The AMTI USB Device Software Development Kit

Reference Manual

Version 1.2

Notice

AMTI does not warrant that the AMTI USB Device DLL software will function properly in every hardware software environment. This software is inherently complex, and users are cautioned to verify the results of their work.

AMTI has tested the software and reviewed the documentation. AMTI MAKES NO WARANTY OR REPRESENTATION, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THIS SOFTWARE OR DOCUMENTATION, THEIR QUALITY, PERFORMANCE, MERCHANTIBILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE AND DOCUMENTATION ARE LICENSED "AS IS" AND YOU, THE LICENSEE ARE ASSUMING THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE.

IN NO EVENT WILL AMTI BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE OR DOCUMENTATION, even if advised of the possibility of such damages. In particular, AMTI shall have no liability for any programs or data stored or used with AMTI software, including the costs of recovering such programs or data.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the above disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the above disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

AMTI Contact Information

Advanced Mechanical Technology
176 Waltham St.
Watertown, MA 02478
Phone (617)926-6700
Website www.amti.biz
Email helpdesk@amtimail.com

1.0 Introduction

The AMTI USB Device Software Development Kit (SDK) is designed to assist third party vendors in integrating one or more AMTI digital USB signal conditioners into their applications. The AMTI SDK allows vendors to communicate with AMTI hardware through a USB 2.0 interface.

The SDK consists of a regular dynamic-link library (DLL) named **AMTIUSBDevice.dll**, a library file named **AMTIUSBDevice.lib**, and a header file named **AMTIUSBDevice.h**. To get started, these three files should be included in the project as when integrating any regular DLL.

The DLL is written in Visual C++, using Visual Studio 2010. It is configured for both the Win 32 and Win 64 platforms, and is compatible with Windows 7 Wow64. The reader is expected to be familiar with Dynamic Link Libraries and their uses.

2.0 The AMTI Digital Signal Conditioner

The overall function of the AMTI Digital Signal Conditioner is to condition data from six strain gauge inputs and output the results as six analog channels and/or a six-channel digital data stream. The analog outputs are high level and suitable as inputs to a multi-channel Analog to Digital Converter (ADC). The digital data are transmitted to a host Personal Computer (PC) via a Universal Serial Bus (USB) connection. The USB port is also used to send and receive control and status information used by the signal conditioner.

The overall signal conditioner functionality can be divided as follows:

1. Provide analog signal conditioning for six strain gauge inputs including production of six independently selectable strain gauge excitation voltages, bridge balancing with independently selectable offsets, filtering and amplification at independently selectable gains.

2. Perform periodic sampling of the six conditioned analog signals at selectable rates.

3. Perform numerical processing of digitized signals including conversion to engineering units.

4. Convert numerically processed data to high-level analog signals suitable for an ADC via a Digital to Analog Converter (DAC) and analog signal conditioning.

5. Provide an industry standard USB port for data transmission and reception.

6. Provide non-volatile memory for the storage of calibration and configuration data.

7. Provide for the reading of calibration coefficients and other data from AMTI Smart Platforms equipped with Read Only Memory (ROM).

The above functions are implemented by the signal conditioner with analog circuitry and two MCU's. A Silicon Laboratories C8051F120A mixed signal MCU with FLASH and its peripheral circuits perform all functions except the USB port implementation. A Cypress Semiconductor Corporation CY7C68013A-128AC (EZ-USB FX2LP) single-chip USB MCU implements the industry standard USB port.

Integrated Digital Hall-effect Platforms

The AMTI SDK and digital DLL support integrated Hall-effect digital platforms in addition to the digital signal conditioners described above. These platforms contain AMTI Optima digital logic built into the platform hardware. Some of the strain-gauge functionality and analog output functions supported by the SDK are not applicable to these integrated platforms, and calls to certain functions will have no effect on them. Nevertheless, most user application software written to support AMTI digital signal conditioners can be used unaltered to manage and read data from AMTI integrated Hall-effect platforms.

3.0 Software Development Strategy

When considering integrating the AMTI SDK with an application there are two options: full integration or partial integration. Full integration involves integrating most of the features of this SDK into the application; this gives the application full control of the signal conditioners. Partial integration involves integrating only the data collection portion of the SDK.

The AMTI System Configuration program is a utility program which ships with every AMTI signal conditioner. It is used to set up and configure both the DLL and the signal conditioners. For a partial integration strategy, use the AMTI System Configuration program for signal conditioner setup and configuration, and then only integrate the data collection processes into the third party application. Doing this requires only familiarity with the following sections: *Initializing and Configuring the DLL*, and *Data Collection*. For many users this will be the way to go.

4.0 LabView Compatibility

When integrating the AMTI digital SDK with LabView, AMTI recommends using the partial integration strategy described in section 3.0. Use the AMTI System Configuration program for setting up and configuring the DLL and signal conditioners, and then integrating only the data collection processes into the LabView application.

AMTI recommends using the polling method of data collection. For data transfer, the ***fmDLLGetTheFloatDataLBVStyle*** function should be used.

AMTI does provide starter source code for a simple LabView data collection program.

The recommended data collection method above has been tested for LabView compatibility.

5.0 Definitions

AD – Analog to digital conversion

DAC – Digital to analog conversion

MCU – microcontroller unit

Platform – In this manual the word platform may also be substituted with transducer, load cell, any six-channel strain gage multi-axis measurement device.

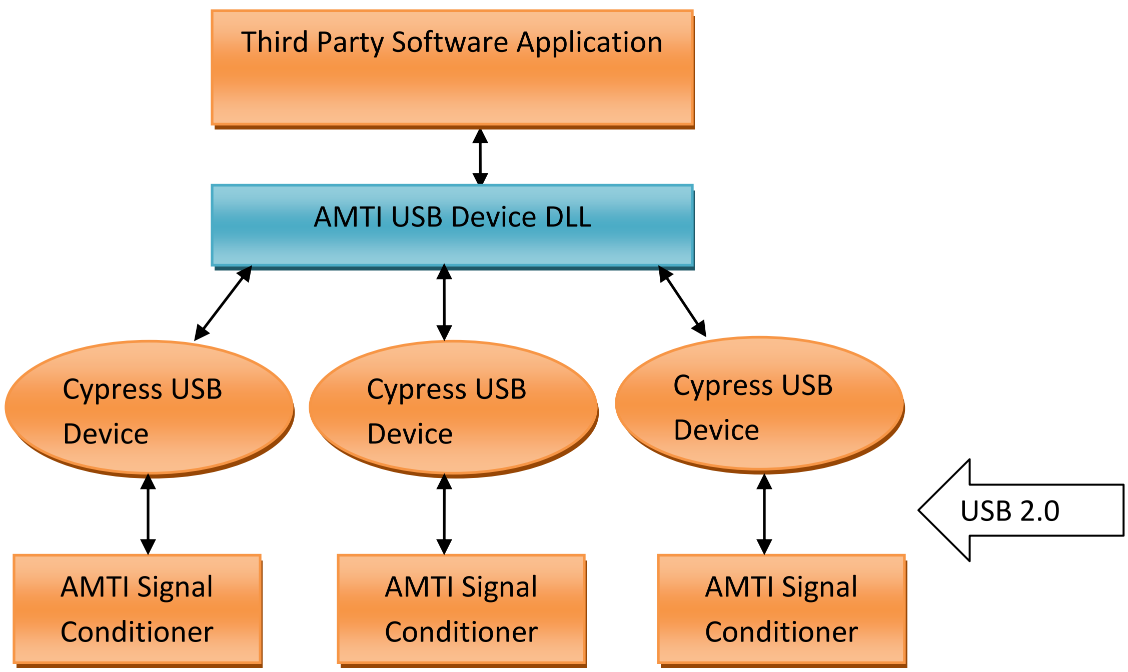
Electrical range – In this manual electrical range is the maximum and minimum measurement capacity of the signal conditioner expressed as engineering units.

Analog output range – The analog output range of an AMTI signal conditioner is ±5 volts. When we discuss the analog output range in this manual we are frequently referring to it in engineering units.

6.0 Software System Overview

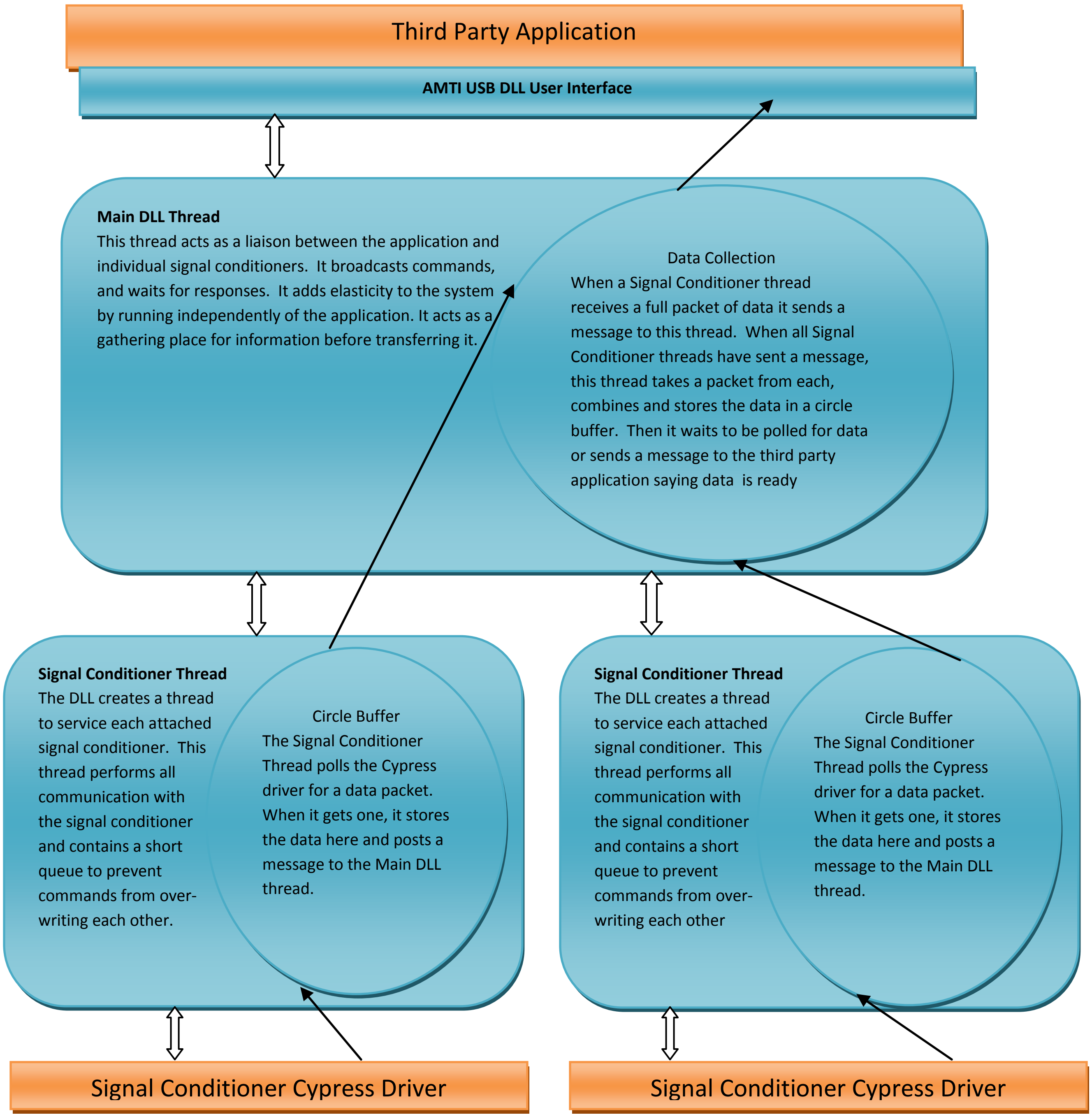
The diagram below illustrates the relationship between the AMTI USB Device DLL and the rest of the system. The DLL handles all communication between the third party application and the Signal Conditioner Cypress device drivers. When initialized, the DLL will find and communicate with each signal conditioner which is connected to the PC through a USB 2.0 port.

Figure 1 - Software System Overview



As illustrated in Figure 2, the DLL creates a thread to act as a liaison between the third party application and all of the AMTI signal conditioners. The DLL spawns additional threads to service each connected signal conditioner.

Figure 2 – Software System Architecture



6.1 Understanding the Different Function Types

There are three different types of functions in the SDK. The function prefix distinguishes them.

Function Prefix	Function Prefix Meaning
fmBroadcast	The ***fmBroadcast*** prefix indicates the function is a global command. The function broadcasts the command to all currently connected signal conditioners.
fmDLL	The ***fmDLL*** prefix indicates the function has to do with the current configuration settings of the DLL, not the signal conditioners.
fm	The ***fm*** *prefix,* excluding the ***fmBroadcast*** and ***fmDLL*** types, is concerned with only one signal conditioner. In order to communicate with a specific signal conditioner the function ***fmDLLSelectDeviceIndex*** must be called to select that signal conditioner. The selected signal conditioner remains selected until another signal conditioner is selected, or the DLL terminates.

6.2 Selecting a Device

Before communicating with a specific signal conditioner, the device must be selected first. The ***fmDLLSelectDeviceIndex*** function selects the signal conditioner by its device index. Once a signal conditioner has been selected it be accessed through of the ***fm*** prefix type functions.

The device indices are ordered from 0 to the number of signal conditioners minus one. They are always ordered in the platform data collection order stored in the DLL configuration file. Use ***fmDLLGetDeviceCount*** to know how many devices are connected.

To find out what signal conditioner or platform is associated with a particular index, call ***fmGetAmplifierSerialNumber*** and other amplifier and platform identification functions.

6.3 Understanding the Signal Conditioner Channel Order

An AMTI signal conditioner is a 6-channel data collection device. It collects data from force plates which measure forces and moments. The channel order is always the three forces, followed by the three moments.

Fx – The force vector along the x axis of a platform
Fy – The force vector along the y axis of a platform
Fz – The force vector along the z axis of a platform
Mx – The moment around the x axis of a platform
My – The moment around the y axis of a platform
Mz – The moment around the z axis of a platform

Channel	Forces			Moments		
Index	0	1	2	3	4	5
Row	Fx	Fy	Fz	Mx	My	Mz

When uploading or downloading parameters the channel order as shown in the above table is always maintained. If we are uploading or downloading a table with multiple entries for each channel the channel order is still maintained as in the following table where each channel has three entries. The table is always organized in row, column order.

Index	0	1	2	3	4	5
Row 1	Fx	Fy	Fz	Mx	My	Mz
Index	6	7	8	9	10	11
Row 2	Fx	Fy	Fz	Mx	My	Mz
Index	12	13	14	15	16	17
Row 3	Fx	Fy	Fz	Mx	My	Mz

6.4 Applying and Saving Parameters

There are multiple functions for applying and saving parameters. To avoid confusion they are each described below.

Applying Signal Conditioner Settings

When new calibration tables or configuration parameters are downloaded to the signal conditioners they are not automatically applied. The functions ***fmBroadcastResetSoftware*** and ***fmResetSoftware*** are used to apply parameters to the signal conditioner hardware. The exceptions to this rule are the Start, Stop, Zero, Blink and Set acquisition rate commands. Additionally, these two functions do not apply to the DLL configuration settings. The two reset software functions require a time delay after being called. The internal signal conditioner software resets itself and the signal conditioner will not accept commands while resetting. It is recommended that all of configuration changes be made, and then a Reset function be called when done. The reset software functions do not save the parameter changes to permanent flash memory.

Saving Signal Conditioner Settings

Each signal conditioner maintains its own calibration tables, platform calibration tables, and current configuration settings all within its own internal flash memory. To save the current configuration settings to permanent flash memory the functions ***fmBroadcastSave*** or ***fmSave*** must be used.

Saving the DLL Configuration Settings

To save the current DLL settings, call ***fmDLLSaveConfiguration***. For a list of DLL configuration settings, please see the section titled *The DLL Configuration File* in Section 7.

7.0 Initializing and Configuring the DLL

Initializing the DLL

The first DLL function called in an application will always be ***fmDLLInit***.

The ***fmDLLInit*** function does the following:

1) The DLL searches for the signal conditioners. When a signal conditioner is found it uploads all of the signal conditioner parameters and stores them in local memory. These parameters include both calibration tables and configuration settings. It does this so it can retrieve parameters instantly without having to query the signal conditioner. Additionally all future parameter changes update both DLL memory and signal conditioner memory.

2) The DLL then loads the configuration file. It compares the saved configuration to the current configuration. It checks to see that the same number of signal conditioners is present as the last time it was run. It compares the serial numbers and makes sure the serial numbers match. It sets up the data collection order of the platforms to insure the data is presented in the same platform order as saved in the last configuration. If some signal conditioners are not present the platform order of the others will be maintained.

3) When the DLL has completed initializing it will set a flag. The flag can be checked by calling ***fmDLLIsDeviceInitComplete***.

4) After the DLL has finished initializing call ***fmDLLSetupCheck***. That will return an status value to alert the application to configuration differences between the current configuration and the last saved configuration.

The DLL Configuration File

The DLL configuration file is AMTIUsbSetup.cfg. This file is located in the C:\AMTI\CFG folder.

The configuration file maintains the last saved DLL configuration settings listed in the table below. In addition to this list it maintains the data collection order of the platforms.

Table 1 – The DLL configuration Settings

Global Settings	**Description , Range, or possible values**
The configuration file version number	101 (current version)
The signal conditioner count	0-16 (range of possible values)
The acquisition rate	10-2000 (range of possible values)
The run mode	0-4 (range of possible values)
The genlock state	0-2 (range of possible values)
Signal Conditioner Settings	**Always saved in data collection order**
Signal conditioner serial number	
Signal conditioner model name	
Platform serial number	
Platform model name	

The acquisition rate, run mode and genlock settings stored in the configuration file are the last broadcast settings downloaded before the configuration file was last saved. It does not mean that all of the signal conditioners are configured to these settings. To be sure all signal conditioners are configured to the same settings, it is recommended that the desired settings be re-broadcast after the DLL has initialized.

The function ***fmDLLSaveConfiguration*** saves the DLL configuration file. It is not automatically updated upon closing the DLL.

Re-initializing the DLL

Use the ***fmDLLInit*** function to reinitialize the DLL. Simply call it again and it will re-initialize. The reason to do this is to find signal conditioners that were either unplugged or added after the application has started.

If a signal conditioner is removed while the application is running, the DLL should be re-initialized. It will no longer function correctly if a signal conditioner is not present.

DLL Cleanup

In order to shut down the DLL, ***fmDLLShutDown*** must be called. Calling this function terminates all running threads and performs cleanup for the DLL. After calling this command adequate time must be allotted for cleanup before closing the application. This time increases per signal conditioner, however 500 msec should be more than adequate.

If the DLL will be reinitialized to search for additional signal conditioners and not terminating the application just call ***fmDLLInit*** again. Do not call ***fmDLLShutDown***.

8.0 Collecting Data

This section describes the decisions which must be made to set up the data acquisition process. Each function presents data acquisition options which must be considered. Consider each one and configure the DLL accordingly.

Choosing a Data Collection Method

There are three ways to collect digital data:

1. The first is polling; simply poll the DLL continuously to see if data is available. The data transfer function will either return a pointer to data, or return 0 if no new data are available.

2. The second is to have the DLL post a message to the application main window every time data is ready. Upon receiving the message the application can use the data transfer function to receive the data. To set up for windows messaging use the functions ***fmDLLPostDataReadyMessages*** and ***fmDLLPostWindowMessages***.

3. The third is to have the DLL post a message to a user thread every time data is ready. Upon receiving the message the application can use the data transfer function to receive the data. To set up for user thread messaging use the functions ***fmDLLPostDataReadyMessages*** and ***fmDLLPostUserThreadMessages***.

Note: The DLL cannot post data ready messages to both a user thread and a window at the same time.

Choosing the Data Transfer Function

The data transfer functions check to see if data is available and if so return the data. There are two data transfer functions. One is designed for integration with C/C++ programs, the other is recommended for Labview.

The function for C/C++ programming is ***fmDLLTransferFloatData***.

The function for Labview programming is ***fmDLLGetTheFloatDataLBVStyle***.

Setting the Data Format

There are two supported data formats. The application can receive datasets in six-channel or eight-channel format. A six-channel dataset will consist solely of the force and moment channels. An eight-channel dataset will have two additional channels, a dataset counter and a trigger state. To set the data format call ***fmDLLSetDataFormat.***

Setting the Data Packet Size

The DLL collects data in packets. Currently there is only one packet size and that is 512 bytes. To set the packet size call ***fmDLLSetUSBPacketSize*** and set it to 512.

Setting the Data Units

There are two ways to receive data from an AMTI signal conditioner. One is through the digital outputs; the other is through analog outputs. For digital outputs the unit choices are bits, English units, or metric units. For analog outputs the choices are fully conditioned and MSA 6 compatible. To set the data collection type call ***fmBroadcastRunMode***.

Setting the Acquisition Rate

The DLL can collect digital data at different rates. The function ***fmBroadcastAcquisitionRate*** is called to set the acquisition rate. The new acquisition rate will take affect with the next Start command. Different signal conditioner models may support different acquisition rates; please refer to the signal conditioner hardware documentation to see the supported rates.

Alternatively, data collection may be clocked using an external genlock signal. See section 12 *Using the Genlock Feature* for more information. If not using the genlock feature, the function ***fmBroadcastGenlock*** should be called to make sure it is turned off.

Zeroing the Platform

Before collecting data the platform should be zeroed in an unloaded state. The function ***fmBroadcastZero*** sends a zero command to all of the signal conditioner/platform pairs. This function may be called before or during acquisition.

The ***fmBroadcastZero*** function performs both a hardware zero and software tare on the platform.

Starting Acquisition

To start data acquisition the function ***fmBroadcastStart*** must be called. This function sends a start command to all connected signal conditioners.

This function only starts digital data collection. The analog outputs of the signal conditioners are always active.

When this function has been called additional start commands will be ignored until a stop command has been received.

Data collection will be automatically stopped if any other commands are sent to the signal conditioners after the start command has been broadcast. This is necessary to maintain the integrity of the synchronization between signal conditioners. The exception to this is the broadcast zero command. The zero command is the only command which may be broadcast during data collection that will not stop data collection.

Stopping Acquisition

To stop data acquisition, call the function ***fmBroadcastStop.***

Data collection will be automatically stopped if any other DLL commands are sent to the signal conditioners after the start command has been broadcast. This is necessary to maintain the integrity of the synchronization scheme between signal conditioners.

The List of Data Collection Functions

fmDLLSetUSBPacketSize
fmBroadcastRunMode
fmDLLGetRunMode
fmGetRunMode
fmBroadcastGenlock
fmDLLGetGenlock
fmBroadcastAcquisitionRate
fmDLLGetAcquisitionRate
fmGetAcquisitionRate
fmBroadcastStart
fmBroadcastStop
fmBroadcastZero
fmDLLPostDataReadyMessages
fmDLLPostWindowMessages
fmDLLPostUserThreadMessages
fmDLLSetDataFormat
fmDLLTransferFloatData
fmDLLGetTheFloatDataLBVStyle

9.0 Using the Signal Conditioner Configuration Functions

In order to use a signal conditioner it must be configured for use. The following table describes the configuration choices which must be made. Consult the signal conditioner user manual for additional information.

Table 2 - AMTI Signal Conditioner Configuration Parameter List

Current Gain	For each channel an amplifier gain must be selected. The choices are 500, 1000, 2000 or 4000.
Current Excitation	For each channel a strain gauge excitation voltage must be selected. The choices are 2.5, 5.0, or 10.0 volts.
Matrix Mode	The signal conditioner can either use the full platform calibration matrix when processing data or just the main diagonal terms of the matrix.
Channel Offset	For each channel a channel offset can be set. The channel offset allows the mechanical range of the signal conditioner to be offset by ± 99 %. It must be entered as a value between -0.99 and 0.99. The default setting is zero (no offset).
Platform Rotation	A single value entered in degrees instructing the platform to perform a coordinate transformation on the data. The default is zero (unrotated).
Cable Length	A single value giving the length of the cable between the platform and the signal conditioner in feet (1 foot = 30.5cm).
DAC Sensitivities	When using analog output in fully-conditioned mode, a DAC conversion value should be entered for each channel. It is used for scaling the analog outputs to a user supplied conversion factor. This factor is only applied when the analog outputs are set to fully conditioned mode.
Run Mode	A single value selecting the output modes of the signal conditioner. For digital outputs the choices are metric, English, or bits. For analog outputs the choices are fully conditioned or MSA 6 compatible.
Acquisition Rate	The digital data collection rate of the signal conditioner represented in datasets per second (Hz).
Genlock	A single value turning genlock mode on or off. The default is off.
Product Type	A read-only parameter representing the product type of the signal conditioner. Current product types are: 100 for a Gen 5; 300 for an Optima; and 400 for an integrated digital Hall-effect platform.

The following is the list of functions used to configure the signal conditioner

9.1 The List of Signal Conditioner Configuration Functions

fmSetCurrentGains
fmGetCurrentGains
fmSetCurrentExcitations
fmGetCurrentExcitations
fmSetMatrixMode
fmGetMatrixMode
fmSetChannelOffsetsTable
fmGetChannelOffsetsTable
fmSetPlatformRotation
fmGetPlatformRotation
fmSetCableLength
fmGetCableLength
fmSetDACSensitivityTable
fmGetDACSensitivities
fmDLLSetUSBPacketSize
fmBroadcastRunMode
fmDLLGetRunMode
fmGetRunMode
fmBroadcastGenlock
fmDLLGetGenlock
fmBroadcastAcquisitionRate
fmDLLGetAcquisitionRate
fmGetAcquisitionRate
fmGetProductType

10.0 Retrieving the Signal Conditioner Mechanical Limits

All force platforms have mechanical capacities which may not be exceeded. Each channel of the signal conditioner has an electrical range. Depending on the configuration the electrical range is mapped to different different mechanical ranges.

When a signal conditioner is first turned on it calculates the mechanical limits for each channel. The function ***fmGetMechanicalMaxAndMin*** returns the mechanical limits of the signal conditioner in engineering units.

If the functions ***fmBroadcastResetSoftware*** or ***fmResetSoftware*** are called to apply settings, the mechanical range is recalculated. To retrieve the recalculated mechanical range from the signal conditioner the function ***fmUpdateMechanicalMaxAndMin*** is required to upload the recalculated mechanical limits to the DLL. The function ***fmGetMechanicalMaxAndMin*** may then be called to retrieve the limits.

The digital output range of the signal conditioner is always the same as the mechanical range, not the platform capacity. The analog output range expressed in engineering units is always the same as the digital output range except when the analog output is in fully conditioned mode.

For a signal conditioner whose analog output is in fully conditioned mode, the output range is scaled to a user supplied digital to analog conversion factor. The output range expressed in engineering units will always be either less than or equal to the mechanical range.

The analog output range in engineering units is calculated when the signal conditioner is first turned on. The function ***fmGetAnalogMaxAndMin*** returns the analog output range in engineering units. This data returned by this function is indeterminate if the signal conditioner is not running in analog fully conditioned mode.

If the functions ***fmBroadcastResetSoftware*** or ***fmResetSoftware*** are called to apply settings the analog output range in engineering units is recalculated. To retrieve the recalculated range, the function ***fmUpdateAnalogMaxAndMin*** is required to upload the recalculated limits to the DLL. The function ***fmGetAnalogMaxAndMin*** may then be called to retrieve the new limits.

11.0 Determining the Platform Order

When more than one platform is installed it is important the the data are always presented in the same order. A dataset consists of a single sample of data concatenated together from each platform. The question is which platform's data should be presented first in each dataset.

The DLL allows the user to set and save the dataset platform order in the DLL configuration file. That way the dataset platform order is remembered from one session to the next. The function ***fmDLLSaveConfiguration*** saves the DLL configuration file.

There are two ways to set the dataset platform order. The platform order may be set manually or automatically.

Manually Setting the Dataset Platform Order

To manually set the platform order, call the function ***fmDLLSetPlatformOrder***. The identity of the signal conditioner/platform pairs must be known to the calling program.

Auto-ordering the Dataset Platform Order

The second method for setting the dataset platform order is called auto-ordering. In this scenario a user steps on the platforms in the desired dataset platform order. Four functions are used: ***fmBroadcastPlatformOrderingThreshold, fmDLLStartPlatformOrdering, fmDLLIsPlatformOrderingComplete***, and ***fmDLLCancelPlatformOrdering.***

For auto-ordering, the function ***fmBroadcastPlatformOrderingThreshold*** is called to set a load detection threshold in the DLL. When the function ***fmDLLStartPlatformOrdering*** is called the DLL goes into listen mode to detect the order in which the load detection threshold is triggered by a person stepping on each platform in the desired order. The function ***fmDLLIsPlatformOrderingComplete***, may then be called to confirm the platform ordering is complete. The function ***fmDLLCancelPlatformOrdering*** may be called at any time to cancel the operation.

The required steps for completing the platform auto-ordering are:

A. Call ***fmDLLSetDataFormat*** and set the format to parameter to 0 (6 channel collection).

B. Call ***fmBroadcastRunMode*** to collect using mode 4 (digital data as bits).

C. Call ***fmBroadcastPlatformOrderingThreshold*** and set an appropriate platform load detection threshold in bits (full scale bit range is ± 16384).

D. Call ***fmBroadcastResetSoftware*** to apply the changes.

E. Use a Sleep command (at least 250 msec) to allow the signal conditioners time to reset.

F. Call ***fmBroadcastZero*** to zero the unloaded platforms.

G. Call ***fmDLLStartPlatformOrdering*** to put the DLL into listening mode.

H. Call ***fmBroadcastStart*** to start data collection. The DLL will now check all incoming platform data to detect the order in which the platform threshold is crossed. It will not stop listening until all platforms have had their threshold crossed. Call ***fmDLLCancelPlatformOrdering*** to cancel the process.

I. Call ***fmDLLIsPlatformOrderingComplete*** to detect if the process has completed. We suggest either using a timer or a sleep function to periodically check for process completion.

J. Once the process has completed remember that this has changed the device index order of the signal conditioners. Loop through each device and request the serial numbers to figure out the new order.

K. Call ***fmDLLSaveConfiguration*** if to maintain the new order when the DLL is next initialized.

12.0 Using the Genlock Feature

Genlock is a common technique where the output of one source is used to synchronize multiple devices. AMTI digital signal conditioners have a genlock input port, which is designed to receive a clocking pulse that will cause a dataset to be recorded. When genlock is on, the signal conditioner will collect a single dataset on either the rising or falling edge of an analog input signal, usually a square wave of some sort. The function ***fmBroadcastGenlock*** is used to set the genlock mode of all connected signal conditioners.

The low state of the genlock input must be less than one volt. The high state must be greater than 3 volts but never more than 10 volts. The duration in either state must be greater than 20 microseconds to be detected. The genlock signal must be sent to all connected signal conditioners.

Before reading data using genlock, the signal conditioners should have their nominal acquisition rates set to a value as high or higher than the expected genlock pulse rate. Use the ***fmBroadcastAcquisitionRate*** to set the nominal acquisition rate.

The ***fmBroadcastStart*** function can be called before or after the genlock signal is started.

13.0 Using the External Trigger

AMTI signal conditioners can use the genlock port as a trigger input port. In the eight-channel data mode of digital output one of the channels is the trigger signal. The function ***fmDLLSetDataFormat*** determines whether the DLL delivers the full 8 channels or only the 6 force and moment channels. To see the trigger signal the DLL must be set up to deliver the full eight channels of data. A trigger channel value of 1 indicates the trigger input is high, and a value of 0 indicates the trigger input is low.

The low state of the trigger input must be less than one volt. The high state must be greater than 3 volts but never more than 10 volts. The duration in either state should be greater than the duration between datasets.

14.0 Using the Signal Conditioner Calibration Functions

The AMTI signal conditioner arrives already calibrated at the factory. The signal conditioner maintains its calibration tables within its permanent flash memory. The following table describes the calibration information thus stored. For further information about these settings refer to the user manual for the signal conditioner.

Table 3 – Signal Conditioner Calibration Parameter List

Item	**Description**
Model Number	The model number (name) of the signal conditioner
Serial Number	The serial number of the signal conditioner
Firmware Version	The firmware version of the signal conditioner
Calibration Date	Date the signal conditioner was last calibrated
Gain Table	A 24-element table containing the gain correction values for the 4 possible gain settings for each of the 6 channels
Excitation Table	An 18-element table containing the excitation correction values for the 3 possible excitation settings for each of the 6 channels

DAC Gains Table	A 6-element table containing the DAC gain corrections for each of the 6 analog output channels
DAC Offsets Table	A 6-element table containing the DAC zero offset corrections for each of the 6 analog output channels
ADRef	The nominal AD reference voltage

The List of Signal Conditioner Calibration Functions

The following functions are used to retrieve the calibration settings listed in the table above. These settings were set at the factory after a detailed calibration of the signal conditioner.

fmGetAmplifierModelNumber
fmGetAmplifierSerialNumber
fmGetAmplifierFirmwareVersion
fmGetAmplifierDate
fmGetGainTable
fmGetExcitationTable
fmGetDACGainsTable
fmGetDACOffsetTable
fmGetADRef

15.0 Using the Platform Calibration Functions

The AMTI signal conditioner delivers fully processed data to the PC through the USB connection. In order to do that, it must have the calibration tables for the attached platform available to it. The signal conditioner has space allocated within its permanent flash memory for storing calibration information about the attached platform. The table below describes all of the platform calibration information the signal conditioner should maintain.

Newer AMTI platforms come with smart chips embedded in them which contain the platform calibration information. If the attached platform is a smart platform, the signal conditioner will read the smart chip and load the calibration settings from it. If the attached platform is not a smart platform, the signal conditioner will use its locally saved settings.

Table 4 – The Platform Calibration Parameter List

Item	Description
Platform Date	Date the platform was last calibrated
Model Number	The model number of the platform
Serial Number	The serial number of the platform
Length	The length of the platform in inches
Width	The width of the platform in inches
X,Y,Z Offsets	A 3-element table giving the spatial coordinates of the platform's electrical center
Platform Capacity	A 6-element table containing the platform capacity for each of the 3 forces and 3 moments in English units
Bridge Resistances	A 6-element table containing the strain gauge bridge resistance for each platform channel in ohms (Ω)
Inverted Sensitivity Matrix	A 36-element table containing the inverted sensitivity matrix measured in English units (comes calibrated with the platform)

The List of Platform Calibration functions

The following functions are used to configure and retrieve the platform calibration settings.

fmSetPlatformDate
fmGetPlatformDate
fmSetPlatformModelNumber
fmGetPlatformModelNumber
fmSetPlatformSerialNumber
fmGetPlatformSerialNumber
fmSetPlatformLengthAndWidth
fmGetPlatformLengthAndWidth
fmSetPlatformXYZOffsets

fmGetPlatformXYZOffsets
fmSetPlatformCapacity
fmGetPlatformCapacity
fmSetPlatformBridgeResistance
fmGetPlatformBridgeResistance
fmSetInvertedSensitivityMatrix
fmGetInvertedSensitivityMatrix

16.0 Data Synchronization and the Signal Conditioners

The DLL handles all data synchronization between signal conditioners. When using a single USB hub the is skew is approximately ± 1.5 microseconds between signal conditioners. If using multiple hubs the skew is less than ± 125 microseconds between hubs.

17.0 AMTI Smart Platform and Signal Conditioner Communication

An AMTI smart platform contains all of its calibration information stored in a memory chip within the platform. When a signal conditioner is turned on, it checks to determine if it is connected to a smart platform. If it is so connected, it uploads the smart platform's calibration information and uses it.

NOTE: If a platform is hot-swapped to a running signal conditioner, the signal conditioner must be power-cycled to detect the smart platform. AMTI does not recommend hot swapping equipment.

18.0 Troubleshooting Tips

Question: Why is the DLL is not delivering data after sending the start command?

Answer: The individual signal conditioners may be set for different acquisition rates, data types and genlock states. This happens because some signal conditioners have been turned off for a while or a new one is introduced. When first stating up, broadcast the desired acquisition rate, genlock state and data types to prevent this.

If using genlock, remember that a nominal acquisition rate must be set at least as high as the highest expected genlock pulse rate. If the rate is too low, it may cause stalling in the data flow.

Question: Why are data received from the DLL reaching a 'plateau value' and not showing accurate data at the higher levels?

Answer: It is important to set the excitation voltages and gain factors to a range appropriate for the expected data loads on each channel. Excessive signals to the signal conditioner will result in the AD converters saturating at their peak value and not reflecting actual force data. The excitations and gains can be set in the application using SDK functions (see Section 23), or can be set in advance and saved by using the AMTI System Configuration program.

19.0 Function Definitions

The following Sections contain the definitions for each of the DLL functions. They are grouped according to following categories:

DLL Initialization Functions
Data Collection Functions
Apply and Save Functions
Signal Conditioner Configuration Functions
Signal Conditioner Mechanical Limits Functions
Platform Ordering Functions
Signal Conditioner Calibration Functions
Platform Calibration Functions
Signal Conditioner Hardware Functions

20.0 DLL Initialization Function Definitions

fmDLLInit

Description

This function initializes the DLL for all activities, and must be called first in any application program.

After calling ***fmDLLInit*** the program should either set a timer or sleep for 250 milliseconds, followed by a call to ***fmDLLIsDeviceInitComplete*** to see if the DLL is loaded and the devices ready. If ***fmDLLIsDeviceInitComplete*** returns 0 the initialization is not complete; reset the timer or go back to sleep and try again later.

When ***fmDLLInit*** is called, the DLL conducts a search for connected signal conditioners. For any connected signal conditioners it uploads the settings to the DLL for rapid access.

The DLL loads the last saved configuration file, AMTIUsbSetup.cfg, and compares the previous configuration against the current setup to detect whether all the signal conditioners are present. By calling ***fmDLLSetupCheck*** the application may determine whether the current setup matches the configuration file or whether changes have been made.

The DLL always uses the platform data collection order from the configuration file. It will maintain that dataset platform order even if some signal conditioners are not present.

Format

void fmDLLInit(void)

Related Functions

fmDLLIsDeviceInitComplete
fmDLLSetupCheck

fmDLLIsDeviceInitComplete

Description

This function works in conjunction with ***fmDLLInit***. After ***fmDLLInit*** has been called, call ***fmDLLIsDeviceInitComplete*** to see if the DLL has completed initialization. See ***fmDLLInit*** for more information.

Format

int fmDLLIsDeviceInitComplete(void)

Returns

Initialization status:

Returns	Description
0	Not completed initializing the DLL
1	The DLL is initialized, no signal conditioners are present
2	The DLL is initialized

Related Functions

fmDLLInit

fmDLLSetupCheck

Description

This function should be called after the DLL initialization has been completed and confirmed by ***fmDLLIsDeviceInitComplete*.** The function ***fmDLLSetupCheck*** compares the last saved DLL configuration file to the current DLL setup and notes any changes or discrepancies which may need attending.

Format

int fmDLLSetupCheck(void)

Returns

DLL setup status value:

Value	Description
0	No signal conditioners were found
1	The current setup is the same as the last saved configuration
211	The configuration file was not found
213	A configuration file was found but for the wrong version of the software
214	The configuration has changed: a different number of signal conditioners were detected than the previously saved setup
215	The configuration has changed: the serial numbers of the signal conditioners don't match the previously saved setup

Related Functions

fmDLLInit

fmDLLSetUSBPacketSize

Description

Set the size of a packet being sent from the signal conditioner to the PC.

The current size of a packet is 512 bytes. Each packet has 16 datasets, with 8 elements in each dataset. Each element is a 4-byte IEEE floating point value. The 8 elements consist of a dataset counter, 6 data channels, and a trigger channel.

Note that this size value has no connection with the dataset type set in ***fmDLLSetDataFormat.***

Format

void fmDLLSetUSBPacketSize(int size)

Arguments

Size in bytes of the packet. Currently this must always be set to 512.

fmDLLGetDeviceCount

Description

This function returns the current number of connected signal conditioners.

Format

int fmDLLGetDeviceCount(void)

Returns

Number of signal conditioners attached to the DLL:

Return	Description
0	No signal conditioners found
> 0	Number of signal conditioners found

Related Functions

fmDLLSelectDeviceIndex

fmDLLSelectDeviceIndex

Description

Before communicating with a specific signal conditioner, the device must be selected first. The ***fmDLLSelectDeviceIndex*** function selects a specific signal conditioner by its device index. Once a signal conditioner has been selected it may be accessed through any of the ***fm*** prefix type functions. The device indexes are ordered 0 to the number of signal conditioners minus one. They are always ordered in the platform data collection order stored in the DLL configuration file. Use ***fmDLLGetDeviceCount*** to know how many devices are connected.

To find out what signal conditioner is associated with a device index call ***fmGetAmplifierSerialNumber***.

The functions using the ***fmBroadcast*** or ***fmDLL*** prefix in their names do not require this function as they are general functions not specific to any signal conditioner. The ***fmBroadcast*** prefixed functions broadcast commands to all connected signal conditioners. The ***fmDLL*** prefixed functions are commands which concern DLL settings and are not specific to signal conditioners.

Format

void fmDLLSelectDeviceIndex(int index)

Arguments

The index of the signal conditioner to select for communication

Related Functions

fmDLLGetDeviceCount
fmDLLGetDeviceIndex

fmDLLGetDeviceIndex

Description

This function returns the device index of the currently selected signal conditioner. Use ***fmDLLSelectDeviceIndex*** to select a signal conditioner as the currently selected device.

Format

int fmDLLGetDeviceIndex(void)

Returns

The device index of the signal conditioner currently selected for communication

Related Functions

fmDLLSelectDeviceIndex

fmDLLSaveConfiguration

Description

This function saves the current DLL settings to a configuration file stored in the AMTI configuration directory. The configuration file is named AMTIUsbSetup.cfg.

The configuration file contains the following:

Global Settings	**Description , Range, or possible values**
The configuration file version number	101 (current version)
The signal conditioner count	0-16 (range of possible values)
The acquisition rate	10-2000 (range of possible values)
The run mode	0-4 (range of possible values)
The genlock state	0-2 (range of possible values)
Signal Conditioner Settings	**Always saved in data collection order**
Signal conditioner serial number	
Signal conditioner model name	
Platform serial number	
Platform model name	

Format

int fmDLLSaveConfiguration(void)

Returns

1 indicating a successful save operation, 0 if failed

Related Functions

fmDLLInit

fmDLLShutDown

Description

In order to shut down the DLL, ***fmDLLShutDown*** must be called. Calling this function terminates all running threads and performs cleanup for the DLL. After calling this command adequate time must be allotted for cleanup before closing the application. Although this time increases per signal conditioner, 500 msec should be more than adequate in all cases.

If the DLL will be reinitialized to search for additional signal conditioners and not terminating the application, just call ***fmDLLInit*** again. Do not call ***fmDLLShutDown***.

Format

int fmDLLShutDown(void)

Returns

1 indicating success

Related Functions

fmDLLInit

21.0 Data Collection Function Definitions

fmBroadcastRunMode

Description

This function sets the type of data output by all attached signal conditioners. For digital USB data the choices are English units, metric units, or bits. For analog data the choices are fully conditioned and MSA 6 compatible.

For digital data, if the units are metric the forces are newtons and the moments are newton-meters. If the units are English the forces are pounds and the moments are foot-pounds. If the units are bits the full scale range is ±16384 bits for all channels.

In MSA 6 compatible analog output mode the signal conditioner performs as a traditional analog amplifier with software selectable gains of 500, 1000, 2000, or 4000. Calibration corrections are applied for channel excitations, channel gains, cable length and bridge resistances.

In fully conditioned analog output mode calibration corrections are applied for excitations, channel gains, cable length and bridge resistances, and a platform sensitivity matrix is used to correct crosstalk. A user-supplied conversion factor is used to scale the analog outputs.

Format

void fmBroadcastRunMode(int mode)

Arguments

Data run mode to set:

Mode	Digital	Analog Volts
0	Metric	MSA 6 Compatible
1	Metric	Fully Conditioned
2	English	MSA 6 Compatible
3	English	Fully Conditioned
4	Bits	MSA 6 Compatible

Related Functions

fmDLLGetRunMode
fmSetDACSensitivityTable

fmDLLGetRunMode

Description

This function returns the last data output mode received by the DLL. For digital USB data, the choices are English units, metric units, or bits. For analog data the choices are MSA 6 compatible and fully conditioned.

For digital data, if the units are metric the forces are newtons and the moments are newton-meters. If the units are English the forces are pounds and the moments are foot-pounds. If the units are bits the full scale range is ±16384 bits

In MSA 6 compatible analog output mode the signal conditioner performs as a traditional analog amplifier with software selectable gains of 500, 1000, 2000, or 4000. Calibration corrections are applied for channel excitations, channel gains, cable length and bridge resistances.

In fully conditioned analog output mode calibration corrections are applied for excitations, channel gains, cable length and bridge resistances, and a platform sensitivity matrix is used to correct crosstalk. A user supplied conversion factor is used to scale the analog outputs.

Format

int fmDLLGetRunMode(void)

Returns

Current DLL data run mode:

Mode	Digital	Analog Volts
0	Metric	MSA 6 Compatible
1	Metric	Fully Conditioned
2	English	MSA 6 Compatible
3	English	Fully Conditioned
4	Bits	MSA 6 Compatible

Related Functions

fmBroadcastRunMode
fmGetRunMode

fmGetRunMode

Description

This function returns the run mode of the currently selected signal conditioner. For digital USB data, the choices are English units, metric units, or bits. For analog data the choices are MSA 6 compatible and fully conditioned.

For digital data, if the units are metric the forces are newtons and the moments are newton-meters. If the units are English the forces are pounds and the moments are foot-pounds. If the units are bits the full scale range is ±16384 bits

In MSA 6 compatible analog output mode the signal conditioner performs as a traditional analog amplifier with software selectable gains of 500, 1000, 2000, or 4000. Calibration corrections are applied for channel excitations, channel gains, cable length and bridge resistances.

In fully conditioned analog output mode calibration corrections are applied for excitations, channel gains, cable length and bridge resistances, and a platform sensitivity matrix is used to correct crosstalk. A user supplied conversion factor is used to scale the analog outputs.

Format

int fmGetRunMode(void)

Returns

Data run mode for the currently selected signal conditioner:

Mode	Digital	Analog Volts
0	Metric	MSA 6 Compatible
1	Metric	Fully Conditioned
2	English	MSA 6 Compatible
3	English	Fully Conditioned
4	Bits	MSA 6 Compatible

Related Functions

fmBroadcastRunMode
fmDLLGetRunMode

fmBroadcastGenlock

Description

This function sets the genlock mode for all attached signal conditioners.

In genlock mode, the signal conditioner collects a dataset only on the rising or falling edge of an electrical signal input into the genlock port of the signal conditioner. For more information refer to the section *Using the Genlock Signal* (section 12), and the signal conditioner user manual.

The function ***fmBroadcastStart*** must still be called to start data collection.

Format

void fmBroadcastGenlock (int mode)

Arguments

Genlock mode to set:

Mode	Description
0	Genlock off
1	Collect datasets on rising edge
2	Collect datasets on falling edge

Related Functions

fmDLLGetGenlock

fmDLLGetGenlock

Description

This function returns the last genlock configuration setting received by the DLL.

In genlock mode, the signal conditioner collects a dataset only on the rising or falling edge of an electrical signal input into the genlock port of the signal conditioner. For more information, refer to the section *Using the Genlock Feature* (section 12), and the signal conditioner user manual.

Format

int fmDLLGetGenlock(void)

Returns

Current DLL genlock mode:

Mode	Description
0	Genlock mode is off
1	Collect datasets on rising edge
2	Collect datasets on falling edge

Related Functions

fmBroadcastGenlock

fmBroadcastAcquisitionRate

Description

This function sets the acquisition rate, in datasets per second, for all connected signal conditioners.

Note that various models of AMTI signal conditioners support different acquisition rates; check the documentation for the particular devices being used to make sure a specific rate is supported. The table below represents the rates supported by the Gen5 signal conditioner.

If a signal conditioner is running in genlock mode, the actual data rate will be determined by the genlock pulse rate. However, the DLL requires that the nominal acquisition rate be set to a value at least as high as the highest rate to be received on the genlock port.

Format

void fmBroadcastAcquisitionRate(int rate)

Arguments

Acquisition rate, in datasets per second (Hz)

The following acquisition rates are permissible. If the acquisition rate is not recognized it will default to 500.

Acquisition Rates									
2000	1800	1500	1200	1000	900	800	600	500	450
400	360	300	250	240	225	200	180	150	125
120	100	90	80	75	60	50	45	40	30
25	20	15	10						

Related Functions

fmDLLGetAcquisitionRate
fmGetAcquisitionRate

fmDLLGetAcquisitionRate

Description

This function returns the last acquisition rate setting received by the DLL.

The acquisition rate is in datasets per second (Hz).

Format

int fmDLLGetAcquisitionRate(void)

Returns

Current DLL acquisition rate:

Acquisition Rates									
2000	1800	1500	1200	1000	900	800	600	500	450
400	360	300	250	240	225	200	180	150	125
120	100	90	80	75	60	50	45	40	30
25	20	15	10						

Related Functions

fmBroadcastAcquisitionRate
fmGetAcquisitionRate

fmGetAcquisitionRate

Description

This function returns the acquisition rate of the currently selected signal conditioner, in datasets per second (Hz).

Format

int fmGetAcquisitionRate(void)

Returns

Acquisition rate of the currently selected signal conditioner:

Acquisition Rates									
2000	1800	1500	1200	1000	900	800	600	500	450
400	360	300	250	240	225	200	180	150	125
120	100	90	80	75	60	50	45	40	30
25	20	15	10						

Related Functions

fmBroadcastAcquisitionRate
fmDLLGetAcquisitionRate

fmBroadcastStart

Description

Call this function to start data acquisition from all connected signal conditioners.

Any other SDK function called after ***fmBroadcastStart*** (except for ***fmBroadcastZero***) will automatically stop acquisition. This is to preserve signal conditioner synchronization. The formal stop acquisition function is ***fmBroadcastStop.***

Note that when genlock mode is active, data will not be sent from a signal conditioner to the SDK until the genlock pulses start to arrive at the signal conditioner's genlock port.

This function does not affect the analog outputs of a signal conditioner, as they are always active.

Format

void fmBroadcastStart(void)

Related Functions

fmBroadcastStop

fmBroadcastStop

Description

Call this function to stop data acquisition from all connected signal conditioners.

This function does not affect analog outputs, as they are always active.

Format

void fmBroadcastStop(void)

Related Functions

fmBroadcastStart

fmBroadcastZero

Description

This function tells all connected signal conditioners to zero their platforms. This function may be called before or after the data collection start command. Data collected while the zero process is taking place will consist of all zeros. If this function is called after the start command it will not cause data collection to stop unlike most other DLL functions.

Format

void fmBroadcastZero(void)

Related Functions

fmBroadcastStart
fmBroadcastStop

fmDLLPostDataReadyMessages

Description

This function enables or disables the sending of asynchronous messages indicating the availability of new data in the SDK.

There are three ways to receive data: by polling on a periodic basis, or by receiving data-ready messages at either the main application window or in a user thread each time a data buffer is ready. To enable the sending of data-ready messages to a window or user thread, the ***fmDLLPostDataReadyMessages*** function must be called with a 1 parameter. If polling is used, call ***fmDLLPostDataReadyMessages*** with a 0 parameter.

If messages are to be used, either the ***fmDLLPostUserThreadMessages*** or ***fmDLLPostWindowMessages*** function must be called to identify the recipient of the data-ready message. Messages cannot be posted to both the main application window and user threads at the same time.

Format

void fmDLLPostDataReadyMessages(int mode)

Arguments

Data ready message mode:

Mode	Description
0	Do not post data ready messages
1	Post data ready messages

Related Functions

fmDLLPostUserThreadMessages
fmDLLPostWindowMessages

fmDLLPostWindowMessages

Description

This function enables the posting of messages to an application window each time data is ready in the SDK. The function ***fmDLLPostWindowMessages*** passes the window's handle to the DLL. To enable messaging from the SDK, the function ***fmDLLPostDataReadyMessages*** must first be called with a non-zero value.

When using the Microsoft Foundation Classes, the *GetSafeHwnd* function will return a handle to the window.

The window message identifier sent by the SDK will always be WM_USER + 108 .

Format

void fmDLLPostWindowMessages(HWND handle)

Arguments

A handle to the window which will receive the data-ready messages

Related Functions

fmDLLPostDataReadyMessages
fmDLLPostUserThreadMessages

fmDLLPostUserThreadMessages

Description

This function enables the posting of messages to a user thread each time data is ready in the SDK. The function ***fmDLLPostUserThreadMessages*** passes the thread ID to the DLL. To enable messaging from the SDK, the function ***fmDLLPostDataReadyMessages*** must first be called with a non-zero value.

The thread message identifier sent by the SDK will always be WM_USER + 109 .

Format

void fmDLLPostUserThreadMessages(unsigned int threadID)

Arguments

ID of the thread to receive the messages (a CWinThread)

(Note: refer to m_nThreadID, a member of the CWinThread class)

Related Functions

fmDLLPostDataReadyMessages
fmDLLPostUserThreadMessages

fmDLLSetDataFormat

Description

There are two data formats. The user can receive each dataset in eight element or six element format. A six element dataset will consist solely of the force and moment channels. An 8 element dataset will have two additional channels, a dataset counter and a trigger state.

The dataset counter records the number of datasets after the start command was received. The dataset counter rolls over at 16,777,215 ($2^{24} – 1$). The trigger state will be either 0 or 1 depending on the electrical state of the trigger port on the signal conditioner.

Eight Element format

Channel	0	1	2	3	4	5	6	7
Element	Counter	Fx	Fy	Fz	Mx	My	Mz	Trigger

Six Element Format

Channel	0	1	2	3	4	5
Element	Fx	Fy	Fz	Mx	My	Mz

Format

void fmDLLSetDataFormat(int DataFormat)

Arguments

Packet data format:

Value	Description
0	Six channel format
1	Eight channel format

Related Functions

fmDLLTransferFloatData
fmDLLGetTheFloatDataLBVStyle

fmDLLTransferFloatData

Description

This function is used to transfer incoming data from the SDK to an application. If the development environment is Visual C++ or a similar C language this is the recommended data collection function. Development in Labview or Matlab may require the use of the ***fmDLLGetTheFloatDataLBVStyle*** function.

If data is available the referenced argument will return pointing to a full data buffer. The function does not return partial data buffers. **Note:** the buffers are allocated within the SDK and should not be allocated or deallocated at the application level.

The data buffer consists of 16 datasets from each connected signal conditioner. For one signal conditioner the data buffer consists of 16 datasets; for two signal conditioners there are data buffer consist of 16 datasets from signal conditioner one, 16 datasets from signal conditioner two, and so on. The datasets from multiple signal conditioners are interlaced, that is to say, the first dataset from each signal conditioner is found in sequence, followed by the second dataset from each signal conditioner, and so on.

A single dataset will consist of either 6 or 8 elements of data depending on the selected data format. Each data element is of the type float. The data format is set by calling ***fmDLLSetDataFormat***. A six element dataset will consist solely of the force and moment channels. An 8 element dataset will have two additional channels, a dataset counter and a trigger state. The dataset counter records the number of dataset after the start command was received. The trigger state will be either 0 or 1 depending on the electrical state of the trigger port on the signal conditioner.

Dataset Format								
Channel index	0	1	2	3	4	5	6	7
6 element	Fx	Fy	Fz	Mx	My	Mz		
8 element	Data counter	Fx	Fy	Fz	Mx	My	Mz	Trigger state

The order of the datasets in the data buffer must be considered. If a data buffer contains data from three signal conditioners the first dataset in the data buffer would be from the first signal conditioner, the second dataset from the second signal conditioner, and so on.

The size of the data buffer in floating point values is as follows:

DBS = the data buffer size
NCD = the number of channels per dataset
NOSC = the number of signal conditioners
16 = the number datasets from each signal conditioner in every packet

DBS = NCD * NOSC * 16

Format

int fmDLLTransferFloatData(float *&ptr)

Arguments

The function requires a reference to a pointer to floating point data. If data is available the pointer will return pointing to a full data buffer of type float. If no data is available the pointer will be unchanged.

Returns

The number of data sets available. Note that only one buffer will actually be accessible at the pointer, however, so ***fmDLLTransferFloatData*** should be called repeatedly until it returns a zero, indicating no more data are available.

Returns	Description
0	No new data available
> 0	Data returned at *ptr*

Related Functions

fmDLLGetDeviceCount
fmDLLSetDataFormat
fmDLLGetTheFloatDataLBVStyle

fmDLLGetTheFloatDataLBVStyle

Description

This function is used to transfer incoming data from the SDK to an application. This is the recommended data collection function for development using Labview or Matlab. For development using Visual C++ or some other C language the ***fmDLLTransferFloatData*** function should be used.

The difference between the two data transfer functions is that ***fmDLLGetTheFloatDataLBVStyle*** passes in an array to be filled by the SDK, while ***fmDLLTransferFloatData*** function simply returns a pointer to an array of floating point values allocated by the SDK.

If data are available the data argument will return with a full data buffer. The function does not return partial data buffers.

The data buffer consists of 16 datasets from each connected signal conditioner. For one signal conditioner the data buffer consists of 16 datasets, For two signal conditioners are data buffer consist of 16 datasets from signal conditioner one, and 16 datasets from signal conditioner two etc.

A single dataset will consist of either 6 or 8 elements of data depending on the selected data format. Each data element is of the type float. The data format is set by calling ***fmDLLSetDataFormat***. A six element dataset will consist solely of the force and moment channels. An 8 element dataset will have two additional channels, a dataset counter and a trigger state. The dataset counter records the number of dataset after the start command was received. The trigger state will be either 0 or 1 depending on the input state of the trigger port on the signal conditioner.

Dataset Format								
Channel index	0	1	2	3	4	5	6	7
6 element	Fx	Fy	Fz	Mx	My	Mz		
8 element	Data counter	Fx	Fy	Fz	Mx	My	Mz	Trigger state

The order of the datasets in the data buffer must be considered. If a data buffer contains data from three signal conditioners the first dataset in the data buffer would be from conditioner one, the second dataset from conditioner two etc.

The size of the data buffer in floating point values is as follows:

DBS = the data buffer size
NCD = the number of channels per dataset
NOSC = the number of signal conditioners
16 = the number datasets from each signal conditioner in every packet

DBS = NCD * NOSC * 16

Format

int fmDLLGetTheFloatDataLBVStyle(float *dptr, int size)

Arguments

A pointer to an array of type float which will be filled with data, and a size. The array size should be calculated according to the formula above.

If data is available the array will returned filled. If no data is available the array will return unchanged.

Returns

The number of data sets available. Note that only one buffer will actually be copied into the array, however, so ***fmDLLGetTheFloatDataLBVStyle*** should be called repeatedly until it returns a zero, indicating no more data is available.

Returns	Description
0	No new data available
> 0	Data returned in the array

Related Functions

fmDLLGetDeviceCount
fmDLLSetDataFormat
fmDLLTransferFloatData

22.0 Apply and Save Function Definitions

fmBroadcastResetSoftware

Description

This function resets the software state of all connected signal conditioners.

When new signal conditioner settings are downloaded, the changes are not implemented until this function is called. First make all the configuration changes (excitations, gains, acquisition rate, etc.), then call this function for the changes to be applied. After this function is called do not follow it directly with another function call as the signal conditioner will go into an indeterminate state while resetting; pause for at least 250 milliseconds.

This function does not save the changes to flash memory. Power cycling the signal conditioner will reset the last saved settings. Use ***fmBroadcastSave*** to store changes permanently.

Note: the function ***fmBroadcastAcquisitionRate*** does not need an ***fmResetsoftware*** function call to be applied. It is applied on the next ***fmBroadcastStart*** command.

Format

void fmBroadcastResetSoftware(void)

Related Functions

fmResetSoftware
fmBroadcastSave
fmSave

fmResetSoftware

Description

This function resets the software state of the currently selected signal conditioner

When new signal conditioner settings are downloaded, the changes are not implemented until this function is called. First make all the configuration changes (excitations, gains, acquisition rate, etc.), then call this function for the changes to be applied. After this function is called do not follow it directly with another function call as the signal conditioner will go into an indeterminate state while resetting; pause for at least 250 milliseconds.

This function does not save the changes to flash memory. Power cycling the signal conditioner will reset the last saved settings. Use ***fmSave*** to store changes permanently.

Note: the function ***fmBroadcastAcquisitionRate*** does not need an ***fmResetsoftware*** function call to be applied. It is applied on the next ***fmBroadcastStart*** command.

Format

void fmResetSoftware(void)

Related Functions

fmBroadcastResetSoftware
fmBroadcastSave
fmSave

fmBroadcastSave

Description

This function saves the current signal conditioner software settings to non-volatile memory for all attached signal conditioners. The saved settings are restored whenever the signal conditioner is powered on.

It takes a fair amount of time to write to flash. Do not send any signal conditioner commands for at least 250 milliseconds after calling this function as the signal conditioner is busy. The flash chip in the signal conditioner is rated for 20000 to 50000 writes, to it is best to make all necessary configuration changes and then save.

Format

void fmBroadcastSave(void)

Related Functions

fmBroadcastResetSoftware
fmResetSoftware
fmSave

fmSave

Description

This function saves the current signal conditioner software settings to non-volatile memory for the currently selected signal conditioner. The saved settings are restored whenever the signal conditioner is powered on.

It takes a fair amount of time to write to flash. Do not send any signal conditioner commands for at least 250 milliseconds after calling this function as the signal conditioner is busy. The flash chip in the signal conditioner is rated for 20000 to 50000 writes, to it is best to make all necessary configuration changes and then save.

Format

void fmSave(void)

Related Functions

fmBroadcastSave
fmBroadcastResetSoftware

fmApplyLimited

Description

This function saves the current hardware zero settings to the signal conditioner flash memory. When the signal conditioner is powered on these zero settings will automatically be loaded.

Format

void fmApplyLimited(void)

Related Functions

fmBroadcastSave
fmBroadcastResetSoftware

23.0 Signal Conditioner Configuration Function Definitions

fmSetCurrentGains

Description

This function sets the nominal gain levels for each force and moment channel on the currently selected signal conditioner.

The function requires a 6-element array of type long integer. The values for each element are shown in the table below:

Gain Setting	Corresponding Gain
0	500
1	1000
2	2000
3	4000

Note that the gain settings will not take effect until a call to the ***fmResetsoftware*** or ***fmBroadcastResetsoftware*** function is made.

Format

void fmSetCurrentGains(long *gains)

Arguments

A pointer to a 6-element array containing gain settings for each channel

Related Functions

fmGetCurrentGains

fmGetCurrentGains

Description

This function returns the nominal gain levels for each force and moment channel on the currently selected signal conditioner.

The function requires a 6-element array of type long integer. The values for each element are shown in the table below:

Gain Setting	Corresponding Gain
0	500
1	1000
2	2000
3	4000

Format

void fmGetCurrentGains(long *gains)

Arguments

A pointer to a 6-element array to receive gain settings for each channel

Related Functions

fmSetCurrentGains

fmSetCurrentExcitations

Description

This function sets the nominal excitation voltage levels for each force and moment channel on the currently selected signal conditioner.

The function requires a 6-element array of type long integer. The values for each element are shown in the table below:

Excitation Setting	Corresponding Excitation
0	2.5 volts
1	5.0 volts
2	10.0 volts

Note that the excitation settings will not take effect until a call to the ***fmResetsoftware*** or ***fmBroadcastResetsoftware*** function is made.

Format

void fmSetCurrentExcitations(long *excitations)

Arguments

A pointer to a 6-element array containing excitation settings for each channel

Related Functions

fmGetCurrentExcitations

fmGetCurrentExcitations

Description

This function returns the nominal excitation voltage levels for each force and moment channel on the currently selected signal conditioner.

The function requires a 6-element array of type long integer. The values for each element are shown in the table below:

Excitation Setting	Corresponding Excitation
0	2.5 volts
1	5.0 volts
2	10.0 volts

Format

void fmGetCurrentExcitations(long *excitations)

Arguments

A pointer to a 6-element array to receive excitation settings for each channel

Related Functions

fmSetCurrentExcitation

fmSetChannelOffsetsTable

Description

This function sets the channel offset parameters for each force and moment channel on the currently selected signal conditioner.

The channel offset parameter allows the user to offset the mechanical range of the signal conditioner to better adapt to the test being conducted. The channel offsets table is a 6 element array of type float. The value for each channel must lie between -0.99 and 0.99. Zero is the default value.

For example, consider a test that involves jumping on a platform. The expected physical range of channel Fz platform loading may be between -25 and 1500 newtons. Traditionally the electrical range of the signal conditioner would need to be -2000 to +2000 newtons in order to encompass the physical load range. However a better signal conditioner resolution could be accomplished by doubling the gain and offsetting the load range from -250 to 1750 newtons.

The channel offset table allows the user to set a zero offset. The tables below illustrates the effects of three different zero offset settings for a single channel on a signal conditioner with an electrical range configured for ±1000 newtons. The first table is referring to the digital outputs and the second table is referring to the analog outputs.

Digital Output in newtons			
channel offset	0	0.75	-0.75
maximum electrical range	1000	250	1750
zero load output	0	0	0
minimum electrical range	-1000	-1750	-250

Analog Output in Volts			
channel offset	0	0.75	-0.75
maximum output range	5.0	5.0	5.0
zero load output	0.0	3.75	-3.75
minimum output range	-5.0	-5.0	-5.0

The ***fmSetChannelOffsetsTable*** function downloads the channel offsets table to the currently selected signal conditioner. The array should be loaded in channel order as follows:

	Channel Offset Table					
Channel	Fx	Fy	Fz	Mx	My	Mz
Index	0	1	2	3	4	5
Range	Nominal Values					
(-0.99 to 0.99)	0.0	0.0	0.0	0.0	0.0	0.0

Note that the channel offset settings will not take effect until a call to the ***fmResetsoftware*** or ***fmBroadcastResetsoftware*** function is made.

Format

void fmSetChannelOffsetsTable(float *offsets)

Arguments

A pointer to a 6-element float array containing offsets for each channel

Related Functions

fmGetChannelOffsetsTable
fmUpdateMechanicalMaxAndMin
fmGetMechanicalMaxAndMin

fmGetChannelOffsetsTable

Description

This function returns the channel offset for each force and moment channel on the currently selected signal conditioner. Channel offset values should all lie between -0.99 and +0.99. The offset table returned is as follows:

	Channel Offset Table					
Channel	Fx	Fy	Fz	Mx	My	Mz
Index	0	1	2	3	4	5
Range	Nominal Values					
(-0.99 to 0.99)	0.0	0.0	0.0	0.0	0.0	0.0

Format

void fmGetChannelOffsetsTable(float *offsets)

Arguments

A pointer to a 6-element array to receive offsets for each channel

Related Functions

fmSetChannelOffsetsTable

fmSetCablelength

Description

This function sets the cable length factor for the currently selected signal conditioner. The value is defined in feet (1 foot = 30.5 cm), and should correspond to the length of the cable connecting the signal conditioner to its associated force platform or load cell.

The strength of the electrical signal will drop in proportion to the cable length. By setting the cable length the signal conditioner can apply a correction factor and produce more accurate force and moment values.

Note that the cable length setting will not take effect until a call to the ***fmResetsoftware*** or ***fmBroadcastResetsoftware*** function is made.

Format

void fmSetCableLength(float length)

Arguments

The cable length in feet between the platform and the signal conditioner

Related Functions

fmGetCableLength

fmGetCableLength

Description

This function returns the cable length factor for the currently selected signal conditioner. The value is defined in feet (1 foot = 30.5 cm), and should correspond to the length of the cable connecting the signal conditioner to its associated force platform or load cell.

The strength of the electrical signal will drop in proportion to the cable length. By setting the cable length the signal conditioner can apply a correction factor and produce more accurate force and moment values.

Format

float fmGetCableLength(void)

Return

The cable length in feet between the platform and the signal conditioner

Related Functions

fmSetCableLength

fmSetMatrixMode

Description

This function sets the matrix mode for the currently selected signal conditioner.

The inverted sensitivity matrix is a 36-element array of type float; it is used to eliminate crosstalk. It consists of calibration coefficients which convert microvolts to engineering units. Occasionally the user may only want to use the main diagonal terms as opposed to the full calibration matrix. See the function description for ***fmSetInvertedSensitivityMatrix*** for a full description of the inverted sensitivity matrix.

Note that the matrix mode setting will not take effect until a call to the ***fmResetsoftware*** or ***fmBroadcastResetsoftware*** function is made.

Format

void fmSetMatrixMode(long mode)

Arguments

A mode value representing the matrix mode:

Mode	Description
1	Use full matrix
0	Use main diagonal terms only

Related Functions

fmGetMatrixMode
fmSetInvertedSensitivityMatrix
fmGetInvertedSensitivityMatrix

fmGetMatrixMode

Description

This function returns the matrix mode for the currently selected signal conditioner.

The inverted sensitivity matrix is a 36-element array of type float; it is used to eliminate crosstalk. It consists of calibration coefficients which convert microvolts to engineering units. Occasionally the user may only want to use the main diagonal terms as opposed to the full calibration matrix. See the function description for ***fmSetInvertedSensitivityMatrix*** for a full description of the inverted sensitivity matrix.

Format

long fmGetMatrixMode(void)

Returns

A mode value representing the matrix mode:

Mode	Description
1	Use full matrix
0	Use main diagonal terms only

Related Functions

fmSetMatrixMode
fmSetInvertedSensitivityMatrix
fmGetInvertedSensitivityMatrix

fmSetPlatformRotation

Description

This function sets the platform rotation factor for the currently selected signal conditioner.

The rotation factor allows the signal conditioner to perform a rotational transformation on the data. Sometimes a platform must be rotated from its original orientation to get the cable connectors out of the way. This function allows the user to change the platform orientation while maintaining the X, Y axis orientation. The rotation must be entered in degrees (0 to 360). The default setting is zero.

The transformation does not apply to all run modes:

Output modes		Transformation applied
Digital	English	Yes
	Metric	Yes
	Bits	No
Analog	Fully Conditioned	Yes
	MSA 6 Compatible	No

Note that the platform rotation setting will not take effect until a call to the ***fmResetsoftware*** or ***fmBroadcastResetsoftware*** function is made.

Format

void fmSetPlatformRotation(float rotation)

Arguments

A rotation value in degrees, from 0 to 360

Related Functions

fmGetPlatformRotation

fmGetPlatformRotation

Description

This function returns the platform rotation factor for the currently selected signal conditioner.

The rotation will be from 0 to 360 degrees. The default rotation is zero.

Format

```
float  fmGetPlatformRotation(void)
```

Returns

A rotation value in degrees, from 0 to 360

Related Functions

fmSetPlatformRotation

24.0 Signal Conditioner Mechanical Limits Function Definitions

fmUpdateMechanicalMaxAndMin

Description

This function uploads the last calculated mechanical range of the signal conditioner under its current configuration to the DLL. The mechanical range is recalculated every time the signal conditioner is reset. The functions ***fmBroadcastResetSoftware*** and ***fmResetSoftware*** reset the signal conditioner.

NOTE: This function is uploads the currently configured mechanical limits of the signal conditioner, not that of the attached platform.

Format

void fmUpdateMechanicalMaxAndMin(void)

Related Functions

fmGetMechanicalMaxAndMin

fmGetMechanicalMaxAndMin

Description

This function retrieves the mechanical maximum and minimum for each channel under the current signal conditioner configuration. The mechanical max and min table is a 12 element array of type float. The array will be loaded in row, column order, the first row being mechanical maximums and the second row being mechanical minimums. The values will be in either English or metric units depending on the current run mode selection.

The function ***fmUpdateMechanicalMaxAndMin*** must be called prior to calling this function unless no parameters have been modified after initializing the DLL. The program should wait a short period before calling ***fmGetMechanicalMaxAndMin*** to give the signal conditioner time to calculate and upload the values. If the upload is not complete, this function will return a zero indicating that the data are not correct.

NOTE: This function retrieves the currently configured mechanical limits of the signal conditioner, not that of the attached platform.

Format

int fmGetMechanicalMaxAndMin(float *data)

Parameter

A pointer to a 12-element array to contain the mechanical maximum and minimum data

Return

The status of the upload process

Return	Description
0	The DLL is currently uploading the mechanical range data after a call to ***fmUpdateMechanicalMaxAndMin*** - wait and try again
1	The array contains the last updated mechanical range data

Related Functions

fmUpdateMechanicalMaxAndMin
fmDLLGetRunMode

fmUpdateAnalogMaxAndMin

Description

This function uploads the last calculated analog output range of the signal conditioner under its current configuration to the DLL. The mechanical range is recalculated every time the signal conditioner is reset. The functions ***fmBroadcastResetSoftware*** and ***fmResetSoftware*** reset the signal conditioner. Upon DLL initialization ***fmUpdateAnalogMaxAndMin*** is automatically called.

The maximum output is calculated by dividing the channel DAC sensitivity value by 5.0. The minimum output is calculated by dividing the channel DAC sensitivity by -5.0. The DAC sensitivity values are always in millivolts per pound for forces and millivolts per inch-pound for moments.

If the analog output range is greater than the configured signal conditioner mechanical range, the analog output range will be constrained by the mechanical range.

This function is for informational purposes only.

NOTE: When the analog outputs are set to MSA 6 compatible mode this function is indeterminate. The analog output range is then nominally the same as the electrical range.

Format

void fmUpdateAnalogMaxAndMin(void)

Related Functions

fmGetAnalogMaxAndMin
fmUpdateMechanicalMaxAndMin
fmGetMechanicalMaxAndMin

fmGetAnalogMaxAndMin

Description

This function retrieves the analog output range of the signal conditioner from the DLL. The analog output maximum and minimum table is a 12 element array of type float. The first 6 elements are the analog maximums; the last 6 elements are the analog minimums. The values will be in either English or metric units depending on the current run mode selection.

The function ***fmUpdateAnalogMaxAndMin*** must be called prior to ***fmGetAnalogMaxAndMin*** unless the DAC Sensitivities have not been modified after initializing the DLL.

NOTE: When the analog outputs are set to MSA 6 compatible mode this function is indeterminate. The analog output range is then nominally the same as the electrical range.

Format

int fmGetAnalogMaxAndMin(float *data)

Parameter

A pointer to a 12-element array to contain the analog output maximum and minimum data

Returns

The status of the upload process

Return	Description
0	The DLL is currently uploading the analog range data after a call to ***fmUpdateAnalogMaxAndMin*** `- wait and try again`
1	The array contains the last updated analog output range data

Related Functions

fmUpdateAnalogMaxAndMin
fmDLLGetRunMode
fmSetDACSensitivityTable

25.0 Platform Ordering Function Definitions

fmDLLSetPlatformOrder

Description

This function sets a new platform data collection order based on the current ordering. The platform order is important in analyzing output data in order to match up digital data with physical positions of the platforms when more than one are used.

To use this function, the current order of the platforms and the identity of the platforms and/or their associated signal conditioners must be known. To determine the order of the platforms do the following. First call ***fmDLLGetDeviceCount*** to get the number of signal conditioners. Then create a loop to cycle through the signal conditioners. Use the functions ***fmDLLSelectDeviceIndex*** and ***fmGetAmplifierSerialNumber*** to get the serial number of each signal conditioner. Once the serial number for each device index is known, simply map the new desired device index order into an array of integers and pass a pointer to the array into the ***fmDLLSetPlatformOrder*** function.

Format

void fmDLLSetPlatformOrder(int *indexarray)

Arguments

An array of platform indices referring to the current platform ordering. The array must be of size at least equal to the current number of attached signal conditioners. Each element of the array contains a current signal conditioner device index, and after ***fmDLLSetPlatformOrder*** is called the index of each array element will become the new device index.

Related Functions

fmDLLGetPlatformOrder

fmBroadcastPlatformOrderingThreshold

Description

This function sets the threshold for Fz data to be used for automated ordering of platforms (see Section 11 for details).

When auto-ordering is used the DLL is set to detect when each platform is stepped on. The order in which the platforms are stepped on determines the platform order in the collected data. The platform threshold value is a value which is crossed when a user steps on the platform. Be sure it is not set so low as to be triggered by vibration or noise.

The threshold value is defined in bits. The full scale range of the signal conditioner in bits is ±16384, though a useful threshold would nearly always be a positive value, since zero indicates an unloaded state and Fz increases in the positive direction as force is applied to the top of the platform.

Format

void fmBroadcastPlatformOrderingThreshold(float value)

Arguments

Threshold value in bits

Related Functions

fmDLLStartPlatformOrdering
fmDLLCancelPlatformOrdering

fmDLLStartPlatformOrdering

Description

This function initiates the automated platform ordering procedure (see Section 11 for details).

When ***fmDLLStartPlatformOrdering*** is called the DLL continuously polls all platforms to detect if the Fz force threshold (as set in ***fmBroadcastPlatformOrderingThreshold***) is crossed. The order in which each platform's threshold is crossed determines the data collection order.

Once all platforms have been detected the new platform order is set.

Format

void fmDLLStartPlatformOrdering(void)

Related Functions

fmBroadcastPlatformOrderingThreshold
fmDLLIsPlatformOrderingComplete
fmDLLCancelPlatformOrdering

fmDLLIsPlatformOrderingComplete

Description

This function is called to ascertain that the automated platform ordering procedure is complete (see Section 11 for details).

After ***fmDLLStartPlatformOrdering*** is called it is expected that the interactive user will apply force to the platforms in the desired order. The data will be read by the DLL to establish the platform order. When the last attached signal conditioner has indicated an ordering force, the ordering is complete and ***fmDLLIsPlatformOrderingComplete*** will return 1. This routine may be called in a periodic loop to indicate that the process is finished.

Format

int fmDLLIsPlatformOrderingComplete(void)

Returns

Auto-ordering procedure status:

Status	Description
0	Platform Ordering is not complete
1	Platform Ordering is complete

Related Functions

fmBroadcastPlatformOrderingThreshold
fmDLLStartPlatformOrdering
fmDLLCancelPlatformOrdering

fmDLLCancelPlatformOrdering

Description

This function is called to cancel the automated platform ordering process before it has completed, presumably if the user has determined that something is incorrect (a bad threshold value, for example). If the process is not cancelled, the DLL will continue to monitor all platform channels to watch for Fz data, and normal force measurement will be impossible.

Format

void fmDLLCancelPlatformOrdering(void)

Related Function

fmBroadcastPlatformOrderingThreshold
fmDLLStartPlatformOrdering
fmDLLIsPlatformOrderingComplete

26.0 Signal Conditioner Calibration Function Definitions

fmGetProductType

Description

This function returns a model type value specific to the currently selected signal conditioner.

The model type allows application code to check attached signal conditioners for possible special handling due to variations in product capabilities. Current product types supported are:

100 - Gen 5 signal conditioner
300 - Optima signal conditioner
400 - Hall-effect integrated Optima platform (Accusway/Accugait)

For the specific characteristics of the various products, please refer to the particular manuals for those products.

Format

long fmGetProductType(void)

Returns

A type value specific to the signal conditioner product line

Related Functions

fmGetAmplifierModelNumber
fmGetAmplifierSerialNumber

fmGetAmplifierModelNumber

Description

This function returns the model number (really a name) of the currently selected signal conditioner. The model name reflects the general class of the signal conditioner.

The model name may contain up to 16 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetAmplifierModelNumber(char *buf)

Arguments

A pointer to a character array to hold the model number (minimum size: 16)

Related Functions

fmGetAmplifierSerialNumber

fmGetAmplifierSerialNumber

Description

This function retrieves the serial number of the currently selected signal conditioner.

The serial number may contain up to 16 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetAmplifierSerialNumber(char *buf)

Arguments

A pointer to a character array to hold the serial number (minimum size: 16)

Related Functions

fmGetAmplifierModelNumber

fmGetAmplifierFirmwareVersion

Description

This function retrieves a string identifying the version of the internal firmware installed in the currently selected signal conditioner.

The firmware version string may contain up to 16 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetAmplifierFirmwareVersion(char *buf)

Arguments

A pointer to a character array to hold the firmware version string (minimum size: 16)

Related Functions

fmGetAmplifierSerialNumber

fmGetAmplifierDate

Description

This function retrieves the last calibration date of the currently selected signal conditioner.

The date string may contain up to 12 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetAmplifierDate(char *buf)

Arguments

A pointer to a character array to hold the date (minimum size: 12)

Related Functions

fmGetAmplifierModelNumber
fmGetAmplifierSerialNumber

fmGetGainTable

Description

This function retrieves the gain correction table of the currently selected signal conditioner.

The gain table is a 24-element array of type float. The array will be retrieved in row, column order as in the table below. The values represent the accurately calibrated values of each gain channel and gain level; these will be will be close to but not precisely the same as the nominal values shown in the table. The calibrated values are used to correct the engineering output values from the signal conditioner with respect to the internal amplification circuits.

Format

void fmGetGainTable(float *data)

Arguments

A pointer to a 24-element array to contain calibrated gain values:

	Gain Table					
Channel	**0**	**1**	**2**	**3**	**4**	**5**
Gain	**Nominal Values**					
500	1.0	1.0	1.0	1.0	1.0	1.0
1000	2.0	2.0	2.0	2.0	2.0	2.0
2000	4.0	4.0	4.0	4.0	4.0	4.0
4000	8.0	8.0	8.0	8.0	8.0	8.0

Related Functions

fmSetCurrentGains
fmGetCurrentGains

fmGetExcitationTable

Description

This function retrieves the excitation correction table of the currently selected signal conditioner.

The excitation table is an 18-element array of type float. The array will be retrieved in row, column order as in the table below. The values represent the accurately calibrated values of each excitation level on each channel; these will be will be close to but not precisely the same as the nominal values shown in the table. The calibrated values are used to correct the engineering output values from the signal conditioner with respect to the internal amplification circuits.

Format

void fmGetExcitationTable(float *data)

Arguments

A pointer to an 18-element array to contain calibrated excitation values:

	Excitation Table					
Channel	**0**	**1**	**2**	**3**	**4**	**5**
Excitation	**Nominal Values**					
2.5	2.5	2.5	2.5	2.5	2.5	2.5
5.0	5.0	5.0	5.0	5.0	5.0	5.0
10.0	10.0	10.0	10.0	10.0	10.0	10.0

Related Functions

fmSetCurrentExcitations
fmGetCurrentExcitations

fmGetDACGainsTable

Description

This function retrieves the digital to analog converter (DAC) gain correction table for the currently selected signal conditioner.

The gain correction table is a 6-element array of type float. The array will be retrieved in channel order as in the table below. The values represent the accurately calibrated values of analog conversion gain for each channel; these will be will be close to but not precisely the same as the nominal values shown in the table. The calibrated values are used to correct the engineering unit analog output values from the signal conditioner with respect to the internal amplification circuits.

Format

void fmGetDACGainsTable(float *data)

Arguments

A pointer to a 6-element array to contain calibrated analog conversion gain values:

	DAC Gains Table					
Channel	**0**	**1**	**2**	**3**	**4**	**5**
	Nominal Values					
	-2.49423	-2.49423	-2.49423	-2.49423	-2.49423	-2.49423

Related Functions

fmGetDACOffsetTable
fmSetDACSensitivityTable
fmGetDACSensitivities

fmGetDACOffsetTable

Description

This function retrieves the digital to analog converter (DAC) offset correction table for the currently selected signal conditioner.

The offset correction table is a 6-element array of type float. The array will be retrieved in channel order as in the table below. The values represent the accurately calibrated values of analog conversion zero offset for each channel; these will be will be close to but not precisely the same as the nominal values shown in the table. The calibrated values are used to correct the engineering unit analog output values from the signal conditioner with respect to the internal amplification circuits.

Format

void fmGetDACOffsetTable(float *data)

Arguments

A pointer to a 6-element array to contain calibrated analog conversion offset values:

	DAC Offset Table					
Channel	0	1	2	3	4	5
	Nominal Values					
	0.0	0.0	0.0	0.0	0.0	0.0

Related Functions

fmGetDACGainsTable
fmSetDACSensitivityTable
fmGetDACSensitivities

fmSetDACSensitivityTable

Description

This function sets the digital to analog conversion (DAC) sensitivity table.

The DAC sensitivity table contains conversion factors, one for each channel. This conversion factor is used to convert internally calculated digital force and moment values into analog output volts at a user-defined proportional voltage. The force channel conversions are always millivolts per pound. The moment channel conversions are always millivolts per inch-pound.

The DAC sensitivity Table is only applied when the analog outputs are set to fully conditioned mode. These same conversion values or the metric equivalents must be entered in the user application to convert the analog signal to engineering units within the PC.

Format

void fmSetDACSensitivityTable(float *data)

Arguments

A pointer to a 6-element array containing the DAC sensitivity values, one per channel

Related Functions

fmGetDACSensitivities

fmGetDACSensitivities

Description

This function retrieves the digital to analog conversion (DAC) sensitivity table.

The DAC sensitivity table contains conversion factors, one for each channel. This conversion factor is used to convert internally calculated digital force and moment values into analog output volts at a user-defined proportional voltage. The force channel conversions are always millivolts per pound. The moment channel conversions are always millivolts per inch-pound.

The DAC sensitivity Table is only applied when the analog outputs are set to fully conditioned mode. These same conversion values or the metric equivalents must be entered in the user application to convert the analog signal to engineering units within the PC.

Format

void fmGetDACSensitivities(float *data)

Arguments

A pointer to a 6-element array to contain the DAC sensitivity values, one per channel

Related Functions

fmSetDACSensitivityTable

fmGetADRef

Description

This function retrieves the nominal analog to digital reference voltage value for the currently selected signal conditioner.

Format

float fmGetADRef(void)

Returns

The nominal reference voltage value for the currently selected signal conditioner

27.0 Platform Calibration Function Definitions

fmSetPlatformDate

Description

This function sets the platform calibration date stored in the currently selected signal conditioner. This date should reflect the latest calibration of the platform.

The date string may contain up to 12 characters. Any characters beyond that limit will be ignored.

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite this parameter with data loaded from the smart platform.

Format

void fmSetPlatformDate (char *buf)

Arguments

A pointer to a character array holding the date (maximum size: 12)

Related Functions

fmGetPlatformDate

fmGetPlatformDate

Description

This function retrieves the platform calibration date stored in the currently selected signal conditioner.

The date string may contain up to 12 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetPlatformDate(char *buf)

Arguments

A pointer to a character array to hold the date (minimum size: 12)

Related Functions

fmSetPlatformDate

fmSetPlatformModelNumber

Description

This function sets the platform model number stored in the currently selected signal conditioner.

The model number may contain up to 28 characters. Any characters beyond that limit will be ignored.

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite this parameter with data loaded from the smart platform.

Format

void fmSetPlatformModelNumber(char *buf)

Arguments

A pointer to a character array holding the model number (maximum size: 28)

Related Functions

fmGetPlatformModelNumber

fmGetPlatformModelNumber

Description

This function retrieves the platform model number stored in the currently selected signal conditioner.

The platform model number may contain up to 28 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetPlatformModelNumber(char *buf)

Arguments

A pointer to a character array to hold the model number (minimum size: 28)

Related Functions

fmSetPlatformModelNumber

fmSetPlatformSerialNumber

Description

This function sets the platform serial number stored in the currently selected signal conditioner.

The serial number may contain up to 16 characters. Any characters beyond that limit will be ignored.

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite this parameter with data loaded from the smart platform.

Format

void fmSetPlatformSerialNumber(char *buf)

Arguments

A pointer to a character array holding the serial number (maximum size: 16)

Related Functions

fmGetPlatformSerialNumber

fmGetPlatformSerialNumber

Description

This function retrieves the platform serial number stored in the currently selected signal conditioner.

The platform serial number may contain up to 16 characters, so the array specified in the parameter should be of at least that length.

Format

void fmGetPlatformSerialNumber(char *buf)

Arguments

A pointer to a character array to hold the serial number (minimum size: 16)

Related Functions

fmSetPlatformSerialNumber

fmSetPlatformLengthAndWidth

Description

This function sets the platform length and width stored in the currently selected signal conditioner.

The length and width are stored in text format, 16 characters per field. Any characters beyond that limit will be ignored. The signal conditioner has no internal use for this information.

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite these parameters with data loaded from the smart platform.

Format

void fmSetPlatformLengthAndWidth(char *length, char *width)

Arguments

Two pointers to character arrays holding the length and width strings (maximum size: 16)

Related Functions

fmGetPlatformLengthAndWidth

fmGetPlatformLengthAndWidth

Description

This function retrieves the platform length and width stored in the currently selected signal conditioner.

The length and width are stored in text format, 16 characters per field, so the arrays specified in the parameters should be of at least that length.

Length and width data are calibrated at the factory in inches (1 inch = 25.4 mm).

Format

void fmGetPlatformLengthAndWidth(char *length, char *width)

Arguments

Two pointers to character arrays to hold the length and width strings (minimum size: 16)

Related Functions

fmSetPlatformLengthAndWidth

fmSetPlatformXYZOffsets

Description

This function sets the X, Y, and Z platform offsets stored in the currently selected signal conditioner.

Each platform has an electrical center which has a physical location somewhere within the platform, near the physical center of the platform surface. This electrical center is calibrated at the factory, and can be used to more accurately define the center of pressure of a force applied to the force platform. These offsets represent that electrical center location as offset from the physical platform origin. For more information refer to the platform calibration information and manual. The signal conditioner has no internal use for these values.

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite these parameters with data loaded from the smart platform.

Format

void fmSetPlatformXYZOffsets(float *data)

Arguments

A pointer to a 3-element array containing the platform electrical center coordinates:

Array Element	0	1	2
Value	X offset	Y offset	Z offset

Related Functions

fmGetPlatformXYZOffsets

fmGetPlatformXYZOffsets

Description

This function retrieves the X, Y, and Z platform offsets stored in the currently selected signal conditioner.

Each platform has an electrical center which has a physical location somewhere within the platform, near the physical center of the platform surface. This electrical center is calibrated at the factory, and can be used to more accurately define the center of pressure of a force applied to the force platform. These offsets represent that electrical center location as offset from the physical platform origin. For more information refer to the platform calibration information and manual.

Offset data are calibrated at the factory in inches (1 inch = 25.4 mm).

Format

void fmGetPlatformXYZOffsets(float *data)

Arguments

A pointer to a 3-element array to contain the platform electrical center coordinates:

Array Element	0	1	2
Value	X offset	Y offset	Z offset

Related Functions

fmSetPlatformXYZOffsets

fmSetPlatformCapacity

Description

This function sets the platform capacities stored in the currently selected signal conditioner.

The function requires a 6 element array of type float as shown in the example below. These parameters are always shipped with the platform calibration, and are provided for informational use only.

	Platform Capacity Table					
Channel	**Fx**	**Fy**	**Fz**	**Mx**	**My**	**Mz**
Units	**Lb**	**lb**	**Lb**	**In-lb**	**In-lb**	**In-lb**
Index	**0**	**1**	**2**	**3**	**4**	**5**
	Nominal Capacity Values for a 1000 lb OR6-7					
Capacity	500	500	1000	10000	10000	5000

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite these parameters with data loaded from the smart platform.

Format

void fmSetPlatformCapacity(float *data)

Arguments

A pointer to a 6-element array containing the platform channel capacities

Related Functions

fmGetPlatformCapacity

fmGetPlatformCapacity

Description

This function retrieves the platform capacities stored in the currently selected signal conditioner.

The function requires a 6 element array of type float as shown in the example below. These parameters are always shipped with the platform calibration, and are provided for informational use only.

Offset data are calibrated at the factory in English units.

	Platform Capacity Table					
Channel	**Fx**	**Fy**	**Fz**	**Mx**	**My**	**Mz**
Units	**Lb**	**lb**	**lb**	**In-lb**	**In-lb**	**In-lb**
Index	**0**	**1**	**2**	**3**	**4**	**5**
	Nominal Capacity Values for a 1000 lb OR6-7					
Capacity	500	500	1000	10000	10000	5000

Format

void fmGetPlatformCapacity(float *data)

Arguments

A pointer to a 6-element array to contain the platform channel capacities

Related Functions

fmSetPlatformCapacity

fmSetPlatformBridgeResistance

Description

This function sets the bridge resistance array stored in the currently selected signal conditioner.

The function requires a 6 element array of type float as shown in the example below. These parameters are always shipped with the platform calibration, and are used to calculate accurate force and moment values in engineering units.

These parameters are stored in the signal conditioner in ohms.

	Platform Capacity Table					
Channel	**Fx**	**Fy**	**Fz**	**Mx**	**My**	**Mz**
Units	**ohms**	**ohms**	**Ohms**	**ohms**	**ohms**	**ohms**
Index	**0**	**1**	**2**	**3**	**4**	**5**
	Nominal Bridge Resistance Values for an OR6-7					
Bridge Resistance	700	700	350	700	700	700

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite these parameters with data loaded from the smart platform.

Format

void fmSetPlatformBridgeResistance(float *data)

Arguments

A pointer to a 6-element array containing the platform bridge resistances

Related Functions

fmGetPlatformBridgeResistance

fmGetPlatformBridgeResistance

Description

This function retrieves the bridge resistance array stored in the currently selected signal conditioner.

The function requires a 6 element array of type float as shown in the example below. These parameters are always shipped with the platform calibration, and are used to calculate accurate force and moment values in engineering units.

These parameters are stored in the signal conditioner in ohms.

	Platform Bridge Resistance Table					
Channel	**Fx**	**Fy**	**Fz**	**Mx**	**My**	**Mz**
Units	**ohms**	**ohms**	**ohms**	**ohms**	**ohms**	**ohms**
Index	**0**	**1**	**2**	**3**	**4**	**5**
	Nominal Bridge Resistance Values for an OR6-7					
Bridge Resistance	700	700	350	700	700	700

Format

void fmGetPlatformBridgeResistance(float *data)

Arguments

A pointer to a 6-element array to contain the platform bridge resistances

Related Functions

fmSetPlatformBridgeResistance

fmSetInvertedSensitivityMatrix

Description

This function sets the inverted sensitivity matrix stored in the currently selected signal conditioner.

The inverted sensitivity matrix is a 36 element array of type float, stored in row, column order as shown in the example table below. The inverted sensitivity array is used to convert micro volts to engineering units and eliminate crosstalk.

The English version of the array should be used. This matrix is supplied with every platform shipped.

	Sample inverted Sensitivity Matrix					
Channel	**0**	**1**	**2**	**3**	**4**	**5**
	VFx	**VFy**	**VFz**	**VMx**	**VMy**	**VMz**
	Input to channel i(lb,in-lb) is B(I,j)times the electrical output j(uV,Vex)					
	BP 400600-2000					
Fx	0.6519	-0.0068	-0.0019	0.0009	-0.0017	-0.0003
Fy	0.0090	0.6515	-0.0037	0.0009	0.0005	0.0010
Fz	0.0018	0.0017	2.5523	-0.0062	0.0001	0.0026
Mx	-0.0044	-0.0032	0.0003	12.8281	0.0108	-0.0138
My	0.0725	-0.0032	0.0003	0.0058	10.1358	-0.0140
Mz	0.0649	0.0821	0.0792	0.0123	0.0340	5.4451

NOTE: If the signal conditioner is attached to a smart platform, it will overwrite these parameters with data loaded from the smart platform.

Format

void fmSetInvertedSensitivityMatrix(float *data)

Arguments

A pointer to a 36-element array containing the platform inverted sensitivity matrix data

Related Functions

fmSetInvertedSensitivityMatrix

fmGetInvertedSensitivityMatrix

Description

This function retrieves the inverted sensitivity matrix stored in the currently selected signal conditioner.

The inverted sensitivity matrix is a 36 element array of type float, stored in row, column order as shown in the example table below. The inverted sensitivity array is used to convert micro volts to engineering units and eliminate crosstalk.

The array is stored in the signal conditioner in English units. A matrix is supplied for every platform shipped.

	Sample inverted Sensitivity Matrix					
Channel	**0**	**1**	**2**	**3**	**4**	**5**
	VFx	**VFy**	**VFz**	**VMx**	**VMy**	**VMz**
	Input to channel i(lb,in-lb) is B(I,j)times the electrical output j(uV,Vex)					
	BP 400600-2000					
Fx	0.6519	-0.0068	-0.0019	0.0009	-0.0017	-0.0003
Fy	0.0090	0.6515	-0.0037	0.0009	0.0005	0.0010
Fz	0.0018	0.0017	2.5523	-0.0062	0.0001	0.0026
Mx	-0.0044	-0.0032	0.0003	12.8281	0.0108	-0.0138
My	0.0725	-0.0032	0.0003	0.0058	10.1358	-0.0140
Mz	0.0649	0.0821	0.0792	0.0123	0.0340	5.4451

Format

void fmGetInvertedSensitivityMatrix(float *data)

Arguments

A pointer to a 36-element array to contain the platform inverted sensitivity matrix data

Related Functions

fmSetInvertedSensitivityMatrix

28.0 Signal Conditioner Hardware Function Definitions

fmSetBlink

Description

This function tells the currently selected signal conditioner to blink its front panel lamp.

The amber light will blink for ten seconds. This can be useful if there are several signal conditioners attached with complex wiring and it is necessary to determine which is which.

Note that the front panel lamp will remain in the lit state in the case that a correctly wired force platform is not attached to the signal conditioner.

Format

void fmSetBlink(void)

fmResetHardware

Description

This function power cycles the currently selected signal conditioner. The signal conditioner will not lose its USB connection during this process.

Format

void fmResetHardware(void)

Related Functions

fmBroadcastResetSoftware
fmResetSoftware

fmBroadcastResetUSB

Description

This function resets the USB pipes from the PC to the signal conditioner for all connected signal conditioners.

This function call may be useful if there is a problem with the USB connection of several signal conditioners. It should not be necessary in normal operation.

Format

void fmBroadcastResetUSB(void)

29.0 Sample Code

The following sample code is all written in the Microsoft Foundation Classes using Visual Studio 2008. To use any DLL functions AMTIUSBDeviceDefinitions.h must be included as a class header.

```
#include "AMTIUSBDeviceDefinitions.h"
```

DLL Initialization Using a Sleep Statement

```
#include "AMTIUSBDeviceDefinitions.h"

void USBDeviceDlg::InitializeDeviceDLL(void)
{
	fmDLLInit();
	Sleep(250);
	int countdown = 20;
	while(fmDLLIsDeviceInitComplete() == 0)
	{
		Sleep(250);
		if ( countdown-- <= 0 )
		{
			//  Handle a timeout error here
		}
	}
	ret = fmDLLSetupCheck();

	// If return is not 1 configuration has changed
	// Go to funtion description for more information

	ConfigureDataCollection();
}
```

DLL Initialization Using an MFC timer

```
#include "AMTIUSBDeviceDefinitions.h"

void USBDeviceDlg::InitializeDeviceDLL(void)
{

	fmDLLInit();
	TimerID = SetTimer( TimerIDUSBInit, 250, NULL );
}

void USBDeviceDlg::OnTimer(UINT_PTR nIDEvent)
{
	int ret;
	int i;
	ret = 0;

	if(nIDEvent == TimerIDUSBInit)
	{

		ret = fmDLLIsDeviceInitComplete();
		// ret = 0 Wait still initializing
		// ret = 1 Finished, No devices found
		// ret = 2 Finished, device found
		if(ret != 0 )
		{
			KillTimer(TimerIDUSBInit);
			ret = fmDLLSetupCheck();
			// If return is not 1 configuration has changed
			// Go to funtion description for more information

			ConfigureDataCollection();

		}
		else
		{
			timerCount++;
			if(timerCount > MAX_TIMER_ITERATIONS)
			{
				KillTimer(TimerIDUSBInit);
				AfxMessageBox("USB DLL Timeout");

			}
			else
			{
				SetTimer( TimerIDUSBInit, 250, NULL );
			}
		}
	}

}
```

The Acquisition Rate being Broadcast to the Signal Conditioners

```
#include "AMTIUSBDeviceDefinitions.h"

int acqRate;

acqRate = 1000;
fmBroadcastAcquisitionRate(acqRate);
```

The Platforms being Zeroed

```
#include "AMTIUSBDeviceDefinitions.h"

fmBroadcastZero();
```

Starting Acquisition

```
#include "AMTIUSBDeviceDefinitions.h"

fmBroadcastStart();
```

Stopping Acquisition

```
#include "AMTIUSBDeviceDefinitions.h"

fmBroadcastStop();
```

An MFC dialog class being setup to do data collection using windows messaging

```
#include "AMTIUSBDeviceDefinitions.h"

//Standard Data collection settings that should be set
void USBDeviceDlg::ConfigureDataCollection(void)
{

	HWND h_Wnd;

	//Decided to post messages to a window
	fmDLLPostDataReadyMessages(TRUE);
	h_Wnd = GetSafeHwnd();
	fmDLLPostWindowMessages((HWND) h_Wnd);

	fmDLLSetUSBPacketSize(512 ); // Set the packet size to 512

	fmBroadcastGenlock(0); // Make sure Genlock is off

	fmBroadcastRunMode(0); // Set collection mode to metric

	fmBroadcastResetSoftware(); //Apply the settings
	Sleep(250);

}
```

Windows messaging being set up to do data collection

```
//Setting up the data collection function in the dialog class header message
map to receive widows messages.

Must always use (WM_USER + 108) as an identifier
#define WM_BUFFER_READY                        (WM_USER + 108) //Add this line
…….
…….
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
long OnBufferReady( DWORD wParam, long lParam );//Add this line
DECLARE_MESSAGE_MAP()
…..
…….


//Setting up the data collection function in the dialog class main body
message map
…….
…….
ON_BN_CLICKED(IDC_B_START, &CdummygenDlg::OnBnClickedBStart)
	ON_BN_CLICKED(IDC_B_STOP, &CdummygenDlg::OnBnClickedBStop)
	ON_MESSAGE( WM_BUFFER_READY,(LRESULT(AFX_MSG_CALL CWnd::*)(WPARAM,
LPARAM))  OnBufferReady //Add this line
	ON_BN_CLICKED(IDC_B_SHUTDOWN, &CdummygenDlg::OnBnClickedBShutdown)
```

```
      ON_BN_CLICKED(IDC_B_BLINK, &CdummygenDlg::OnBnClickedBBlink)
END_MESSAGE_MAP()
…….
……..
```

Collecting Data When a Windows message is received

```
// The data collection function received a message indicating data is ready

long USBDeviceDlg::OnBufferReady( DWORD wParam, long lParam )
{
      float *ptr;
      int ret,i;
      CString str,dum;

      //getting the Data
      ret = fmDLLTransferFloatData((float *&)ptr);

      if(ret == 0 )
      {
            return 0;
      }

      str = "";
      dum = "";
      for(i = 0;i < 16;i++)
      {
            dum.Format("%6.3f, %6.3f, %6.3f, %6.3f, %6.3f, %6.3f, %6.3f,
            %6.3f \r\n", ptr[0], ptr[1], ptr[2], ptr[3], ptr[4], ptr[5],
            ptr[6], ptr[7]);
            ptr += 8;
            str+= dum;
      }

      UpdateData(TRUE);
      m_Data = str; //Data being copied to the display
      UpdateData(FALSE);


      return 0;
}
```

User Thread Messaging being set up to do data collection

Sample 4 - This example shows an MFC User thread function being created for data collection using thread messaging.

```
//Setting up the data collection function in the user thread header (.h)
//Always use WM_USER + 109 for the message identifier


#define WM_GENFIVE_THREAD_BUFFER_READY  (WM_USER + 109)//Add this line
………….
……….
………..
…….
…….
void Cleanup(void);
int SetMessageDestinationWindow( HWND hWnd );
long OnGenDataReadyMsg(DWORD lParem,long rParem); //Add this line
long  OnCommandDispatch(DWORD lParem,long rParem);
……..
…….



//Setting up the data collection function in the user thread main body (.cpp)
message map
…………..
……………
BEGIN_MESSAGE_MAP(CGenThread, CWinThread)
      ON_THREAD_MESSAGE( WM_COMMAND_DISPATCH, (void (AFX_MSG_CALL
CWinThread::*)(WPARAM, LPARAM)) OnCommandDispatch )
    ON_THREAD_MESSAGE( WM_KILL_THREAD, (void (AFX_MSG_CALL
CWinThread::*)(WPARAM, LPARAM)) OnKillThread )
      ON_THREAD_MESSAGE( WM_GENFIVE_THREAD_BUFFER_READY, (void (AFX_MSG_CALL
CWinThread::*)(WPARAM, LPARAM)) OnGenDataReadyMsg )//Add this line
END_MESSAGE_MAP()

…………………

…………
```

Collecting Data When a User Thread message is received

```
//The user thread data collection function

long  CGenThread::OnGenDataReadyMsg(DWORD lParem,long rParem)
{
      float *pSrc;
      float *pSrcA;

      //Get a pointer to the data
      ret = fmDLLTransferFloatData(pSrcA);

      if(ret == 0 )
      {
            return 0;
      }

      //Unload the data

      return( 0 );
}
```

Downloading Some Parameters

```
#include "AMTIUSBDeviceDefinitions.h"

int numdevices
int currentGain[6];
int currentExc[6];
float zeroOffset[6];
int cableLen;


for(i = 0; i<6; i++)
{
	currentGain[i] = 2;  //2000
	currentExc[i]= 0;    //2.5
	zeroOffset[i] = 0.0;
}

numDevices = fmDLLGetDeviceCount();

for(i = 0;i < numDevices;i++)
{
	fmDLLSelectDeviceIndex(i);
	fmSetCurrentExcitations(curentExc);
	fmSetCurrentGains(currentGain);
	fmSetChannelOffsetsTable((float*)zeroOffset);
}

fmSetCableLength(cableLen);

fmResetSoftware();
```

Retrieving Some Parameters

```
#include "AMTIUSBDeviceDefinitions.h"

int i;

char len[30];
char width[30];
char model[30];
char serial[30];
char theDate[30];

float lenWdth[2];
float offsets[3];
float sen[36];
float bridgeResis[6];

memset(len,'\0',30);
memset(width,'\0',30);
memset(model,'\0',30);
memset(serial,'\0',30);
memset(theDate,'\0',30);

offsets[0] = 0.0;
offsets[1] = 0.0;
offsets[2] = 0.0;

for(i = 0;i < 6;i++)
{
      bridgeResis[i] = 0.0;
}

for(i =0;i< 36;i++)
{
      sen[i] = 0.0;
}

fmDLLSelectDeviceIndex(0);
fmGetPlatformModelNumber(model);
fmGetPlatformSerialNumber(serial);
fmGetPlatformDate(theDate);

fmGetPlatformLengthAndWidth((char *)len,(char *) width);

fmGetPlatformXYZOffsets(offsets);

fmGetPlatformBridgeResistance(bridgeResis);
fmGetInvertedSensitivityMatrix(sen);
```

Auto-Ordering the Dataset Platform Order using an MFC timer

```
#include "AMTIUSBDeviceDefinitions.h"


int oldRunMode;

void USBDeviceDlg::StartPlatformOrdering(void)
{
	fmDLLSetDataFormat(0);
	oldRunMode = fmDLLGetRunMode();
	fmBroadcastRunMode(4);
	fmBroadcastPlatformOrderingThreshold(30);
	fmBroadcastResetSoftware();

	Sleep(1000);
	fmBroadcastZero();
	Sleep(500);
	fmDLLStartPlatformOrdering();
	fmBroadcastStart();
	TimerID = SetTimer( TimerIDPltfrmOrder, 500, NULL );

}

void USBDeviceDlg::OnTimer(UINT_PTR nIDEvent)
{

	if(fmDLLIsPlatformOrderingComplete())
	{

		KillTimer(TimerIDPltfrmOrder);

		fmBroadcastRunMode(oldRunMode);
		fmBroadcastResetSoftware();
		Sleep(500);


	}
	else
	{
		SetTimer(TimerIDPltfrmOrder, 250, NULL );
	}
}
```

Appendix A – Integration of the AMTI Optima Signal Conditioner into the USB Device SDK

Introduction

In September 2011 AMTI launched the Optima line of force plate systems introducing an unparalleled level of accuracy for biomechanics force platform measurement. The following section covers additions to the USB Device SDK for integrating the Optima line of signal conditioners.

All third party software modifications to integrate the new Optima signal conditioner will occur in the signal conditioner initialization section of their code. Data collection procedures remain exactly the same for Optima signal conditioners as for other AMTI signal conditioners.

The Optima Binary Calibration File

An AMTI binary calibration file (*.bcf), is shipped with every Optima force platform. The calibration file is too large to store on the platform smart chip and, therefore, has to be written from the PC to the signal conditioner whenever a new Optima signal conditioner / platform combination is detected.

Optima binary calibration files should be installed on the PC through the use of the AMTI System Configuration program. The AMTI System Configuration program is a utility program which ships with all AMTI USB Devices. It is used to configure both the USB Device DLL and all AMTI USB devices. When this program encounters a new Optima signal conditioner / platform combination, it requests the appropriate binary calibration file from the user. It then stores the calibration file on the PC while simultaneously downloading it to the signal conditioner. If later the DLL needs the binary file again it can simply retrieve it from the storage location on the PC.

When storing the initial binary calibration file, the AMTI System Configuration program creates a storage folder and records the folder location in the system registry.

For Windows 7 – 64 the registry location is:

HKEY_CURRENT_CONFIG->SOFTWARE->Wow6432Node ->AMTI->HPS

For Windows XP - 32 the registry location is:

HKEY_CURRENT_CONFIG->SOFTWARE->AMTI->HPS

The current default PC folder location for storing binary calibration files is C:\AMTI\HPS

How the AMTI USB Device DLL Initializes an Optima Signal Conditioner

When an Optima signal conditioner is powered on it queries the connected force platform to determine if the platform is equipped with smart chip technology. If the platform is so equipped, the signal conditioner uploads platform identification information, including the platform serial number. The signal conditioner compares the serial number of the platform to that of the calibration table last stored in local memory. If the serial numbers match the process ends; if there is no match the signal conditioner will require the correct calibration file to be downloaded from the PC.

When the USB Device DLL initializes it detects all connected AMTI USB devices. If an Optima signal conditioner is detected, the SDK will query the signal conditioner to determine if the correct platform calibration table is present. If the correct calibration table is not present the DLL will check the PC registry to obtain the folder location of the Optima calibration files. If the folder location exists, the DLL will search the folder for the correct binary calibration file (*.bcf). If the file is found, the DLL will automatically download it to the Optima. The calibration file download can take up to 15 seconds, meaning the initialization process of the USB Device DLL can take up to 15 seconds.

There are four new SDK functions associated with Optima technology. If these functions are not integrated into third party applications, the software will run fine provided the AMTI System Configuration software was previously used to install the Optima signal conditioners with the correct binary calibration files. If this has not taken place, depending on implementation, the potential 15 second file download time could be problematic.

Gen 5 Compatibility

All Gen 5 functions apply to the Optima signal conditioner except for the following:

fmSetPlatformDate
fmSetPlatformModelNumber
fmSetPlatformSerialNumber
fmSetPlatformLengthAndWidth
fmSetPlatformXYZOffsets
fmGetPlatformXYZOffsets
fmSetPlatformCapacity
fmSetPlatformBridgeResistance
fmGetInvertedSensitivityMatrix
fmSetInvertedSensitivityMatrix

The Optima signal conditioner doesn't have as wide a range of acquisition rates as the Gen 5. The acquisition rates highlighted in orange are common to both the Gen 5 and Optima. Note that the Gen 5 has three additional high speed rates highlighted in blue.

Acquisition Rates									
2000	1800	1500	1200	1000	900	800	600	500	450
400	360	300	250	240	225	200	180	150	125
120	100	90	80	75	60	50	45	40	30
25	20	15	10						

Optima Only Functions

The following list of functions applies to the Optima signal conditioner only:

fmBroadcastCheckOptima
fmOptimaGetStatus
fmOptimaDownloadCalFile
fmIsOptimaDownloadComplete

fmBroadcastCheckOptima

Description

This function checks all connected Optima signal conditioners and reports whether they are ready. It is primarily checking for correct calibration files. If any Optimas are not ready the function returns the number of Optimas not ready to run and their current device indexes.

The function should be called shortly after ***fmDLLIsDeviceInitComplete*** returns success indicating successful SDK initialization.

Format

long fmBroadcastCheckOptima(long *data)

Arguments

The argument is a pointer to a 16-element array of type long, each element containing the device index for an Optima signal conditioner that is not ready. The array will only fill as many elements as indicated in the function return parameter. Call ***fmDLLSelectDeviceIndex*** and ***fmOptimaGetStatus*** to determine why any individual Optima are not ready.

Returns

The number of Optima signal conditioners which are not ready - zero if all Optimas are ready

Related Functions

fmDLLIsDeviceInitComplete
fmDLLSelectDeviceIndex
fmOptimaGetStatus

fmOptimaGetStatus

Description

The function ***fmOptimaGetStatus*** should be called after ***fmBroadcastCheckOptima*** to check the status code of any Optima signal conditioner that is not ready.

Format

long fmOptimaGetStatus(void)

Returns

The status of the currently selected device:

0: The Signal conditioner is a Gen 5
1: The signal conditioner is an Optima and the calibration file is correct
2: Bad CRC check – the calibration file is corrupted
3: The calibration file does not match the platform
4: The Optima signal conditioner is using factory default settings, not a calibration file
5: The Optima signal conditioner is not attached to an Optima platform

Related Functions

fmBroadcastCheckOptima
fmDLLSelectDeviceIndex

fmOptimaDownloadCalFile

Description

This function reads the system registry to determine the folder in which Optima binary calibration files (*.bcf) are stored on the PC. It then searches the folder for the binary calibration file required for the currently selected Optima signal conditioner. If the file is found, the function will begin the process of downloading the calibration file to the signal conditioner. It can take up to 15 seconds to download a binary calibration file from the PC to the signal conditioner. When a binary calibration file is downloaded it's written directly to the flash; it does not need to be saved with any other DLL call.

Format

long fmOptimaDownloadCalFile(BOOL mode)

Arguments

Currently not utilized - always set to 1

Returns

The status of the download initialization:

0: The file download has begun
1: There are no connected devices
2: The connected device is not an Optima signal conditioner
3: No binary calibration file (*.bcf) was found

Related Functions

fmIsOptimaDownloadComplete
fmDLLSelectDeviceIndex

fmIsOptimaDownloadComplete

Description

This function should be called to determine if the binary calibration file (*.bcf) download which began with a call to ***fmOptimaDownloadCalFile*** has completed.

Format

long fmIsOptimaDownloadComplete(void)

Returns

The status of the download operation:

0: File download in process
1: File download is complete
2: File not found on PC
3: File version not supported
4: Bad CRC Check
5: Other

Related Functions

fmDLLSelectDeviceIndex
fmOptimaDownloadCalFile