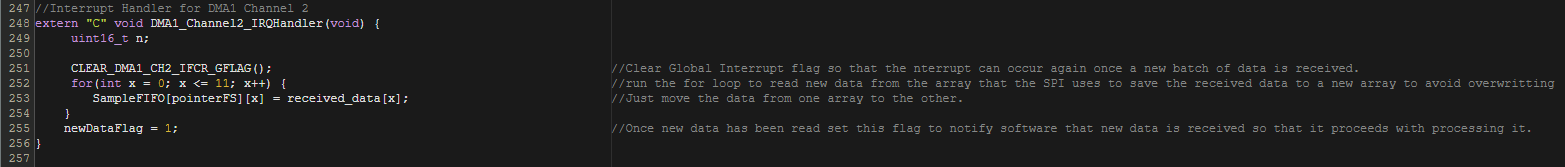
Because the SPI works in DMA mode, data is received when channel 2 of DMA1 triggers an interrupt after 12 samples have been received. The number 12 was chosen because the Data Extraction Block sends 12 samples in a batch (see Appendix A). Therefore, the interrupt only occurs when all the 12 samples are transmitted. This approach to receiving data is more efficient as there are less interrupts occurring in total (which consume some time), enabling the CPU to continue completing other tasks. When this interrupt occurs, the program enters an ‘Interrupt Service Routine’ (ISR) function in order to read the received data. This data is used to update the orientation estimates of the arm. A flag is set every time new data is received in order to notify software that new data has been received and needs to be processed. This has been done to avoid the program updating the old data and instead allow it to maintain other tasks while new data is being transmitted to the Data Processing Block. The following extract shows the how new data is read in the ISR.

Figure 6: C++ code used to receive data from the Data Extraction Block.

### **Preparing Data for Transmission**

Once the data is processed and converted into orientation angles, this data must be transmitted on request from the Processing simulation. By this point the data consist of the identification byte of the IMU that the data belongs to and the x, y and z axis orientation angles which are of the ‘float’ data type which is 32-bits wide. When the request occurs, the program runs a function which splits these floating-point numbers into 16-bit chunks to allow the data to be transmitted through SPI. Once the data is split, the total number of samples to be transmitted to the FPGA Wireless Communication Block increases to 7 samples. Table 7 depicts the order in which the data is loaded for transmission.

|  |  |
| --- | --- |
| **Description** | **Order** |
| IMU identification byte | 1 |
| x-axis orientation angle bits 31-16 | 2 |
| x-axis orientation angle bits 15-0 | 3 |
| y-axis orientation angle bits 31-16 | 4 |
| y-axis orientation angle bits 15-0 | 5 |
| z-axis orientation angle bits 31-16 | 6 |
| z-axis orientation angle bits 15-0 | 7 |

Table 7: Data to be transmitted and the order in which it is transmitted.

The request occurs when the FPGA Wireless communication block sends a pulse through the A0 pin of the STM32L432. This causes an interrupt to occur in the program, forcing it to enter an interrupt service routine in which the following variables are altered:

**dataLoadedFlag** – This variable is set to 0 to prevent the transmission interrupt service routine of the DMA from transferring new data until it has been split and loaded into an array from which the ISR extracts the data for transmission. Once the data is split and loaded this variable is set to 1. When this variable is set to one it prevents the program from constantly splitting the same data allowing the program to continue with other tasks.

**dataRequestFlag** – This flag is used set to 1 to notify the program that data has been requested. Only when this flag is set the interrupt service routine for DMA SPI transmission can load data for transmission. The orientation estimates are updated at a higher rate than the requests from the Processing simulation occur and so this flag is necessary to stop the program transmitting data when it has not been requested.

**IMU\_Data\_Pointer** – This variable is incremented or reset depending on its previous value. This pointer points to the next IMU orientation data that is to be transmitted to the FPGA Wireless Transmission Block. This is done in a round robin fashion to ensure that data from all the internal measurement units reaches the simulation.

The two flags also ensure that processes of splitting data into SPI manageable chunks and updating the orientation estimates are allowed to finish before the data is loaded for transmission. This prevents data from being from being corrupted or being sent incomplete. If the data is not loaded when a DMA transmission interrupt occurs because the splitting or update processes are not finished, the data will be loaded on the next interrupt if the processes are complete.

The extracts below depict the code used to prepare the orientation data for transmission.

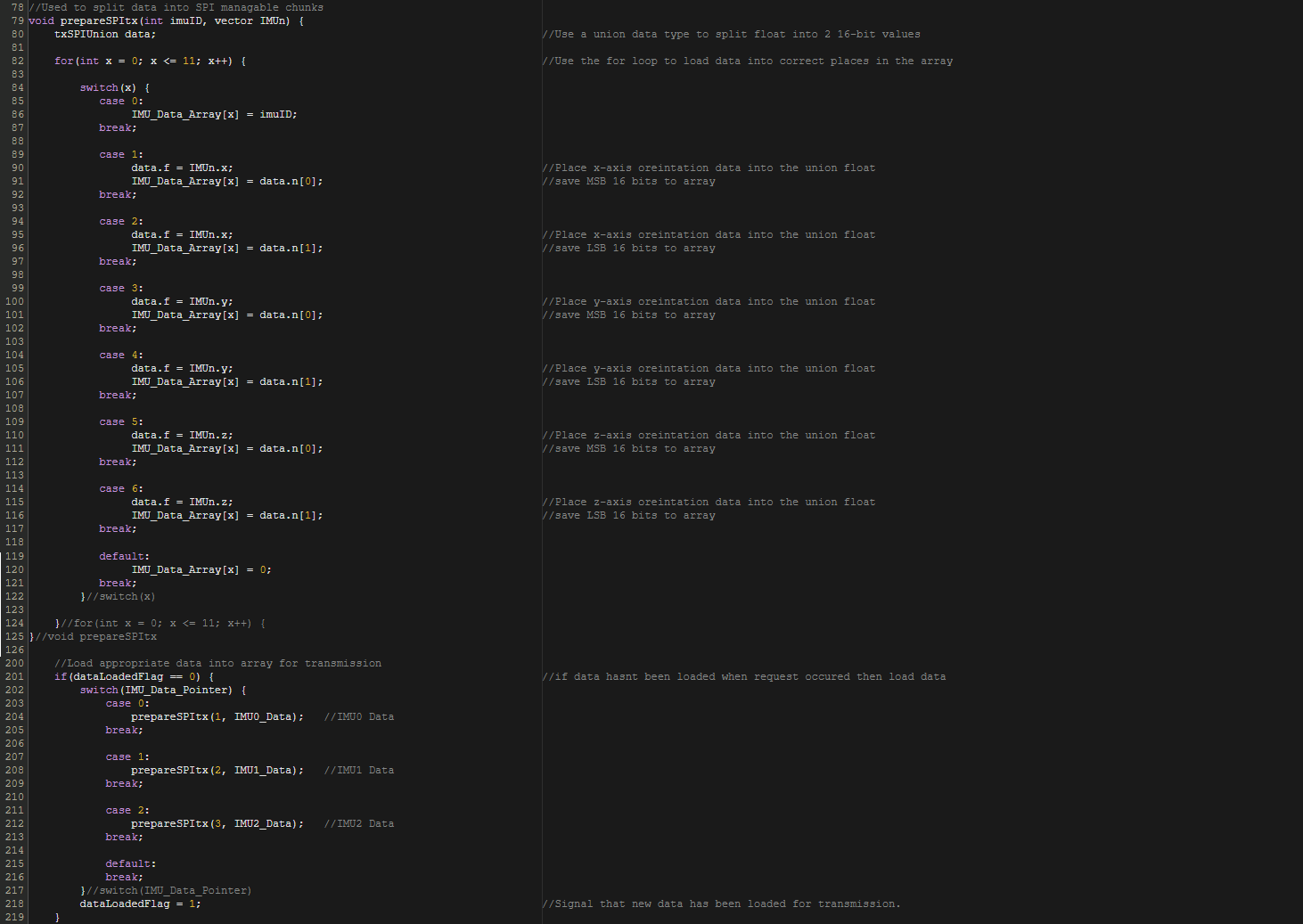


Figure 7: C++ code used to split the orientation data for transmission.

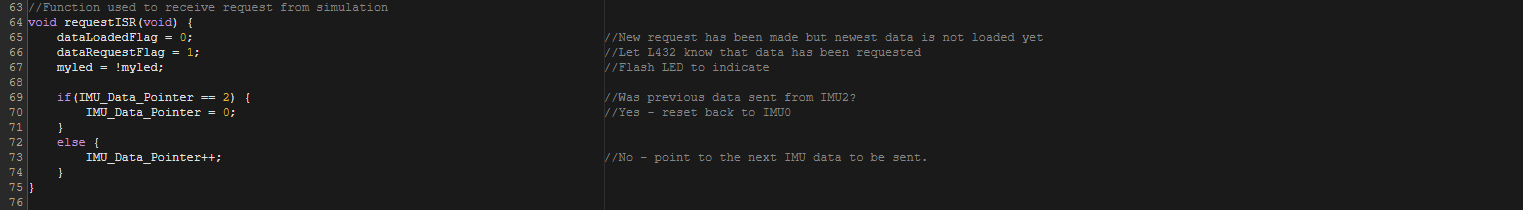


Figure 8: Interrupt service routine triggered by a data request from the Processing simulation.

### **Transmitting Data**

Once the request for new data is received and data is processed in order to allow it to be sent over SPI, data will be placed in the DMA transmission source address on the next channel 3 (the DMA SPI transmission channel) interrupt that occurs once the previous set of data has been transmitted. Once this occurs, the 7 pieces of data (see table 7) are loaded for transmission, the interrupt flag is cleared to allow the interrupt to occur again and the ‘dataRequestFlag’ is cleared to notify the program that the data request has been serviced. When the data is loaded, the first 7 spaces in the array are used, the rest are filled with zeros.

It is important to know that when the interrupt on DMA1 channel 3 occurs due to the fact that the Data Extraction Block controls the FPGA-STM32 SPI. This means that this interrupt occurs even when data is not requested by the Processing simulation as the Data Extraction Block sends data to the Data Processing Block. In such case, the array that the DMA uses to load data for transmission is filled with zeros. This is done because the FPGA Wireless Communication Block waits for the first non-zero value to be transmitted in order to start saving data to be transmitted to the Bluetooth module (assuming that the requested for data has been received in the block and has not yet been transmitted to the Data Processing Block). Sending values of zero over SPI ensures that the wireless transmission process is not triggered before the desired data is sent over SPI for wireless transmission.

The code extract below depicts how the data is transmitted to the FPGA Wireless Communication Block.

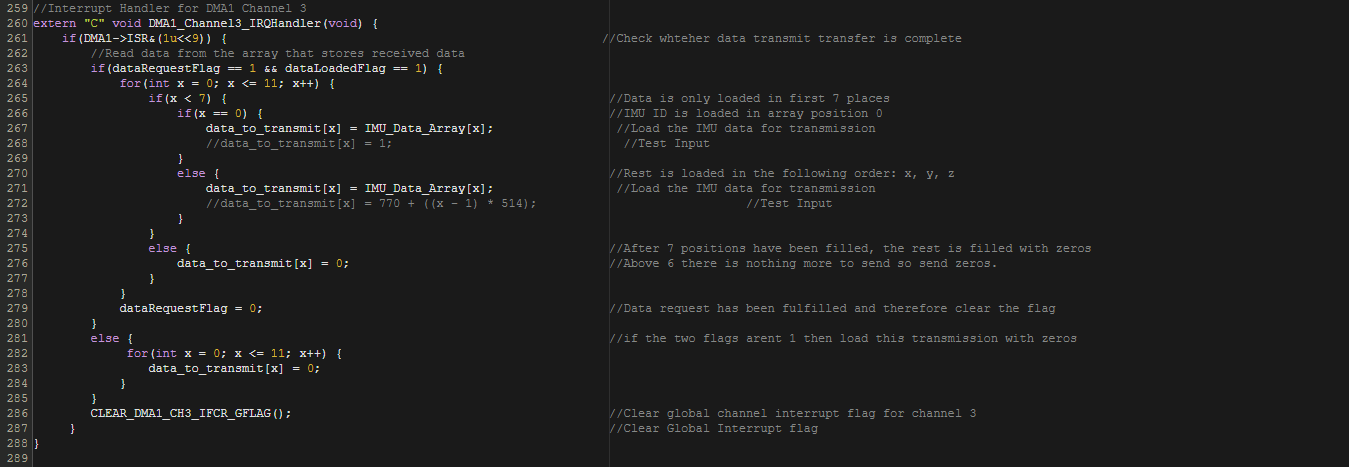


Figure 9: C++ code depicting the transmission process from the Data Processing Block to the FPGA Wireless Communication Block.

## **FPGA Wireless Communication Block**

The FPGA Wireless Communication Block is responsible for transmitting the data that it receives from the Data Processing Block to the HC-05 master module for wireless transmission. Since the data from the FPGA-STM32 SPI is in the 16-bit format and the UART protocol which is used to transmit the data to the module only accepts 8-bit data, this data will ne to be split and stored before it is transmitted to ensure that all the data is transmitted correctly.

Because the Processing simulation controls the frequency of transactions and therefore controls when a transaction is initiated, a UART receiver is added to this block in order to allow the Processing simulation send control signals which will initiate another transaction. A reset timer was added to the design due to the fact that the connection was occasionally lost which sometimes caused the system to freeze because the state machine of the receiver could not transition between states and complete its function as a result of this loss. Because the transactions are continuous and never stop, the timer sends a reset signal to the receiver if it has not responded for a prolonged period of time. Figure 4 depicts the components and how the are connected to one another.

Data 
Request Signal 
FPGA Wireless Communication Block 
IMUn 
Orientation 
Data 
16-bit data 
Reset 
Timer 
Miso 
FPGA-STM32 
SPI 
CLKO 
data 
MOSI 
16-bit data 
Serial Data Line 
8-bit data 
d Reg 
UART 
Receiver 
8-bit data 
UART 
Tramsmitter 
Signal Line 
Data Request 
Dummy Byte 
IMUn 
Orientation 
Data 
Parallel Data Line 

Figure 6: Diagram depicting the component composition and connections inside the FPGA Wireless Communication Block.

### **UART Receiver**

The UART Receiver component sends the control signal once it receives a byte from the HC-05 master module that was sent by the processing simulation. This Byte is a dummy byte and is not used in any way after it is received. Because the receiver is driven by a faster internal clock of 1 MHz and the UART speed on the Bluetooth modules has been configured to a maximum of 115200 bits per second means that this receiver and the transmitter of this block will have to wait a certain number of clock cycles for each bit of data bit that is transmitted or received. The number of clock cycles that the two components will have to wait per bit is given by the equation below.

Equation n: Equation determining the number of clock cycles required per UART bit.

Table 7 describes the functions that the inputs and outputs of this controller serve to drive the state transitions and the following descriptions of each state describe the operation of the controller.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin Name** | **Input/Output** | **Usage** | **Signal Type** |
| clk | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock Pulse |
| iRx | INPUT | Using this pin, the Bluetooth module can send data to the Receiver and therefore to the FPGA. | HIGH/LOW |
| Reset | INPUT | Using this pin, the Reset Timer can send a reset signal to the UART receiver | 1 clock cycle pulse |
| sendRequest | INPUT | Using this pin, the UART receiver can send signals to the STM32L432, the UART transmitter and the Reset Timer. | 1 clock cycle pulse |
| oRx | OUTPUT | Using this bus, the UART Receiver can output data received to the FPGA | Data |

Table 8: Description of inputs and outputs of the UART Receiver component.

* When in the IDLE state, this component waits for the iRx input to go low. This notifies the receiver to prepare for transaction. If this input remains high, the receiver will remain in the IDLE state. In this state the newly received data is asserted on the oRx output.
* When in the Rxstrb, the component waits for the Bluetooth module to transmit its half of the start bit which is equal to 5 clock cycles (rounded up from 4.5), before it transitions to the Rxdb state. The point of waiting to half of the bit transmission is so that the transmission remains synchronised and no bits are misread as the data will not be captured too early or too late. When 5 clock cycles pass, the receiver aligns itself to always receive data during half of the duration of each bit transmission and the component transitions to the Rxdb state.
* When in the Rxdb state, the receiver will read the data bits. Since it aligned itself to read in data during half transmission of each bit, it can now count to 9 before reading in each piece of data. It repeats this process 8 times since the data width of UART transmission is 8 bits. It takes a total of 72 clock cycles to capture all 8 bits. Once all 8 bits have been read, the receiver transitions to Rxstpb state.
* When in the Rxstpb state, the component receives the stop bit from the Bluetooth module. After 9 clock cycles pass, the stop bit is fully received and the receiver transitions to the sndRequest state.
* In the sndRequest state, the component sends a 1 clock cycle pulse to the STM32L432, UART transmitter and the reset timer.
  + It sends a pulse to the STM32L432 to request it to send data to the processing program.
  + It sends a pulse to the UART transmitter to notify it that data will be transmitted from the STM32L432 and therefore it should prepare to receive and transmit it to the Bluetooth module.
  + It sends a pulse to the reset timer to notify it that it is operational and did not freeze.

After the pulse is generated using the sendRequest output the receiver transitions back to the IDLE state.

The ‘RESET’ input is asynchronous which means that it is not synchronised to a clock and when a signal is sent to this input it automatically resets the receiver, not matter what state it is in. The signal will cause any counters and pointers to reset as well as reset the oRx output to logic of 1 and return this component to the IDLE state.

### **Reset Timer**

This component was added to reset the ‘UART Receiver’ in the when it occasionally locks up during connection loss. Table 8 describes the functions that the inputs and outputs of this component serve.

|  |  |  |  |
| --- | --- | --- | --- |
| Pin Name | Input/Output | Usage | Signal Type |
| CLK | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock Pulse |
| Input | INPUT | Using this pin, the UART Receiver can signal to the Reset Timer commanding it to reset its count. | One clock cycle pulse |
| Reset | OUTPUT | Using this pin, the Reset Timer can signal the UART Receiver to reset it. | One clock cycle pulse |

Table 9: Description of inputs and outputs of the Reset Timer component.

This component consists of a counter which is incremented on every clock cycle. If the UART Receiver sends the signal to the ‘Input’ pin of this component, the counter will be reset. If, however the UART Receiver does not send a signal and the counter reaches a value of 1,000,000 (equivalent of 1 second at a clock speed of 1MHz), the UART Receiver component will unresponsive and the Rest Timer will send a signal using the ‘Reset’ output pin to reset the UART Receiver.

### **UART Transmitter**

The function of the UART Transmitter is to store the data transmitted from the Data Processing Block and then transmit the data to the HC-05 Bluetooth module for wireless transmission. The incoming data is stored in first in first out fashion.

|  |  |  |  |
| --- | --- | --- | --- |
| Pin Name | Input/Output | Usage | Signal Type |
| clk | INPUT | The clock input allows the component to be synchronised to a common clock across the system. Every component in the system has an input clock. | Clock pulse |
| DoneRx | INPUT | Through this pin the FPGA-STM32 SPI can signal to the Transmitter that a transaction has been completed. | One clock cycle pulse |
| dataIn | INPUT | Through this bus the FPGA-STM32 SPI can pass data to the UART transmitter. | Data |
| Start | INPUT | Using this pin the UART receiver can signal the transmitter to start the transmission process. | One clock cycle pulse |
| oTx | OUTPUT | Using this pin the UART transmitter can send bits to the Bluetooth Module | High/Low |

Table 10: Description of inputs and outputs of the UART Transmitter component.

* When in the IDLE state, this component waits for the start signal from the UART Receiver which notifies the component that data has been requested by the Processing program. When this happens, the component transitions to the waitData state, otherwise it remains in the IDLE state. In this state the oTx sgnal is kept high in order to not signal to the Bluetooth module to start transmission. The oTx output is kept high when stopping the transaction or keeping the transmission inactive.

• When in the waitData state, the component waits for the non-zero data piece to be transmitted by the FPGA-STM32 SPI. The first data piece of the batch of data that is sent to the UART Transmitter will always be non-zero such that this component can distinguish between data that is to be transmitted and the data that is not meant to be transmitted. Additionally, the FPGA-STM32 SPI will always send zeros if there is no data to send. The first piece of data sent for transmission is always the Identification byte which helps the Processing simulation determine which IMU the data came from and therefore how to use it. Once the transmitter receives the identification byte it stores in the first position in the buffer and transitions to the readData state. The oTx output is still asserted high in this state.

* When in the readData state, the transmitter stores the rest of the orientation data. When the component receives the DoneRx signal from the FPGA-STM32 SPI, the transmitted data is placed in the next free space of the UART Transmitter buffer. There are 7 transactions in total on every request. Once all seven transactions occurred, the transmitter transitions to the PrepareTx state.
* When in the PrepareTx state, the transmitter loads the data for transmission in a first in first out fashion. The transmitter contains a pointer which points to the data in the buffer that will be transmitted. Since the buffer holds 7 pieces of data that were transmitted from the Data Processing Block, when all the samples are transmitted. The transmitter will transition back to the IDLE state. When the actual orientation data is transmitted, two transmissions occur for a sample stored in the same position. This is because the buffer stores 16-bit values and the UART protocol only allows 8-bit wide data to be transmitted. In the first transmission, the most significant byte is sent of the sample is sent and during the second transmission the least significant byte is transmitted. In the case of the sample containing the identification byte, only one transaction is necessary as this piece of data is only 8-bits wide. After the identification byte is loaded the pointer which points to the position from which data is to be loaded for transmission is incremented. In the case of the orientation data, the pointer is only incremented once the lower byte of each orientation sample is loaded for transmission. Once again, in this state the oTx output is still asserted high. Once the data is loaded, the transmitter transitions to the Txstrb state.
* When in the Txstrb state, the transmitter assertes the oTx pin low to sends the start bit to notify the Bluetooth module that a transaction is starting. As mentioned earlier, the calculated clock cycles per each bit transmitted is 9 and therefore the transmitter has to count to 9 before it starts transmitting other data. Once the counter reaches 9 clock cycles it is reset and the UART transmitter transitions to the Txdb state.
* When in the Txdb state, the byte loaded for transmission is transmitted one bit at a time. There are two counters in this state. The first is incremented on every clock cycle to track whether the bit was asserted for 9 clock cycles. The second counter keeps track of how many bits have been transmitted. Once all 8 bits have been asserted on the oTx output, each for 9 clock cycles, the transmitter transitions to the Txstpb state.
* When in the Txstpb state, the oTx output is asserted high in order to transmit the stop bit to notify the Bluetooth module that the transmission has ended. Once again, a counter tracks whether 9 clock cycles have occurred before transitioning back to the PrepareTx state.

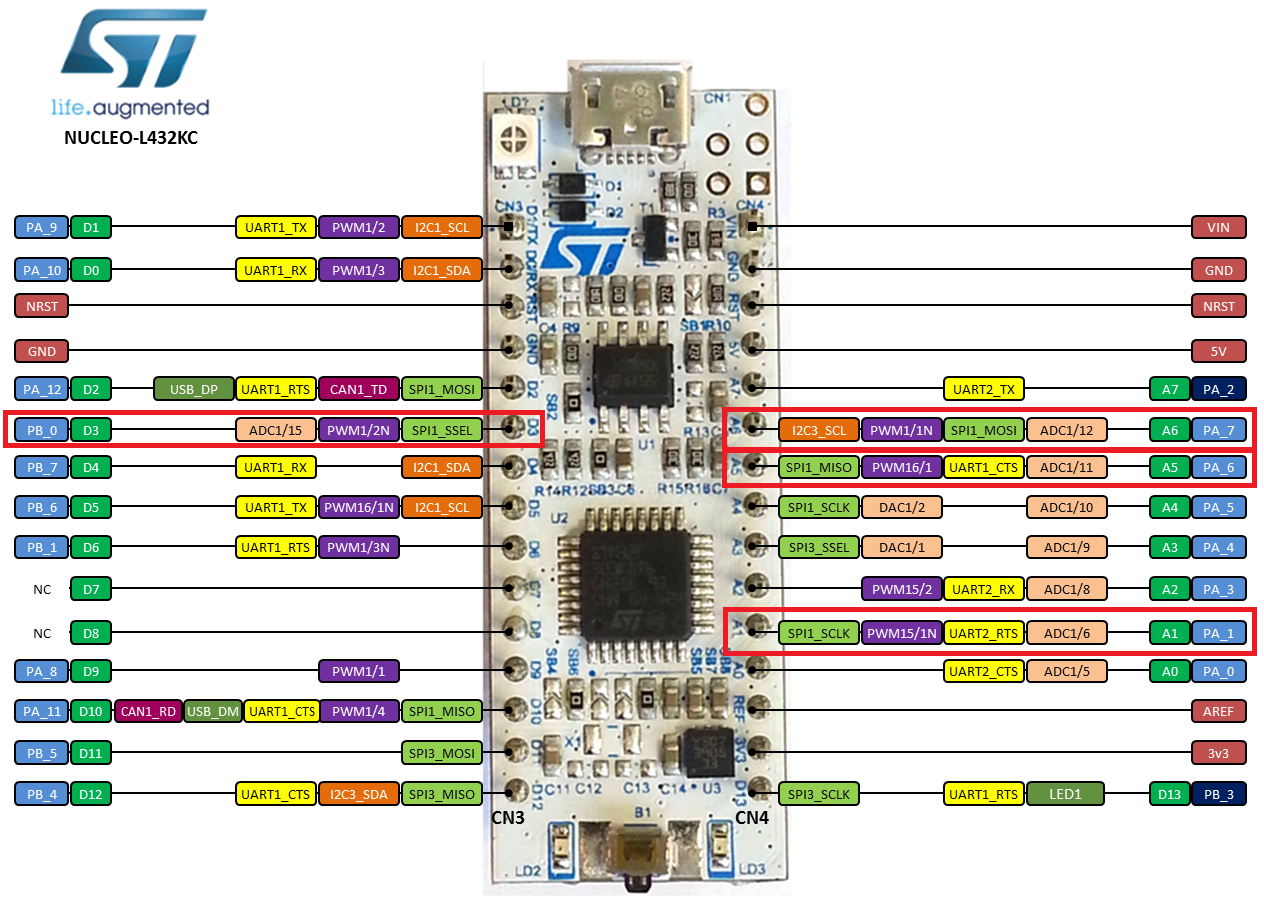
# **Appendix A**

Table of MPU9250 registers which contain accelerometer and gyroscope data and the order in which they are stored inside the ‘Data Loader’ component.

|  |  |  |
| --- | --- | --- |
| **Address Name** | **Address Value (Binary)** | **Order** |
| Accelerometer x-axis High Byte | 1011101100000000 | 1 |
| Accelerometer x-axis Low Byte | 1011110000000000 | 2 |
| Accelerometer y-axis High Byte | 1011110100000000 | 3 |
| Accelerometer y-axis Low Byte | 1011111000000000 | 4 |
| Accelerometer z-axis High Byte | 1011111100000000 | 5 |
| Accelerometer z-axis Low Byte | 1100000000000000 | 6 |
| Gyroscope x-axis High Byte | 1100001100000000 | 7 |
| Gyroscope x-axis Low Byte | 1100010000000000 | 8 |
| Gyroscope y-axis High Byte | 1100010100000000 | 9 |
| Gyroscope y-axis Low Byte | 1100011000000000 | 10 |
| Gyroscope z-axis High Byte | 1100011100000000 | 11 |
| Gyroscope z-axis Low Byte | 1100100000000000 | 12 |

# **Appendix B**

STM32L432 Pinout. Pins that can be configured as SPI pins are contained within red boxes.



# **Appendix C**

Information specifying the configuration and the information that will be sent to the Data Processing Block.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bits** | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Section** | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| **Description** | IMU ID 2 | IMU ID 1 | IMU ID 0 | X, Y or Z axis 1 | X, Y or Z axis 0 | Accel, Gyro or Mag 1 | Accel, Gyro or Mag 0 | Upper or Lower Byte | DATA BIT 7 | DATA BIT 6 | DATA BIT 5 | DATA BIT 4 | DATA BIT 3 | DATA BIT 2 | DATA BIT 1 | DATA BIT 0 |

Section 1

000 – Internal Measurement Unit 1 (Upper Arm)

001 – Internal Measurement Unit 2 (Forearm)

010 – Internal Measurement Unit 3 (Hand)

011 – N/A

100 – N/A

101 – N/A

110 – N/A

111 – N/A

Section 2

00 - x-axis

01 - y-axis

10 - z-axis

11 - N/A

Section 3

00 – Accelerometer

01 – Gyroscope

10 – Magnetometer

11 – N/A

Section 4

1 – Most Significant Byte

0 – Least Significant Byte

Section 5

This section carries the actual data extracted from the IMU.