ANALOG DEVICES

Voyager3-1Z Developers Firmware Guide

Version 0.3

Created:	6/4/2020
Author:	Sharkey, Tom
Last Modified:	16/8/2021
Modified by:	McCarthy, Jack

Overview

This guide serves as a companion document to the Voyager 1Z Firmware guide. It explains the operation of the Voyager 1Z in a greater level of detail than the original firmware guide, and is aimed towards helping developers who may seek to make changes to the operation of the mote and manager system, and to understand the system of communication between the two in a greater level of detail.

It is advised that the reader has already explored the basic documentation for the Voyager 1Z.

- WCBM-01 Firmware Guide
- CBM_Setup

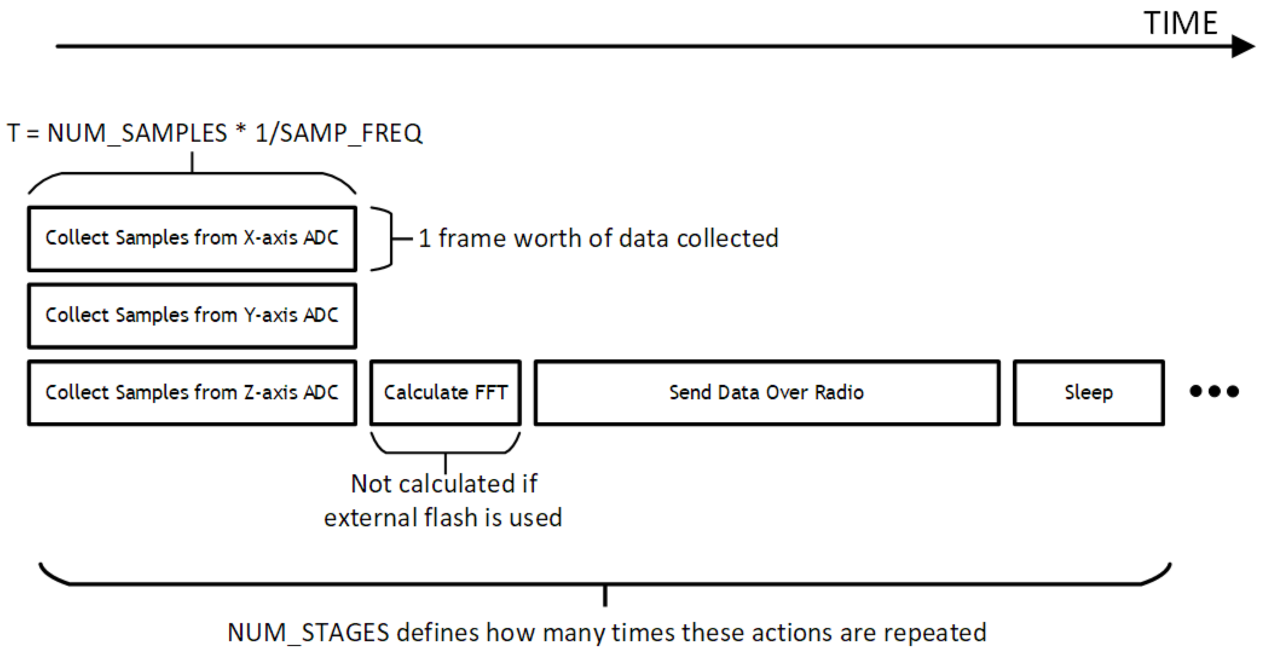
These documents are available on the Voyager-1Z Wiki.

Contents

1. High Level Overview of Operation

A summary of the high-level operation of Voyager 3 can be seen in the below figure. Note, the sequence of events shown in this figure are collectively known as a **Stage**.

When a mote is connected to the manager via the GUI, Stages are repeated until the GUI is disconnected.
If the mote is connected to the manager through the Terminal Mode, the user passes in the argument NUM_STAGES to control how many times to repeat the sequence.


TIME
T = NUM_SAMPLES * 1/SAMP_FREQ
Collect Samples from X-axis ADC
1 frame worth of data collected
Collect Samples from Y-axis ADC
Collect Samples from Z-axis ADC
Calculate FFT
Send Data Over Radio
Sleep
Not calculated if external flash is used
NUM_STAGES defines how many times these actions are repeated

Voyager 3 Timeline of Events. This Collection of Events is Called a Stage

2. Header

2.1 Framing

Normal operation from mote to manager results in a constant stream of acceleration data, separated into raw data and FFT data. Note if the number of samples per frame exceeds 1024 then a flash chip external to the microcontroller's is used and FFT calculations are no longer performed

In order to create complete "frames", such that streamed data can be easily be interpreted by the GUI, the data must be prepended with a header which contains information about the data contained within the frame. SmartMesh places a limit on the size of the data that can be sent to 90 byte packets. Each individual piece of raw data is a 16-bit integer, corresponding to a reading from the on-board ADC.

A header byte is attached to the beginning of a frame which contains information about the frame and creates a separation between frames, as demonstrated in Figure 1.

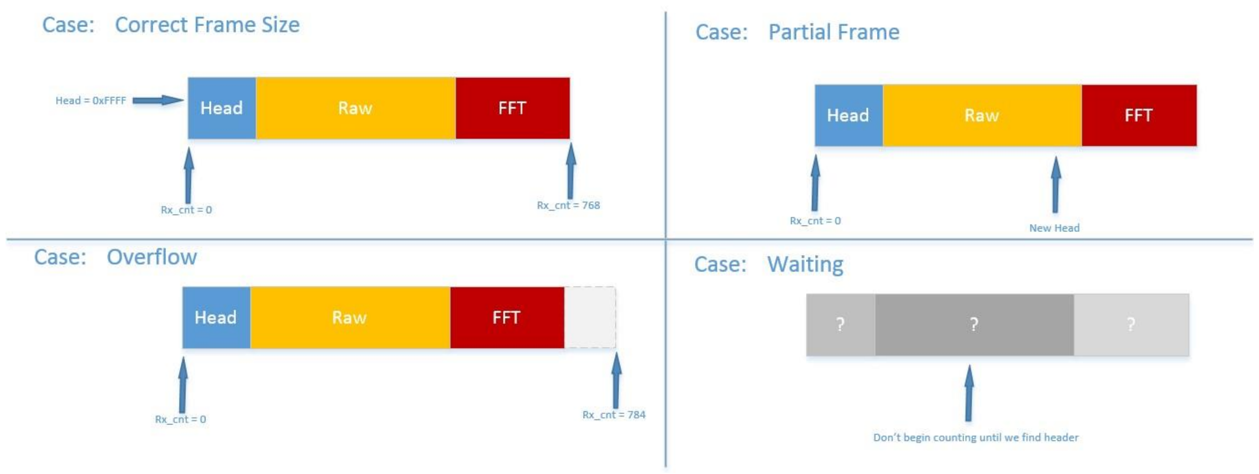
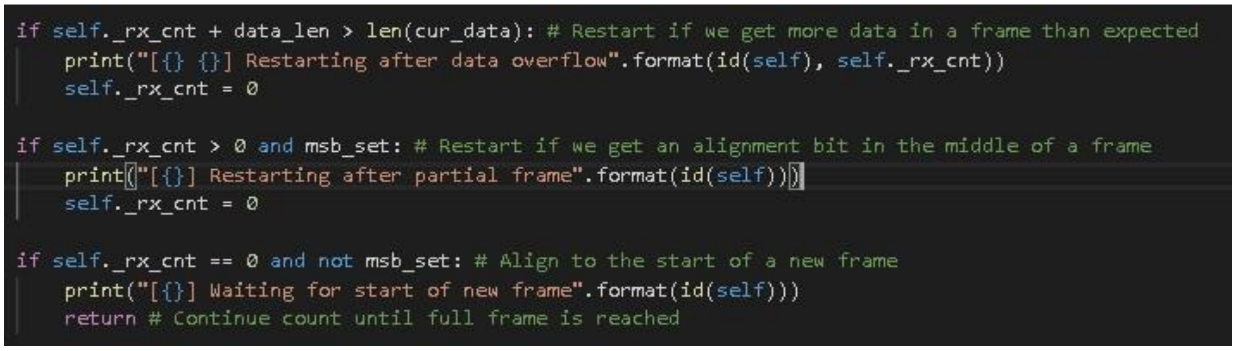


Figure 1: Forming a correct Data Frame

These checks are reflected in the code for the Python GUI, as shown in Figure 2 . Here, _rx_cnt is a variable which counts the number of bytes that have been received. This process takes place in the _handle_bytes function within the Mote class. If any of the conditions are met which indicate that the frame is not the correct size, _rx_cnt is reset to 0.



```python
if self._rx_cnt + data_len > len(cur_data): # Restart if we get more data in a frame than expected
    print("[{} {}] Restarting after data overflow".format(id(self), self._rx_cnt))
    self._rx_cnt = 0

if self._rx_cnt > 0 and msb_set: # Restart if we get an alignment bit in the middle of a frame
    print("[{}] Restarting after partial frame".format(id(self)))
    self._rx_cnt = 0

if self._rx_cnt == 0 and not msb_set: # Align to the start of a new frame
    print("[{}] Waiting for start of new frame".format(id(self)))
    return # Continue count until full frame is reached
```

Figure 2: Forming correct Data Frame (GUI)

2.2 Format of Header

The header is broken into 3 parts: The pattern marking the beginning of the header; the pattern which marks the axis of the current data; and the pattern which marks the version of the firmware.

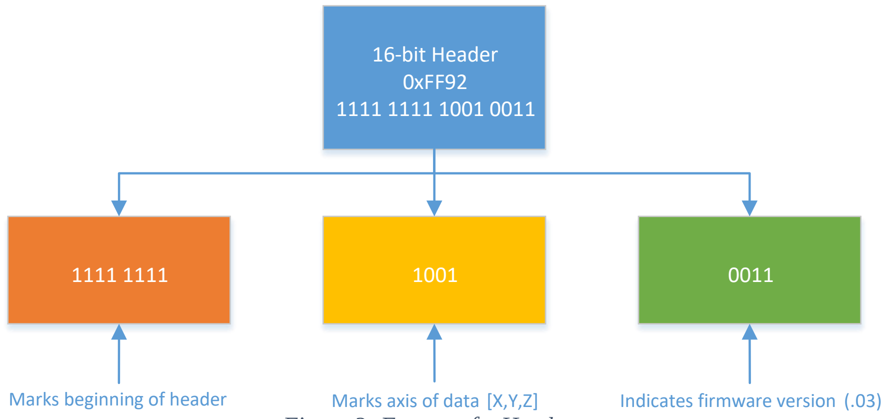

16-bit Header
0xFF92
1111 1111 1001 0011
1111 1111
1001
0011
Marks beginning of header
Marks axis of data [X,Y,Z]
Indicates firmware version (.03)

Figure 3: Format of a Header

Sample headers

Header (Hex)	Binary	Interpretation
0xFF93	0b 1111 1111 1001 0011	Header frame, x-axis, version x.03
0xFFA4	0b 1111 1111 1010 0100	Header frame, y-axis, version x.04

Table 1: Sample Headers

As the unique all-1's pattern of the top 8 bits is used to signify a header, if this pattern occurs in a non-header packet, a partial frame error will occur. However, this value will only occur in the data if the accelerometer reads a value above 99.6% of its maximum value.

2.2.1 Axis

- 0x0090 → 0000 0000 1001 0000
- 0x00A0 → 0000 0000 1010 0000 -
  0x00C0 → 0000 0000 1100 0000

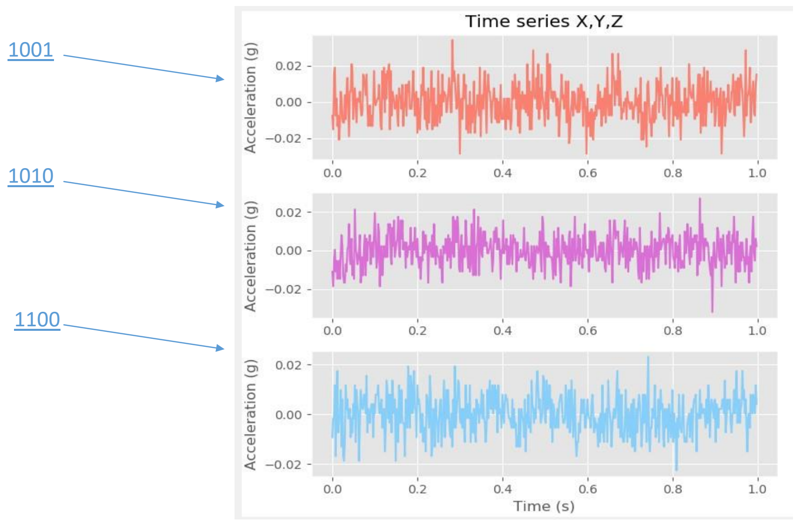


Figure 4: Axis Selection to control Plotting

The top 4-bits of the lower byte in the header signify the axis to which the data belongs, as the accelerometer records data in 3 axes. This information can be used by GUI to choose where to plot the relevant raw and FFT data. Importantly, the GUI allows for the sending of specific axes of data.

2.2.2 Version

Examples

- 0x0000 0000 0000 0002
- 0x0000 0000 0000 0003

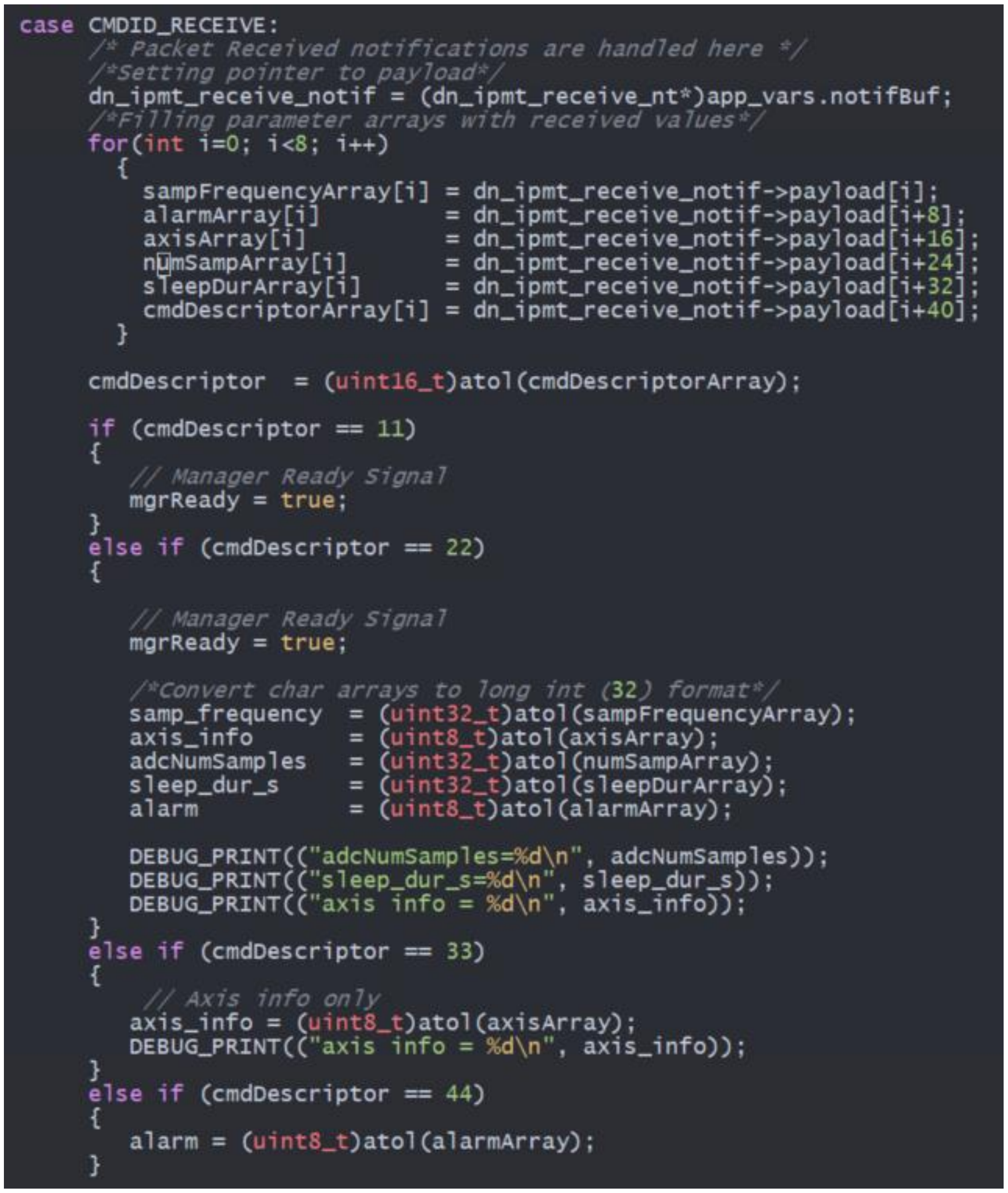
The final 4 bits of the header are reserved for the version number of the currently running firmware. This is used as a means of confirming correct operation between firmware and GUI, which must be updated in tandem for the correct operation of the system. This version system is simplistic in order to communicate the version of the program in the least number of bits possible, and so keep the header overhead small. The version number sent is simply the two numbers after the decimal point for the version (1.03). As only four bits are available, this version number must not exceed .15.

N.B. The version of the firmware and the GUI must be updated as changes are made to the system. Both versions are constants defined at the top of the GUI and in the SmartMesh_RF_Cog.h file of the firmware respectively. If the versions of GUI and firmware do not match, an error is issued to the user in the GUI command window.

3. Downstream communication

In order to control the operation of individual motes, a command packet must be sent to each mote from the GUI. This command packet is used whenever the sampling frequency must be updated for FFT calculation, when an alarm must be triggered, and to communicate which axis information the GUI wants to receive.

Any update to mote operation is sent in the form of a command message, which triggers a specific callback in the motes firmware, as shown in Figure 5.



```
case CMDID_RECEIVE:
      /* Packet Received notifications are handled here */
      /*Setting pointer to payload*/
      dn_ipmt_receive_notif = (dn_ipmt_receive_nt*)app_vars.notifBuf;
      /*Filling parameter arrays with received values*/
      for(int i=0; i<8; i++)
        {
          sampFrequencyArray[i] = dn_ipmt_receive_notif->payload[i];
          alarmArray[i]         = dn_ipmt_receive_notif->payload[i+8];
          axisArray[i]          = dn_ipmt_receive_notif->payload[i+16];
          numSampArray[i]       = dn_ipmt_receive_notif->payload[i+24];
          sleepDurArray[i]      = dn_ipmt_receive_notif->payload[i+32];
          cmdDescriptorArray[i] = dn_ipmt_receive_notif->payload[i+40];
        }

      cmdDescriptor  = (uint16_t)atol(cmdDescriptorArray);

      if (cmdDescriptor == 11)
      {
         // Manager Ready Signal
         mgrReady = true;
      }
      else if (cmdDescriptor == 22)
      {

         // Manager Ready Signal
         mgrReady = true;

         /*Convert char arrays to long int (32) format*/
         samp_frequency   = (uint32_t)atol(sampFrequencyArray);
         axis_info        = (uint8_t)atol(axisArray);
         adcNumSamples    = (uint32_t)atol(numSampArray);
         sleep_dur_s      = (uint32_t)atol(sleepDurArray);
         alarm            = (uint8_t)atol(alarmArray);

         DEBUG_PRINT(("adcNumSamples=%d\n", adcNumSamples));
         DEBUG_PRINT(("sleep_dur_s=%d\n", sleep_dur_s));
         DEBUG_PRINT(("axis info = %d\n", axis_info));
      }
      else if (cmdDescriptor == 33)
      {
          // Axis info only
         axis_info = (uint8_t)atol(axisArray);
         DEBUG_PRINT(("axis info = %d\n", axis_info));
      }
      else if (cmdDescriptor == 44)
      {
         alarm = (uint8_t)atol(alarmArray);
      }
```

Figure 5: Mote Command Received

3.1 Sampling Frequency

The sampling frequency is updated by the user and used to control the time between ADC acquisitions. The updated sampling frequency sent by the user is used to adjust a timer controlling ADC readings. The sampling frequency can be programmed with a minimum granularity of 1Hz.

Examples
Sampling Frequency (f_S = 1000Hz)

$$Acquistion\ Time\ (t) = \frac{1}{f_S}$$

3.2 Number of ADC Samples

The number of samples to be collected per enabled axis i.e., the number of ADC data samples per frame can be controlled by the user from version 0.5 onwards.

Note, if a number larger than 1024 is selected then the data is stored on a flash memory chip that is external to the microcontroller and therefore **FFT calculations will not be performed**.

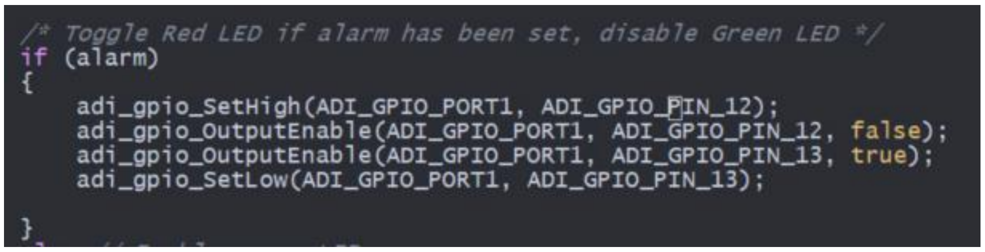
3.3 Sleep Duration

This controls the duration that the microcontroller will enter a low power sleep mode after it has finished transmitting all of the collected ADC data and FFT calculations i.e., after a Stage has been completed. This enables more realistic use-cases where the mote can wake up at user defined intervals and collect a user-defined number of samples before returning to a low power state.

The microcontroller's real-time clock is used to wake the microcontroller up after the selected number of seconds. The minimum granularity of this clock is one second.

3.4 Alarm Triggered

The first check that occurs after a command has been received in the firmware is to see if the alarm variable has been set. An alarm command is sent from the GUI to the Mote that triggered it, if a Mote value such as "Peak" acceleration exceeds a user-set value. The LED used is the same green LED that blinks to indicate a successful network has been developed between a mote and a manager. The green LED is disabled on Port 1, Pin 12, and the red LED is enabled on Port 1, Pin 13, as shown in Figure 6.



```
/* Toggle Red LED if alarm has been set, disable Green LED */
if (alarm)
{
    adi_gpio_SetHigh(ADI_GPIO_PORT1, ADI_GPIO_PIN_12);
    adi_gpio_OutputEnable(ADI_GPIO_PORT1, ADI_GPIO_PIN_12, false);
    adi_gpio_OutputEnable(ADI_GPIO_PORT1, ADI_GPIO_PIN_13, true);
    adi_gpio_SetLow(ADI_GPIO_PORT1, ADI_GPIO_PIN_13);

}
```

Figure 6: Setting Red LED (Smartmesh_RF_Cog.c)

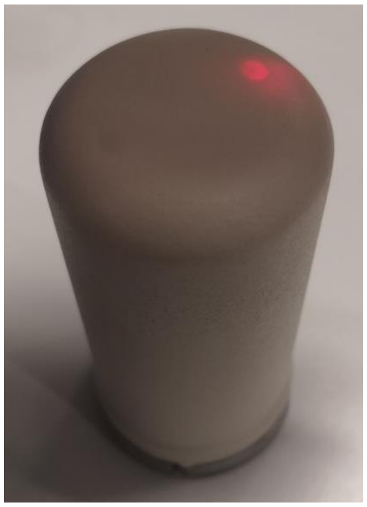


Figure 7: Alarm Triggered

If the alarm variable is not set, the green LED returns to normal operation, blinking if a network still exists between mote and manager.

3.5 Axes Received

Based on the axes that the user is interested in, only specific data is communicated by each mote. This can be any combination of the X, Y and Z axes. As normal operation for the mote is to send all 3 axes in order, changing which axis are sent simply adjusts the pointer to the data, as shown in Figure 8.

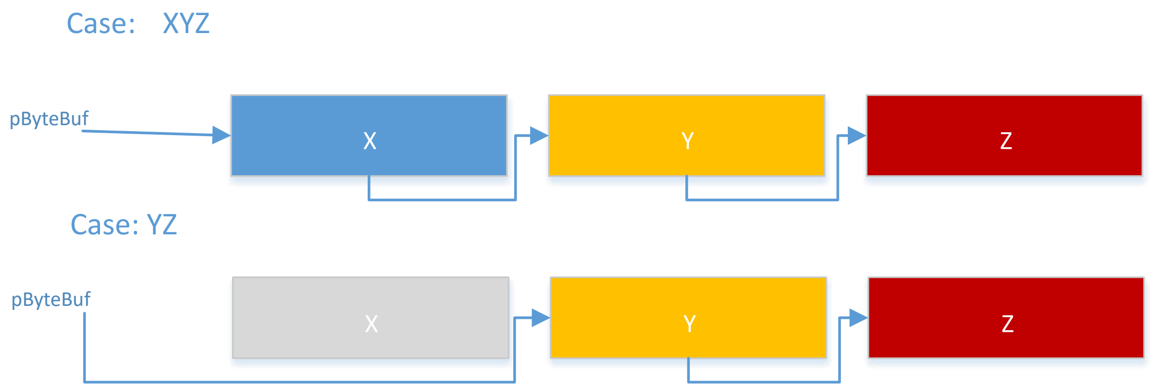
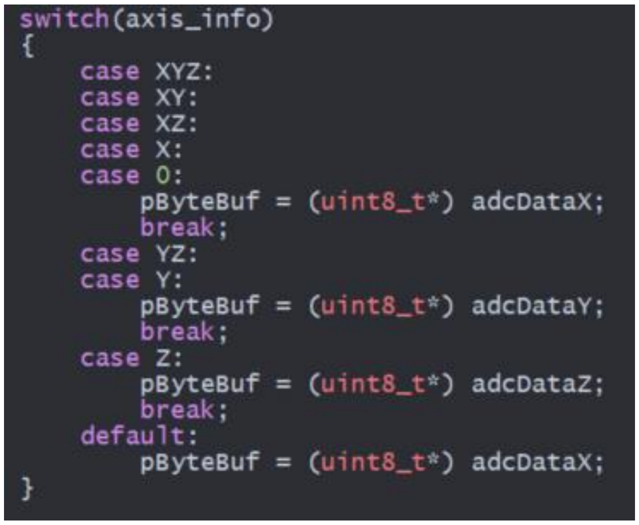


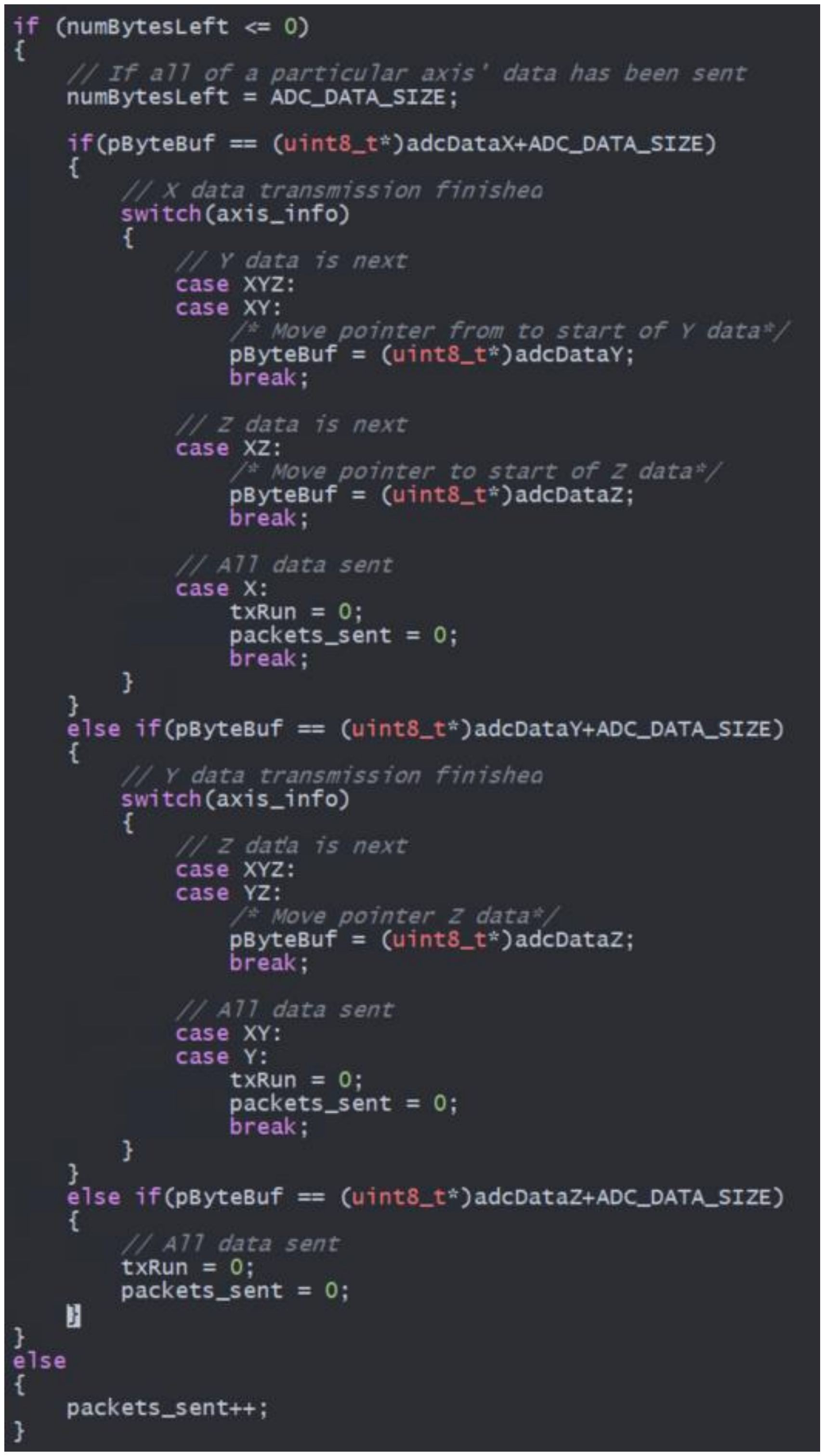
Figure 8: Sending User Defined Axes

As normal operation for the mote is to send all three axis in the order X → Y → Z, variations to this operation simply control the buffer to begin at, and the buffers to exclude, as shown in Figure 9. When all buffers have been sent, the variable txRun is set to 0 signalling the end of transmission to the data handling state machine.



```
switch(axis_info)
{
    case XYZ:
    case XY:
    case XZ:
    case X:
    case 0:
        pByteBuf = (uint8_t*) adcDataX;
        break;
    case YZ:
    case Y:
        pByteBuf = (uint8_t*) adcDataY;
        break;
    case Z:
        pByteBuf = (uint8_t*) adcDataZ;
        break;
    default:
        pByteBuf = (uint8_t*) adcDataX;
}
```

Figure 9: Select Axis Buffer to Begin Sending



```
if (numBytesLeft <= 0)
{
    // If all of a particular axis' data has been sent
    numBytesLeft = ADC_DATA_SIZE;

    if(pByteBuf == (uint8_t*)adcDataX+ADC_DATA_SIZE)
    {
        // X data transmission finished
        switch(axis_info)
        {
            // Y data is next
            case XYZ:
            case XY:
                /* Move pointer from to start of Y data*/
                pByteBuf = (uint8_t*)adcDataY;
                break;

            // Z data is next
            case XZ:
                /* Move pointer to start of Z data*/
                pByteBuf = (uint8_t*)adcDataZ;
                break;

            // All data sent
            case X:
                txRun = 0;
                packets_sent = 0;
                break;
        }
    }
    else if(pByteBuf == (uint8_t*)adcDataY+ADC_DATA_SIZE)
    {
        // Y data transmission finished
        switch(axis_info)
        {
            // Z data is next
            case XYZ:
            case YZ:
                /* Move pointer Z data*/
                pByteBuf = (uint8_t*)adcDataZ;
                break;

            // All data sent
            case XY:
            case Y:
                txRun = 0;
                packets_sent = 0;
                break;
        }
    }
    else if(pByteBuf == (uint8_t*)adcDataZ+ADC_DATA_SIZE)
    {
        // All data sent
        txRun = 0;
        packets_sent = 0;
    }
}
else
{
    packets_sent++;
}
```

Figure 10: Selecting Axis Buffers to Exclude

3.6 Updating Parameters

Mote variables such as sampling frequency, number of samples per frame and sleep duration are updated within the “NEW_PARAMS” state of the data handling state machine in main_prog.c. This prevents errors that can occur when a firmware variable is updated instantly upon receiving a downstream command. By defining a specific state for updating these variables, the time at which these updates occur is known to the user, preventing timing-related errors.

As the red LED that signals an alarm (e.g. the mote exceeding a certain value peak acceleration) can be updated at any time, the check for this alarm flag occurs directly after the setting of downstream variables in SmartMesh_RF_cog.c.