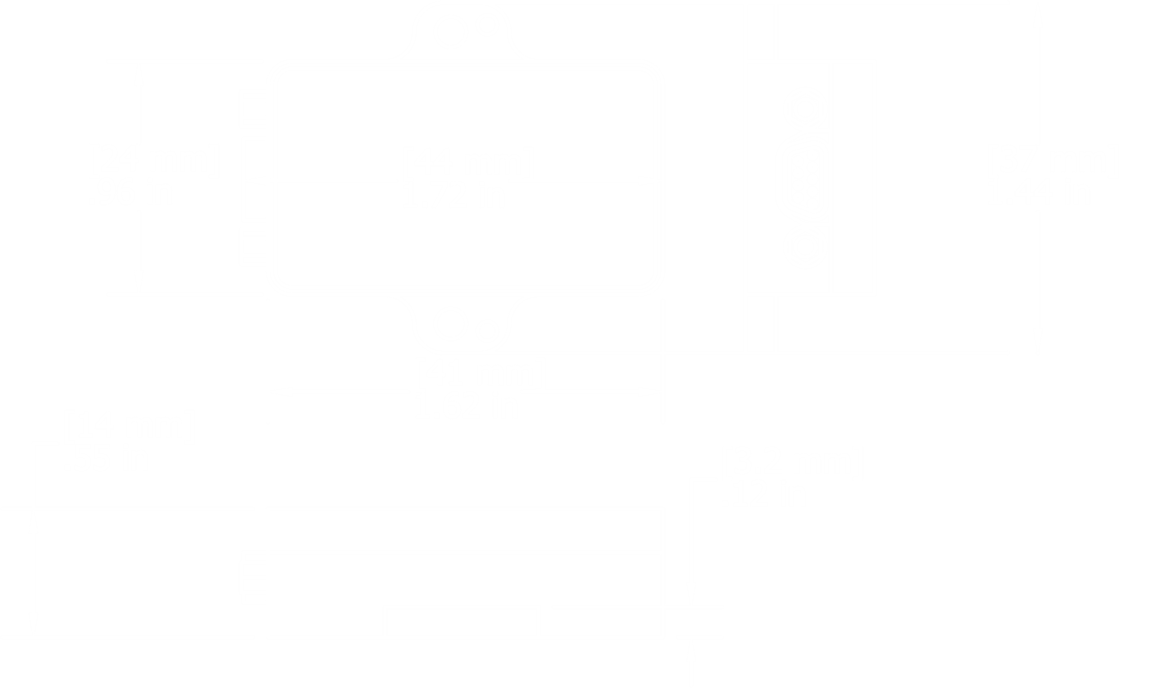
LORD MANUAL

08

**Fall**

C MIP SDK

User’s Guide



LORD-Tech-Note-logo

©2014 LORD Corporation

MicroStrain® Sensing Systems

459 Hurricane Lane

Suite 102

Williston, VT 05495

United States of America

Phone: 802-862-6629

Fax: 802-863-4093

www.microstrain.com

sensing\_support@LORD.com

**REVISED: May 30, 2014**

LORD-Tech-Note-logo

# Table of Contents

[Table of Contents 3](#_Toc389222055)

[1. MIP Packet Overview 5](#_Toc389222056)

[Structure 5](#_Toc389222057)

[Payload Length Range 6](#_Toc389222058)

[Checksum Range 6](#_Toc389222059)

[16-bit Fletcher Checksum Algorithm (C language) 6](#_Toc389222060)

[MIP Packet Examples 7](#_Toc389222061)

[Command Overview 9](#_Toc389222062)

[Example “Ping” Command Packet: 9](#_Toc389222063)

[Example “Ping” Reply Packet: 9](#_Toc389222064)

[Data Overview 10](#_Toc389222065)

[Example Data Packet: 10](#_Toc389222066)

[2. C MIP SDK Overview 12](#_Toc389222067)

[Introduction 12](#_Toc389222068)

[Directory Structure 12](#_Toc389222069)

[SDK Documentation 12](#_Toc389222070)

[SDK Configuration 14](#_Toc389222071)

[Basic Functions Assumed to be Available 14](#_Toc389222072)

[Target-Specific User Functions 15](#_Toc389222073)

[Utility Functions 15](#_Toc389222074)

[3. C MIP SDK Interface Overview 16](#_Toc389222075)

[Interface Overview 16](#_Toc389222076)

[Parser Description 16](#_Toc389222077)

[Parser Callback Functions 17](#_Toc389222078)

[Command-Response Handling 18](#_Toc389222079)

[Standard Interface Use 18](#_Toc389222080)

[4. Included MIP SDK Programming Examples 21](#_Toc389222081)

[GX3-35 Unit Test Example 21](#_Toc389222082)

[GX3-45 Unit Test Example 21](#_Toc389222083)

[GX4-15 Unit Test Example 21](#_Toc389222084)

[GX4-25 Unit Test Example 21](#_Toc389222085)

[GX4-45 Unit Test Example 22](#_Toc389222086)

# 1. MIP Packet Overview

## Structure

Commands and Data are sent and received as fields in the MicroStrain “MIP” packet format. Below is the general definition of the structure:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Payload | | | Checksum | |
| SYNC1  “u” | SYNC2  “e” | Descriptor Set byte | Payload  Length byte | Fields | | | MSB | LSB |
| 0x75 | 0x65 | <desc set selector> | *k1+k2+…kn* | MIP Field 1 length = *k1* | ... | MIP Field *n*  length = *kn* | 0x*MM* | 0x*LL* |

|  |  |  |
| --- | --- | --- |
| Field Header | | Field Data |
| Field Length byte | Field Descriptor byte | Field Data |
| *kn* | <descriptor> | <*kn-2* bytes of data> |

The packet always begins with the start-of-packet sequence “ue” (0x75, 0x65). The “Descriptor Set” byte in the header specifies which command or data set is contained in fields of the packet. The payload length byte specifies the sum of all the field length bytes in the payload section.

### Payload Length Range

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Packet Header | | | | Payload | Checksum | |
| SYNC1 | SYNC2 | Descriptor Set | Payload  Length | MIP Data Fields | MSB | LSB |
|  | | | | <--------------*Payload Length Range* ------------> |  |  |

The payload section can be empty or can contain one or more fields. Each field has a length byte and a descriptor byte. The field length byte specifies the length of the entire field including the field length byte and field descriptor byte. The descriptor byte specifies the command or data that is contained in the field data. The descriptor can only be from the set of descriptors specified by the descriptor set byte in the header. The field data can be anything but is always rigidly defined. The definition of a descriptor is fundamentally described in a “.h” file that corresponds to the descriptor set that the descriptor belongs to.

MicroStrain provides a “MIP Packet Builder” utility to simplify the construction of a MIP packet. Most commands will have a single field in the packet, but multiple field packets are possible. Extensive examples complete with checksums are given in the command reference section of each device DCP.

### Checksum Range

The checksum is a 2 byte Fletcher checksum and encompasses all the bytes in the packet:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Packet Header | | | | Payload | Checksum | |
| SYNC1 | SYNC2 | Descriptor Set | Payload  Length | MIP Data Fields | MSB  (byte1) | LSB  (byte2) |
| <-------------------------------------------- *Checksum Range* --------------------------------------> | | | | |  |  |

### 16-bit Fletcher Checksum Algorithm (C language)

checksum\_byte1 = 0x00;  
checksum\_byte2 = 0x00;

for(i=0; i<checksum\_range; i++)

{  
 checksum\_byte1 += mip\_packet[i];  
 checksum\_byte2 += checksum\_byte1;  
}  
  
checksum = ((u16) checksum\_byte1 << 8) + (u16) checksum\_byte2;

### MIP Packet Examples

The MIP packet “wrapper” consists of a four byte header and two byte checksum footer:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Packet Payload | | | Checksum | |
| *SYNC1*  *“u”* | *SYNC2*  *“e”* | *Descriptor Set byte* | *Payload*  *Length byte* | *Field Length byte* | *Field Descriptor byte* | *Field Data* | *MSB* | *LSB* |
| **0x75** | **0x65** | **0x80** | **0x0E** | **0x0E** | **0x03** | **0x3E 7A 63 A0**  **0xBB 8E 3B 29**  **0x7F E5 BF 7F** | **0x83** | **0xE1** |

Payload Length byte. This specifies the length of the packet payload. The packet payload may contain one or more fields and thus this byte also represents the sum of the lengths of all the fields in the payload.

Descriptor Set. Descriptors are grouped into different sets. The value 0x80 identifies this packet as an AHRS data packet. Fields in this packet will be from the AHRS data descriptor set.

Start of Packet (SOP) “sync” bytes. These are the same for every MIP packet and are used to identify the start of the packet.

2 byte Fletcher checksum of all the bytes in the packet.

The packet payload section contains one or more fields. Fields have a length byte, descriptor byte, and data. The diagram below shows a packet payload with a single field.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Packet Payload | | | Checksum | |
| *SYNC1*  *“u”* | *SYNC2*  *“e”* | *Descriptor Set byte* | *Payload*  *Length byte* | *Field Length byte* | *Field Descriptor byte* | *Field Data* | *MSB* | *LSB* |
| **0x75** | **0x65** | **0x80** | **0x0E** | **0x0E** | **0x06** | **0x3E 7A 63 A0**  **0xBB 8E 3B 29**  **0x7F E5 BF 7F** | **0x86** | **0x08** |

Field Length byte. This represents a count of all the bytes in the field including the length byte, descriptor byte and field data.

Descriptor byte. This byte identifies the contents of the field data. This descriptor indicates that the data is a mag vector (set: 0x80, descriptor: 0x06)

Field data. The length of the data is Field Length – 2. This data is 12 bytes long (14 – 2) and represents the floating point magnetometer vector value from the AHRS data set.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Packet Payload (2 fields) | | | | | | Checksum | | |
| *SYNC1*  *“u”* | *SYNC2*  *“e”* | *Descriptor Set* | *Payload*  *Length* | *Field1 Len* | *Field1 Descriptor* | *Field1 Data* | *Field2 Len* | *Field2 Descriptor* | *Field2 Data* | | *MSB* | *LSB* |
| **0x75** | **0x65** | **0x80** | **0x1C** | **0x0E** | 0x05 | 0x3E 7A 63 A0  0xBB 8E 3B 29  0x7F E5 BF 7F | **0x0E** | 0x06 | 0x3E 7A 63 A0  0xBB 8E 3B 29  0x7F E5 BF 7F | | **0xB1** | **0x1E** |

Below is an example of a packet payload with two fields (gyro vector and mag vector). Note the payload length byte of 0x1C which is the sum of the two field length bytes 0x0E + 0x0E:

## Command Overview

The basic command sequence begins with the host sending a command to the device. A command packet contains a field with the command value and any command arguments.

The device responds by sending a reply packet. The reply contains at minimum an ACK/NACK field. If any additional data is included in a reply, it appears as a second field in the packet.

### Example “Ping” Command Packet:

Below is an example of a “Ping” command packet from the Base command set. A “Ping” command has no arguments. Its function is to determine if a device is present and responsive:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Packet Payload | | | Checksum | |
| *SYNC1*  *“u”* | *SYNC2*  *“e”* | *Descriptor Set byte* | *Payload*  *Length byte* | *Field Length byte* | *Field Descriptor byte* | *Field Data* | *MSB* | *LSB* |
| **0x75** | **0x65** | **0x01** | **0x02** | **0x02** | **0x01** | *N/A* | **0xE0** | **0xC6** |

Copy-Paste version: “7565 0102 0201 E0C6”

The packet header has the “ue” starting sync bytes characteristic of all MIP packets. The descriptor set byte (0x01) identifies the data as being from the Base command set. The length of the payload portion is 2 bytes. The payload portion of the packet consists of one field. The field starts with the length of the field which is followed by the descriptor byte (0x01) of the field. The field descriptor value *is* the command value. Here the descriptor identifies the command as the “Ping” command from the Base command descriptor set. There are no parameters associated with the ping command, so the field data is empty. The checksum is a two byte Fletcher checksum.

### Example “Ping” Reply Packet:

The “Ping” command will generate a reply packet from the device. The reply packet will contain an ACK/NACK field. The ACK/NACK field contains an “echo” of the command byte plus an error code. An error code of 0 is an “ACK” and a non-zero error code is a “NACK”:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Packet Payload | | | Checksum | |
| *SYNC1*  *“u”* | *SYNC2*  *“e”* | *Descriptor Set byte* | *Payload*  *Length byte* | *Field Length byte* | *Field Descriptor byte* | *Field Data: 2 bytes* | *MSB* | *LSB* |
| **0x75** | **0x65** | **0x01** | **0x04** | **0x04** | **0xF1** | Command echo**: 0x01**  Error code**: 0x00** | **0xD5** | **0x6A** |

*Copy-Paste version: “7565 0104 04F1 0100 D56A”*

The packet header has the “ue” starting sync bytes characteristic of all MIP packets. The descriptor set byte (0x01) identifies the payload fields as being from the Base command set. The length of the payload portion is 4 bytes. The payload portion of the packet consists of one field. The field starts with the length of the field which is followed by the descriptor byte (0xF1) of the field. The field descriptor byte identifies the reply as the “ACK/NACK” from the Base command descriptor set. The field data consists of an “echo” of the original command (0x01) followed by the error code for the command (0x00). In this case the error is zero, so the field represents an “ACK”. Some examples of non-zero error codes that might be sent are “timeout”, “not implemented”, and “invalid parameter in command”. The checksum is a two byte Fletcher checksum.

The ACK/NACK descriptor value (0xF1) is the same in all descriptor sets. The value belongs to a set of reserved global descriptor values.

The reply packet may have additional fields that contain information in reply to the command. For example, requesting *Device Status* will result in a reply packet that contains two fields in the packet payload: an ACK/NACK field and a device status information field.

## Data Overview

Data packets are generated by the device. When the device is powered up, it may be configured to immediately stream data packets out to the host or it may be “idle” and waiting for a command to either start continuous data or to get data by “polling” (one data packet per request). Either way, the data packet is generated by the device in the same way.

### Example Data Packet:

Below is an example of a MIP data packet which has one field that contains the scaled accelerometer vector.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Header | | | | Packet Payload | | | Checksum | |
| *SYNC1*  *“u”* | *SYNC2*  *“e”* | *Descriptor Set byte* | *Payload*  *Length byte* | *Field Length byte* | *Field Descriptor byte* | *Field Data: Accel vector (12 bytes, 3 float – X, Y, Z)* | *MSB* | *LSB* |
| **0x75** | **0x65** | **0x80** | **0x0E** | **0x0E** | **0x04** | **0x3E 7A 63 A0**  **0xBB 8E 3B 29**  **0x7F E5 BF 7F** | **0x92** | **0xC0** |

Copy-Paste version: “7565 800E 0E03 3E7A 63A0 BB8E 3B29 7FE5 BF7F 83E1”

The packet header has the “ue” starting sync bytes characteristic of all MIP packets. The descriptor set byte (0x80) identifies the payload field as being from the AHRS data set. The length of the packet payload portion is 14 bytes (0x0E). The payload portion of the packet starts with the length of the field. The field descriptor byte (0x01) identifies the field data as the scaled accelerometer vector from the AHRS data descriptor set. The field data itself is three single precision floating point values of 4 bytes each (total of 12 bytes) representing the X, Y, and Z axis values of the vector. The checksum is a two byte Fletcher checksum.

The format of the field data is fully and unambiguously specified by the descriptor. In this example, the field descriptor (0x04) specifies that the field data holds an array of three single precision IEEE-754 floating point numbers in big-endian byte order and that the values represent units of “g’s” and the order of the values is X, Y, Z vector order. Any other specification would require a different descriptor.

Each packet can contain any combination of data quantities from the same data descriptor set (e.g. any combination of GPS data OR any combination of AHRS data OR and combination of NAV data – you cannot combine different data sets in the same packet).

Data polling commands generate two individual reply packets: An ACK/NACK packet and a data packet. Enable/Disable continuous data commands generate an ACK/NACK packet followed by the continuous stream of data packets.

# 2. C MIP SDK Overview

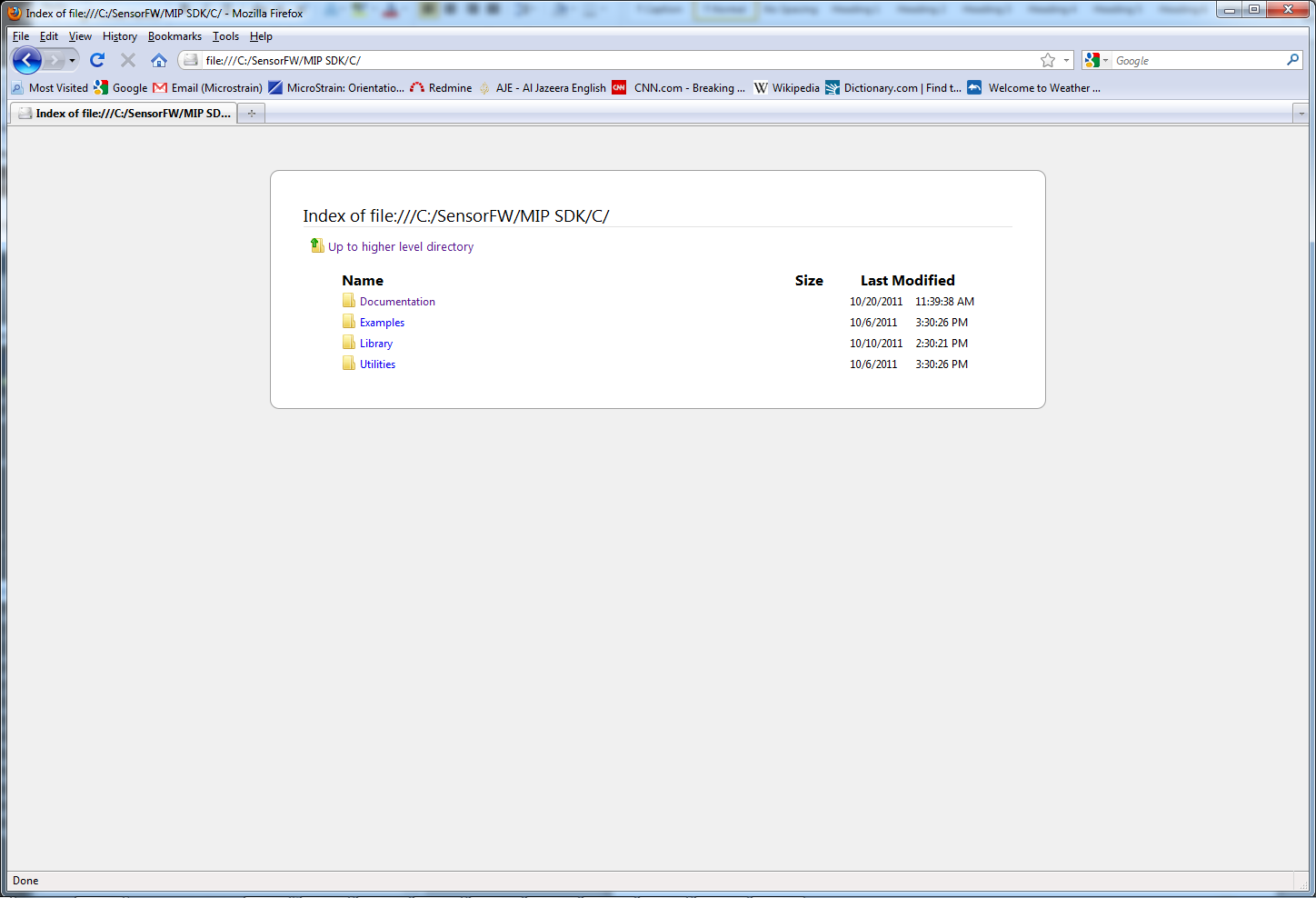
### Introduction

The C MIP SDK has been designed with portability in mind. The target-specific functions have all been grouped into a single source file, which must be edited by the user to implement hardware-level calls (e.g. serial port open, close, read, write, etc.) on their target platform. The remaining files are target-agnostic, and can be compiled on a wide range of platforms. This structure allows the user to develop across several platforms easily.

The SDK include a few key components: a “smart” MIP parser, a callback interface for data packets, functions for assembling and processing MIP packets, and high-level functions for performing descriptor-set-specific commands and byte-swapping MIP data.

### Directory Structure

The following directory structure is found within the MIP SDK/C/ directory:



**Documentation:** Includes this file as well as the SDK documentation generated from Doxygen. All of the functions and structures are documented here.

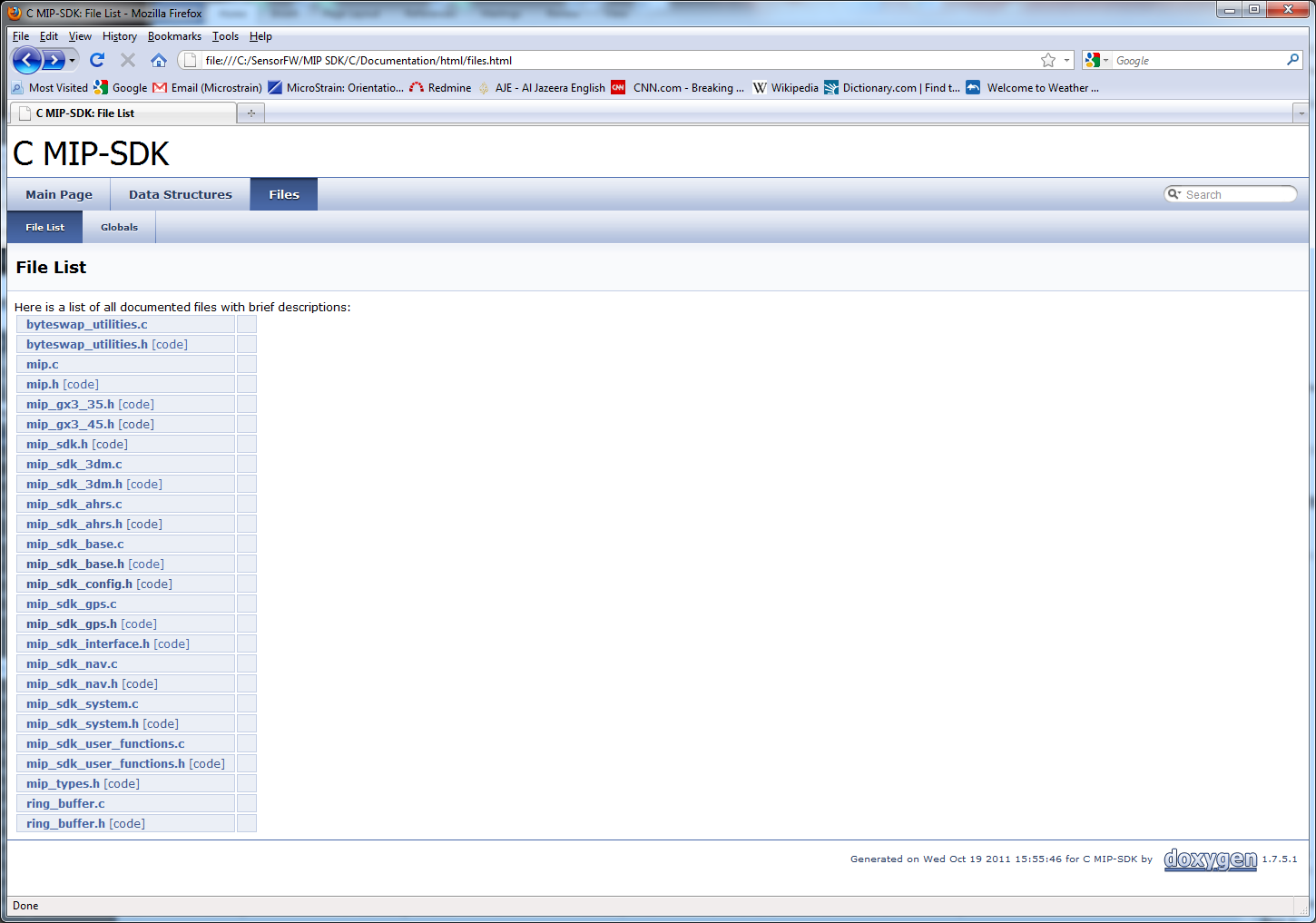
**Examples:** Contains several examples which use the MIP SDK. Visual Studio 2005 on Windows is the primary example project type.

**Library:** Contains the header and source files for the MIP SDK.

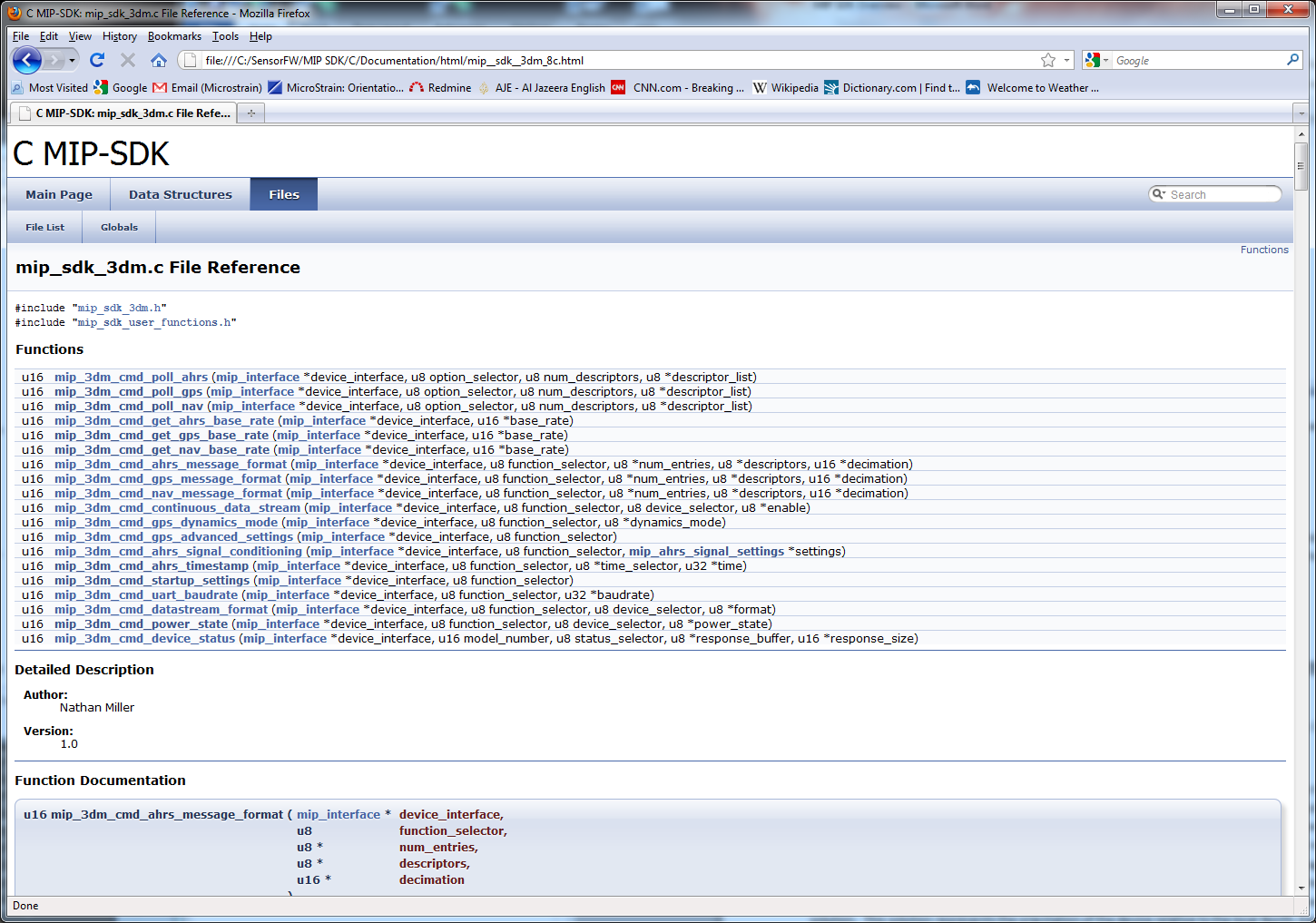
**Utilities:** Contains the header and source files for utility functions used by the SDK, such as byte-swapping.

### SDK Documentation

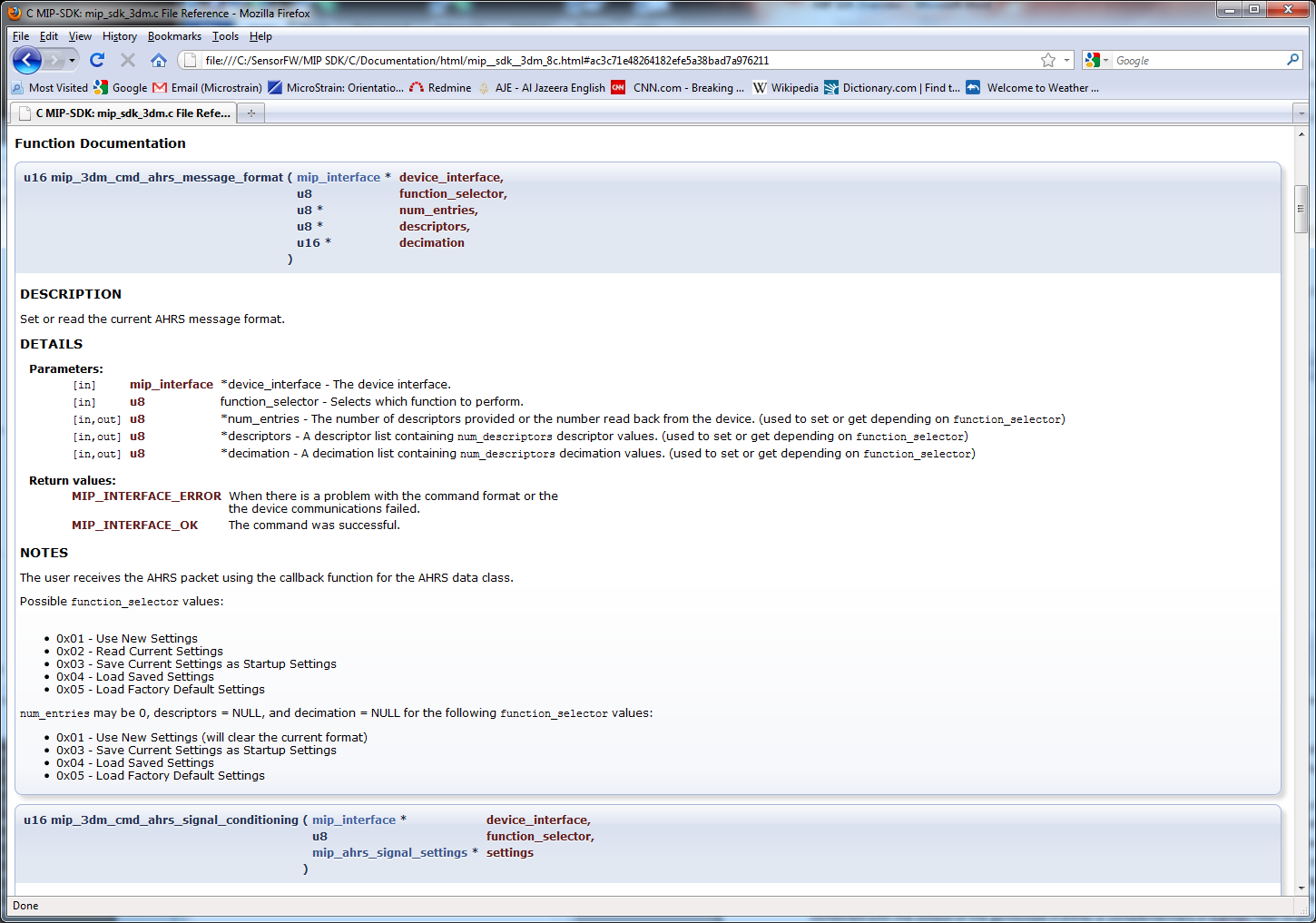
An HTML version of the SDK documentation is found in the “MIP SDK/C/Documentation/html” directory. The entry point is the index.html file. The following shows you what to expect after clicking the “Files” tab:



Each file is a link to a new page containing the documentation for that particular file. For example, if we click the “mip\_sdk\_3dm.c” file we get the following:



Here we see a list of all the functions supported by the 3DM descriptor set. When we click on one of the functions, such as “u16 mip\_3dm\_ahrs\_message\_format(…)”, we get the detailed documentation:



All inputs, outputs, and return values are documented for each function in the SDK. Additional notes are provided when needed to reduce any uncertainty in the outcome of calling a function.

### SDK Configuration

All SDK configurable quantities can be found in the mip\_sdk\_config.h file located in the “MIP SDK/C/Library/Include/” directory. These quantities deal with target-system endianess, available memory, and port timeout values. The default values should be sufficient for most target systems, with the exception of the MIP\_SDK\_CONFIG\_BYTESWAP value. The MIP protocol is big-endian; if the target system is big-endian, then this value should be 0. If the target system is little-endian, then this value should be 1.

### Basic Functions Assumed to be Available

The SDK has been written with the assumption that the following functions are available in the target compiler:

*void \*****memcpy****(void \* destination, void \*source, size\_t num);*

*void \*****memset****(void \*ptr, int value, size\_t num);*

Almost all modern compilers implement these functions, but if they are unavailable on the target system, the user is required to write them.

### Target-Specific User Functions

In order to keep the majority of the SDK platform-independent, there are certain functions within the SDK that must be completed by the user. These functions are found in the “mip\_sdk\_user\_functions.h” and “mip\_sdk\_user\_functions.c” files located in the “MIP SDK/C/Library/User Functions/” directory. The six functions are:

*u16* ***mip\_sdk\_port\_open****(void \*\*port\_handle, int port\_num, int baudrate);*

*u16* ***mip\_sdk\_port\_close****(void \*port\_handle);*

*u16* ***mip\_sdk\_port\_write****(void \*port\_handle, u8 \*buffer, u32 num\_bytes, u32 \*bytes\_written,*

*u32 timeout\_ms);*

*u16* ***mip\_sdk\_port\_read****(void \*port\_handle, u8 \*buffer, u32 num\_bytes, u32 \*bytes\_read,*

*u32 timeout\_ms);*

*u32* ***mip\_sdk\_port\_read\_count****(void \*port\_handle);*

*u32* ***mip\_sdk\_get\_time\_ms****();*

Examples of how the user may implement these functions are given in the SDK Examples section and the expected nature of each is templated in the SDK documentation.

It is recommended that the “mip\_sdk\_user\_functions.h” and “mip\_sdk\_user\_functions.c” files be copied from their default location into the user’s project directory. This allows the user to develop for different targets, while still keeping an unmodified template version in the default directory.

### Utility Functions

The MIP SDK includes the “MIP SDK/C/Utilities” directory, which provides a platform-independent set of standard utility functions. Currently, this directory includes byte-swapping utility functions.

# 3. C MIP SDK Interface Overview

### Interface Overview

The mip\_interface structure and its associated functions form the core of the MIP SDK. It handles all device read and write calls, datastream parsing, command/response handling, and data callback handling. The interface is designed to be utilized in ***single-threaded applications only***. The following diagram illustrates how the pieces of the SDK fit together:

**User Target**

**MIP Interface**

**MIP  
Device**

Physical  
Port

**Target-Specific  
Functions**

(Port Open, Close, Read, Write)

**MIP  
Parser**

Command  
Response  
Handler

Data   
Callback  
Handler

MIP SDK High-Level   
Command Functions

(3DM, NAV, etc.)

User Callback Functions

### Parser Description

The primary job of the MIP parser is to identify packets arriving on the incoming data stream and route them to the appropriate callback function. Callback functions are associated with a specific MIP descriptor set and are registered by the user during setup. The parser calls a callback function when it receives a MIP packet with a descriptor set that it recognizes and one of the following conditions are met: a packet has been received with a valid checksum, a packet has been received with an invalid checksum, or a timeout has occurred. The specific case is provided to the callback function as one of the parameters.

When the user issues one of the high-level commands (e.g. mip\_3dm\_cmd\_poll\_ahrs()), the MIP interface handles the command/response routing behind the scenes using a custom callback function for the command descriptor set to process the response.

The parser is written to minimize packet misreads. It reads data from the internal ring buffer looking for the first start of packet byte (‘u’ 0x75), consuming any bytes as it goes. Once the first start of packet byte has been identified, the parser switches to look-ahead reads, processing bytes in the ring buffer without consuming them; this minimizes the re-sync time and insures more reliable packet reading in the case that a partial packet was received. The parser performs a look-ahead read for the remaining bytes of the MIP packet header. Once the required number of bytes has been received and the second start of packet byte (‘e’ 0x65) and payload length have been validated, the parser performs a look-ahead read for the remaining packet bytes. When the entire packet size has been received, the parser attempts to validate the MIP checksum. If the checksum is valid and the parser has a registered callback for the descriptor set, the parser calls the callback function. If the checksum is invalid, the parser will also call the callback function and provide a flag to indicate the callback is due to an invalid checksum. This is done so the user can identify that a bad packet has been received and take whatever action they feel is appropriate. If at any time after receiving the first start of packet byte (‘u’ 0x75), the parser experiences a timeout on the incoming port, parsing of the current packet is abandoned. If the parsing process if far enough along that it has a valid packet header, the parser will call the callback function associated with the packet descriptor set and provide a flag indicating the callback was due to a timeout. When the packet is successfully read, all of the bytes that have been read using the look-ahead method are now consumed. If the checksum is invalid or a timeout occurs, the bytes are not consumed and the parser will start from the next byte after the first start of packet byte. This process is repeated for all bytes on the port every time the user calls the mip\_sdk\_update() function.

### Parser Callback Functions

The MIP interface holds a list of callback functions that are triggered when the MIP descriptor set of the current packet being parsed matches. *Only the first function in the list that matches the MIP descriptor set is called*. If the user requires the routing of the packet to multiple destinations, he must do it from within the callback function. Callback functions have the following format:

void (\*packet\_callback)(void \*user\_ptr, u8 \*packet, u16 packet\_size, u8 callback\_type)

The following is an example of a callback function declaration for AHRS data packets:

void ahrs\_packet\_callback(void \*user\_ptr, u8 \*packet, u16 packet\_size, u8 callback\_type);

This callback would be registered for the MIP interface “device\_interface” for any AHRS data packets with the following function call:

mip\_interface\_add\_descriptor\_set\_callback(&device\_interface, MIP\_AHRS\_DATA\_SET, NULL, &ahrs\_packet\_callback);

The callback system allows for a pointer to user-defined data be passed along to the callback if desired. This pointer is setup in the mip\_interface\_add\_descriptor\_set\_callback() function and arrives to the callback in the void \*user\_ptr parameter.

A description of the parameters provided to the callback function follow:

void \*user\_pointer – a pointer passed to the callback function that points to a user-defined area of memory. This pointer is stored in the callback list at the time the callback is configured and can be NULL, as in the example above.

u8 \*packet – a pointer to the entire MIP packet, including header and checksum. The standard MIP parsing functions included with the SDK can be used to parse the packet.

u16 packet\_size – the size of the MIP packet, including the header and checksum.

u8 callback\_type – identifies whether this is a valid packet, bad checksum, or timeout callback.

Functions for adding and deleting descriptor set callback functions are provided in the SDK and detailed in the SDK documentation.

### Command-Response Handling

Command-response handling is a special case of the standard callback system. The MIP interface registers an internal callback function to handle command responses. When a command is issued, the descriptor set and command descriptor are stored in the mip\_interface structure at the top of the callback list. When a response packet arrives that matches the stored description, it is parsed for the ACK/NACK response and a pointer to any additional data fields; this information is returned to the calling function. Most of this process is abstracted from the user through the high-level SDK command functions, but an understanding of how it works is crucial for customers who want to implement their own command functions.

An important note: commands are always issued with a timeout for the reception of the response. While waiting for the response, the MIP interface continuously calls the mip\_update() function. This allows packet parsing to continue as normal. If packets arrive that match a stored callback descriptor set, the callback function is called; this ensures that no packet is wasted, even when waiting for a command response. The user should be careful to minimize the time spent in callback functions, since lengthy processing could result in a timeout for command responses. The standard timeout values used for all of the high-level SDK commands are configurable and are located in the mip\_sdk\_config.h file.

### Standard Interface Use

The following example shows how to setup and use the MIP interface. This example will need to be modified for the user’s application and target MIP device, but this provides the basic steps necessary to get the interface up and running. The user may want to issues setup commands (step 6) prior to registering callback functions (step 5) if they do not want the data callback handlers being called during setup.

**Step 1** – Include the MIP SDK library:

#include "mip\_sdk.h"

**Step 2** – Define the interface as either a global variable or a local variable within the main() function:

//The primary device interface structure

mip\_interface device\_interface;

**Step 3** – Define any packet callback functions (only one function is given here, but similar functions would be defined for ahrs\_packet\_callback() and gps\_packet\_callback() for this example):

///////////////////////////////////////////////////////////////////////

// FILTER Packet Callback

///////////////////////////////////////////////////////////////////////

void filter\_packet\_callback(void \*user\_ptr, u8 \*packet, u16 packet\_size,

u8 callback\_type)

{

switch(callback\_type)

{

case MIP\_INTERFACE\_CALLBACK\_VALID\_PACKET:

{

//User will want to do something here

}break;

case MIP\_INTERFACE\_CALLBACK\_CHECKSUM\_ERROR:

{

//User may want to do something here

}break;

case MIP\_INTERFACE\_CALLBACK\_TIMEOUT:

{

//User may want to do something here

}break;

default: break;

}

}

**Step 4** – Initialize the interface with a valid COM port number, baudrate, and default packet parsing timeout (in milliseconds):

///

//Initialize the interface to the device

///

if(mip\_interface\_init(com\_port, baudrate, &device\_interface, DEFAULT\_PACKET\_TIMEOUT\_MS) != MIP\_INTERFACE\_OK)

return -1;

**Step 5** – Register packet callback functions:

///

//Setup the GX4-45 dataset callbacks (FILTER, AHRS, GPS)

///

if(mip\_interface\_add\_descriptor\_set\_callback(&device\_interface, MIP\_FILTER\_DATA\_SET, NULL, &filter\_packet\_callback) != MIP\_INTERFACE\_OK)

return -1;

if(mip\_interface\_add\_descriptor\_set\_callback(&device\_interface, MIP\_AHRS\_DATA\_SET, NULL, &ahrs\_packet\_callback) != MIP\_INTERFACE\_OK)

return -1;

if(mip\_interface\_add\_descriptor\_set\_callback(&device\_interface, MIP\_GPS\_DATA\_SET, NULL, &gps\_packet\_callback) != MIP\_INTERFACE\_OK)

return -1;

**Step 6** – Perform any required setup commands (what follows is just a subset of what a user would probably want to do with a real device):

u8 enable = 1;

///

//Enable the FILTER datastream

///

mip\_3dm\_cmd\_continuous\_data\_stream(&device\_interface,

MIP\_FUNCTION\_SELECTOR\_WRITE, MIP\_3DM\_INS\_DATASTREAM, &enable);

///

//Enable the AHRS datastream

///

mip\_3dm\_cmd\_continuous\_data\_stream(&device\_interface,

MIP\_FUNCTION\_SELECTOR\_WRITE, MIP\_3DM\_AHRS\_DATASTREAM, &enable);

///

//Enable the GPS datastream

///

mip\_3dm\_cmd\_continuous\_data\_stream(&device\_interface,

MIP\_FUNCTION\_SELECTOR\_WRITE, MIP\_3DM\_GPS\_DATASTREAM, &enable);

**Step 7** – Call the mip\_interface\_update() function in the main execution loop (simple Windows example):

///

//Wait for packets to arrive

///

while(1)

{

//Update the parser

mip\_interface\_update(&device\_interface);

//Be nice to other programs

Sleep(1);

}

# 4. Included MIP SDK Programming Examples

### GX3-35 Unit Test Example

The GX3-35 unit test example program is located in the “MIP SDK/C/Examples/Windows/GX3-35/GX3\_35\_Test” directory. This program runs through almost all of the 3DM commands and functions as a unit test for the SDK. Since the goal of this program is testing as many functions as possible, it *should not* be used as an example of how you would normally setup a GX3-35. That being said, most of the functions that one would need to setup the device are present. The only functions missing from this test are those that would alter the start-up state of the device.

### GX3-45 Unit Test Example

The GX3-45 unit test example program is located in the “MIP SDK/C/Examples/Windows/GX3-45/GX3\_45\_Test” directory. This program runs through all of the FILTER commands and functions as a unit test for the FILTER SDK. Since the goal of this program is testing as many functions as possible, it *should not* be used as an example of how you would normally setup a GX3-45. That being said, most of the functions that one would need to setup the device are present. The only functions missing from this test are those that would alter the start-up state of the device.

### GX4-15 Unit Test Example

The GX4-15 unit test example program is located in the “MIP SDK/C/Examples/Windows/GX4-15/GX4\_15\_Test” directory. This program runs through all of the FILTER commands related to Vertical Gyro functionality and functions as a unit test for the FILTER SDK. Since the goal of this program is testing as many functions as possible, it *should not* be used as an example of how you would normally setup a GX4-15. That being said, most of the functions that one would need to setup the device are present. The only functions missing from this test are those that would alter the start-up state of the device.

### GX4-25 Unit Test Example

The GX4-25 unit test example program is located in the “MIP SDK/C/Examples/Windows/GX4-25/GX4\_25\_Test” directory. This program runs through all of the FILTER commands related to AHRS functionality and functions as a unit test for the FILTER SDK. Since the goal of this program is testing as many functions as possible, it *should not* be used as an example of how you would normally setup a GX4-25. That being said, most of the functions that one would need to setup the device are present. The only functions missing from this test are those that would alter the start-up state of the device.

### GX4-45 Unit Test Example

The GX4-45 unit test example program is located in the “MIP SDK/C/Examples/Windows/GX4-45/GX4\_45\_Test” directory. This program runs through all of the FILTER commands related to GPS-INS functionality and functions as a unit test for the FILTER SDK. Since the goal of this program is testing as many functions as possible, it *should not* be used as an example of how you would normally setup a GX4-45. That being said, most of the functions that one would need to setup the device are present. The only functions missing from this test are those that would alter the start-up state of the device.

***End of Document***

|  |  |  |  |
| --- | --- | --- | --- |
| Copyright © 2013 LORD Corporation  Strain Wizard®, DEMOD-DC®, DVRT®, DVRT-Link™, WSDA®, HS-Link®, TC-Link®, G-Link®,  V-Link®, SG-Link®, ENV-Link™, Watt-Link™, Shock-Link™, LXRS®, Node Commander®,  SensorCloud™, Live Connect™, MathEngine®, EH-Link®, 3DM®, FAS-A®,  3DM-GX3®, 3DM-DH®, 3DM-DH3™, 3DM-RQ1™, MicroStrain®, and Little Sensors, Big Ideas®  are trademarks of LORD Corporation.  8500-0035 rev 000 |  | LORD Corporation  MicroStrain® Sensing Systems  459 Hurricane Lane,  Suite 102  Williston, VT 05495 USA  [www.microstrain.com](http://www.microstrain.com) | ph: 800-449-3878  fax: 802-863-4093  [sensing\_support@LORD.com](mailto:sensing_support@LORD.com)  [sensing\_sales@LORD.com](mailto:sensing_sales@LORD.com?subject=sales%20inquiry) |