contactile

3D Force Button Sensor C++ Library for LINUX (Beta v1.0)

Installation and Operation Manual

Document #: 3DFC++LIN_B1.0_MAN_JAN22

January, 2022

Foreword

Information contained in this document is the property of Contactile Pty Ltd. and shall not be reproduced in whole or in part without prior written approval of Contactile Pty Ltd. The information herein is subject to change without notice and should not be construed as a commitment on Contactile Pty Ltd. This manual is periodically revised to reflect and incorporate changes made to the 3D Force Button Sensor Development Kit.

Contactile Pty Ltd assumes no responsibility for any errors or omissions in this document. Users' critical evaluation is welcome to assist in the preparation of future documentation.

Conditions of Sale

Contactile's conditions of sale apply to all products sold by Contactile to the Distributor under this Agreement. The conditions of sale that apply are provided on the USB flash drive shipped with the product in the folder ‘LEGAL’ in the root directory.

End User Licence Agreement

Contactile's end user license agreement applies to all software and algorithms included with the products sold by Contactile. The end user license agreement that applies is provided on the USB flash drive shipped with the product in the folder ‘LEGAL’ in the root directory.

Compliance

The devices are sold as is.

The devices are specifically designed solely for the purposes of research and development only made available on a business-to-business basis.

The devices are not for resale.

Table of Contents

1 Introduction

The 3D Force Button Sensor Development Kit (Beta v1.0) is a system of (up to) five 3D Force Button Sensors per adapter, (up to) two Adaptors, and a Controller. Each 3D Force Button Sensor can measure 3D force. The Controller supplies power for (up to) two Adaptors and coordinates the simultaneous data acquisition from up to ten 3D Force Button Sensors. The Development Kit is shipped with visualisation software and (optional) C++ libraries for Windows and Linux environments and a ROS node for developing software control algorithms using the sensor signals.

The main components of the 3D Force Button Sensor Development Kit (Beta v1.0) are shown in Figure 1.1, connected to a laptop running the visualisation software.

Figure 1.1 – The 3D Force Button Sensor Development Kit (Beta v1.0). Laptop not included.

This document is an installation and operation manual for the C++ Library for LINUX which was provided on the Contactile USB flash drive that was shipped with the 3D Force Button Sensor Development Kit (Beta v1.0).

2 Safety

2.1 General

The customer should verify that the maximum loads and moments expected during operation fall within the sensing range of the sensor as outside this range, sensor reading accuracy is not guaranteed (refer to Document #3DFDK_B1.0_MAN_JAN22). Particular attention should be paid to dynamic loads caused by robot acceleration and deceleration if the sensors are mounted on robotic equipment. These forces can be many multiples of the value of static forces in high acceleration or deceleration situations.

2.2 Explanation of warnings

The warnings included here are specific to the product(s) covered by this manual. It is expected that the user heed all warnings from the manufacturers of other components used in the installation.

Danger indicates that a situation could result in potentially serious injury or damage to equipment.

Caution indicates that a situation could result in damage to the product and/or the other system components.

2.3 Precautions

DANGER: Do not attempt to disassemble the sensor. This could damage the sensor and will invalidate the calibration.

DANGER: Do not attempt to drill, tap, machine, or otherwise modify the sensor casing. This could damage the sensor and will void invalidated the calibration.

DANGER: Do not use the sensor on abrasive surfaces or surfaces with sharp points/edges. This could damage the silicone surface of the sensor.

CAUTION: Sensors may exhibit a small offset in readings when exposed to intense light sources.

CAUTION: Exceptionally strong and changing electromagnetic fields, such as those produced by magnetic resonance imaging (MRI) machines, constitute a possible source of interference with the operation of the sensor and Controller.

CAUTION: Temperature variations can cause drift in sensor readings. Some temperature compensation is performed. However, bias removal in software prior to operation is necessary, and it is recommended that biasing is performed each time the sensor is known to be unloaded.

3 Getting started

This section contains instructions for setting up and using 3D Force Button Sensor C++ Library for LINUX (Beta v1.0). It is recommended that first time users first read the preceding Safety section, then read through this section to get more familiar with the system.

3.1 Hardware installation

The C++ Library is used with the 3D Force Button Sensor Development Kit (Beta v1.0). The Controller should be connected to the 3D Force Button Sensors, then the Controller should be connected via the micro USB port on the Controller to a PC running LINUX before you can use the C++ Library. For more information about connecting the sensors and powering on the Controller, refer to Document #3DFDK_B1.0_MAN_JAN22.

3.2 Software installation

The C++ Library is provided on the Contactile USB flash drive that was shipped with the development kit in a folder named SOFTWARE/*C++LIN*. To install the library, simply copy the entire contents of the *C++LIN* folder to a location on a PC running LINUX.

The files in the *C++LIN* folder are summarised in Table 3.1.

Table 3.1 – Files in *C++LIN* folder

Sub Folder	**File Name**	**File Description**
Include	PTSDKConstants.h	The header file containing constant definitions
	PTSDKListener.h	The header file for the PTSDKListener class
	PTSDKSensor.h	The header file for the PTSDKSensor class
Library	libPTSDK.a	The library file
Example	USER_EXAMPLE.cpp	The example C++ code for using the C++ Library.
	Makefile	The makefile for compiling the example code, linking the library and building an executable.

.

4 Class and function documentation

In this section, the classes and class functions of the C++ Library are described.

4.1 Constants

The PTSDKConstants.h file contains definitions of constants that are used across a number of the library classes. The constants are described in Table 4.1.

Table 4.1 – Constants defined by #define pre-processor directives in PTSDKConstants.h

Name	Value	Description
IN	-	Used in a function declaration to indicate an *input parameter*
OUT	-	Used in a function declaration to indicate an *output parameter*
STARTBYTE0	0x55	The first byte of the start packet
STARTBYTE1	0x66	The second byte of the start packet
STARTBYTE2	0x77	The third byte of the start packet
STARTBYTE3	0x88	The fourth byte of the start packet
ENDBYTE0	0xAA	The first byte of the end packet
ENDBYTE1	0xBB	The second byte of the end packet
ENDBYTE2	0xCC	The third byte of the end packet
ENDBYTE3	0xDD	The fourth byte of the end packet
X_IND	0	The index of the X-dimension
Y_IND	1	The index of the Y-dimension
Z_IND	2	The index of the Z-dimension
NDIM	3	The number of spatial dimensions
MAX_NSENSOR	4	The maximum number of sensors connected to the Controller
LOG_RATE_100	100	Constant representing 100 Hz sampling rate
LOG_RATE_500	500	Constant representing 500 Hz sampling rate
LOG_RATE_1000	1000	Constant representing 1000 Hz sampling rate

4.2 Class list

The classes in the C++ Library and a brief description are listed in Table 4.2.

Table 4.2 – Classes in the C++ Library

Class	Description
PTSDKListener	Describes a listener for the Controller with a number of 3D Force Button Sensors connected
PTSDKSensor	Describes a 3D Force Button Sensor

4.3 Function list

The functions in each class are described in the following subsections. A function called myFunction with N input parameters (with names param1 to paramN), M output parameters (with names paramN+1 to paramN+M) and a return value is described in the following way:

typeR myFunction(IN type1 *param1*, …, IN typeN *paramN*,
OUT typeN+1 *paramN+1*, …, OUT typeN+M *paramN+M*)

Description: A description of the function myFunction

Parameters:

[in]	param1	A description of the input parameter "*param1*" of type "type1".
	⋮	
[in]	paramN	A description of the input parameter "*paramN*" of type "typeN".
[out]	paramN+1	A description of the output parameter "*paramN+1*" of type "typeN+1".
	⋮	
[out]	paramN+M	A description of the output parameter called "*paramN+M*" of type "typeN+M".

Returns: A description of the return value of type "typeR".

4.3.1 PTSDKListener class public functions

The PTSDKListener is the class which interacts with the Controller that is in turn hosting up to ten connected 3D Force Button Sensors. This class describes an object that connects with the Controller via a serial connection emulated on the computer's USB port, and reads and processes the data streaming through the serial connection. This class also logs the data to a log file – See Section 6 Log file. The public member functions of the PTSDKListener class are described below.

PTSDKListener(IN const bool *isLog*)

Description: Constructor.
Parameters: [in] isLog A flag indicating whether to log data to CSV file.

~PTSDKListener()

Description: Destructor.

void addSensor(IN PTSDKSensor * *pSensor*)

Description: Adds a sensor object to the PTSDKListener.
Parameters: [in] pSensor A pointer to the sensor object.

int connect (IN const char **port*,
IN const int *rate*,
IN const int *parity*,
IN const char *byteSize*)

Description: Connects to the COM port.
Used in conjunction with the readNextSample and disconnect functions.
Parameters: [in] port The COM port name.
[in] rate The rate of the connection.
[in] parity The parity of the connection.
[in] byteSize The byte size for the connection.
Returns: 0 if successfully connected, error code if unsuccessful.

int connectAndStartListening(IN const char **port*,
IN const int *rate*,
IN const int *parity*,
IN const char *byteSize,*
IN const int *logFileRate*)

Description: Connects to the COM port and starts listening for data (starts the listening thread), processes the data and logs the data to a log file.
Used in conjunction with the stopListeningAndDisconnect function.
Parameters: [in] Port The COM port name.
[in] Rate The rate of the connection.
[in] Parity The parity of the connection.
[in] byteSize The byte size for the connection.
[in] logFileRate The log file rate. LOG_RATE_100, LOG_RATE_500 Hz, or LOG_RATE_1000 for 100, 500 or 1000 Hz, respectively.
Returns: 0 if successfully connected, error code if unsuccessful.

void disconnect(void)

Description: Disconnects from the COM port.
Used in conjunction with the connect and readNextSample functions.

bool readNextSample(void)

Description: Reads and parses the next sample from the COM port, and stores the sample in the associated PTSDKSensor objects.
Used in conjunction with the connect and disconnect functions.

Returns: True if successfully read a sample, false if unsuccessful.

void run(void)

Description: The 'infinite' loop of the listening thread.
The thread implementation necessitates that this is a public member function. However, this function should not be called except through the connectAndStartListening function when the listening thread is spawned.

bool sendBiasRequest(void)

Description: Sends a bias request to the Controller. A bias should be performed after connecting to the serial port and starting to stream data with the sensor unloaded. A bias should be performed each time the sensor is known to be unloaded. A bias operation takes approximately 2 s. Ensure that the sensor remains unloaded throughout this time.

Returns: True if successfully sent the request, false if unsuccessful.

void stopListeningAndDisconnect(void)

Description: Stops listening for data from the COM port (and kills the listening thread), stops logging data to the log file and disconnects from the COM port.

4.3.2 PTSDKSensor class public functions

The PTSDKSensor is a class that describes a 3D Force Button Sensor (v2.0). This is the main class for accessing the current sensor measurements in a user-defined program.

PTSDKSensor(void)

Description: Constructor - Initialises pillars

~PTSDKSensor(void)

Description: Destructor.

void getGlobalForce(OUT double *result*[NDIMENSION])

Description: Gets the global X,Y,Z force acting on the sensor.

Parameters: [out] result The global X, Y and Z force.

uint32_t getTimestamp_us(void)

Description: Gets the timestamp of the current sample of a pillar in µs.

Returns The timestamp of the current sample of a pillar in us.

5 Writing a user application using the C++ Library

This section contains code snippets to explain each step required to write a user application that uses the C++ Library to monitor ten 3D Force Button Sensors. The full example can be found in the USER_APP_EXAMPLE.cpp file in the *Example* subfolder of the *C++ Library* folder.

5.1 Include files

The examples for a user defined application in the following sections require the include files listed in Example 5.1.

Example 5.1 – Include files for the example user application

```
#include <stdio.h>

#ifndef PTSDKCONSTANTS_H
#include "PTSDKConstants.h"
#endif

#ifndef PTSDKLISTENER_H
#include "PTSDKListener.h"
#endif

#ifndef PTSDKSENSOR_H
#include "PTSDKSensor.h"
#endif
```

5.2 Initialising PTSDKSensor and PTSDKListener objects

To initialise a PTSDKListener object, first, the PTSDKSensor objects must be initialised. The following information is required to initialise the PTSDKSensor objects. Ten sensors should be initialised irrespective of how many physical sensors are connected. An example of initialising two PTSDKSensor objects then initialising the PTSDKListener object is shown in Example 5.2.

Example 5.2 – Initialising two PTSDKSensor objects and a PTSDKListener object

```
/* Initialise 10x PTSDKSensor objects irrespective of number of physical sensors */
PTSDKSensor sen0 = PTSDKSensor();
PTSDKSensor sen1 = PTSDKSensor();
PTSDKSensor sen2 = PTSDKSensor();
PTSDKSensor sen3 = PTSDKSensor();
PTSDKSensor sen4 = PTSDKSensor();
PTSDKSensor sen5 = PTSDKSensor();
PTSDKSensor sen6 = PTSDKSensor();
PTSDKSensor sen7 = PTSDKSensor();
PTSDKSensor sen8 = PTSDKSensor();
PTSDKSensor sen9 = PTSDKSensor();

/* Initialise the PTSDKListener object irrespective of number of physical sensors */
bool isLogging = true; // Create a log file
PTSDKListener listener = PTSDKListener(isLogging);

/* Add 10x sensors to the listener */
listener.addSensor(&sen0); // SENO - A
listener.addSensor(&sen1); // SENO - B
listener.addSensor(&sen2); // SENO - C
listener.addSensor(&sen3); // SENO - D
listener.addSensor(&sen4); // SENO - E
listener.addSensor(&sen5); // SEN1 - A
listener.addSensor(&sen6); // SEN1 - B
listener.addSensor(&sen7); // SEN1 - C
listener.addSensor(&sen8); // SEN1 - D
listener.addSensor(&sen9); // SEN1 - E
```

5.3 Connecting to the COM port and listening for data

After initialising the PTSDKListener, a serial connection must be established. To connect to the COM port, the name of the COM port assigned to the connected Controller must be known. Once the PTSDKListener has established a connection with the COM port of the Controller, the Controller will begin transmitting data through the serial connection.

There are two methods by which a user defined program can retrieve data from the Controller:

1. Single thread
2. Multi-threaded

Note: There should be a COM port associated with the Controller (to power the Controller, the micro-USB should be connected between the micro-USB port on the Controller and the PC).

When data is no longer required, the PTSDKListener object should stop listening for data, disconnect from the COM port and flush and close the log file.

5.3.1 COM port configuration parameters

The COM port configuration parameters are first required. An example of initialising the COM port configuration parameters is shown in Example 5.3.

Example 5.3 – Connecting the PTSDKListener object to the COM port and listen for data in a single thread

```
/* Initialise connection parameters */
char port[] = "/dev/ttyACM0";     // The name of the COM port to connect with
int rate = 9600;                  // The rate of the serial connection
int parity = 0;                   // 0=PARITY_NONE, 1=PARITY_ODD, 2=PARITY_EVEN
char byteSize = 8;          // The number of bits in a byte
```

5.3.2 Single thread

The structure of a user defined application using a single thread to retrieve sensor data from the Controller is shown in Example 5.4.

Example 5.4 – Connecting to the COM port and listening for data in a single thread

```
/* Connect to the serial port */
if(listener.connect(port, rate, parity, byteSize) == 0){
	printf("main(): Successfully connected to %s.\n",port);
}else{
	printf("main(): FAILED to connect to %s\n.",port);
	return -1;
}

while(true){
	/* Read the next sample from the Controller */
	if(listener.readNextSample()){
		printf("main(): Successfully read the next sample.\n");
	}else{
		printf("main(): FAILED to read the next sample.\n");
		break;
	}

	/* Retrieve data from PTSDKSensor objects and do something with it */
	// User specific code goes here - See Example 5.7
}

/* Disconnect from the COM port */
listener.disconnect();
```

5.3.3 Multi-threaded

The PTSDKListener object can launch a thread which listens for and processes the incoming data packets. An example of how to connect to the COM port and start listening for data using a new thread is shown in Example 5.5.

Example 5.5 – Connecting to the COM port and listening for data in a multi-threaded application

```
/* Connect to the serial port and start listening for and processing data */
if(listener.connectAndStartListening(port, rate, parity, byteSize, LOG_RATE_1000) == 0){
	printf("main(): Successfully connected to %s & started listening\n",port);
}else{
	printf("main(): FAILED to connect to %s, didn't start listening\n",port);
	return -1;
}

while(true){
	/* Retrieve data from PTSDKSensor objects and o something with it */
	// User specific code goes here - See Example 5.7, Error! Reference source not f
ound., and Error! Reference source not found.
}

/* Stop listening for and processing data and disconnect from the COM port */
listener.stopListeningAndDisconnect();
```

5.4 Biasing the sensors

Biasing refers to removing any offset in the pillar readings when the pillars are unloaded. It is recommended that the user performs a bias each time the sensors are known to be unloaded. Ensure that the sensor has been unloaded for at least one second before performing a bias to ensure that the bias calculation does not include hysteresis effects. A bias operation can take up to 2 s. Ensure that the sensor remains unloaded throughout this time. An example of how to perform a bias is shown in Example 5.6.

Example 5.6 – Biasing all pillars on all sensors

```
/* Perform bias */
if(listener.sendBiasRequest()){
	printf("main(): Successfully sent bias request.\n");
}else{
	printf("main(): FAILED to send bias request.\n");
	return -1;
}
```

5.5 Accessing sensor data

Once the PTSDKListener object is listening for and processing data and the sensors have been biased, the user application can access the incoming sensor data. An example of how to access data from a sensor is shown in Example 5.7.

Example 5.7 – Accessing data from a sensor

```
/* Get the XYZ global force on sensor 1 */
double globalForce[NDIM];
sen1.getGlobalForce(globalForce);
for(int dInd = 0; dInd < NDIM; dInd++){
	printf("S1: global F%d = %.3f\n", dInd, globalForce[dInd]);
}
printf("\n");
```

6 Log file

6.1 Overview

If the PTSDKListener object was initialised with the isLogging flag being true, the function connectAndStartListening (in a multi-threaded application) and the PTSDK function readNextSample (in a single thread application) also generate a log file of the sensor data.

6.2 Log file location

The log file that is generated is stored in the Logs subfolder in the same location as the user-defined application which uses the C++ Library.

6.3 Log file name

The name of the log file that is generated is LOG_YYYY_MM_DD_hh_mm_ss.csv where:

- YYYY is the four digit year,
- MM is the two digit month,
- DD is the two digit day,
- hh is the two digit hour,
- mm is the two digit minute and
- ss is the two digit second,

from the system clock at the time that the log file was created.

6.4 Log file format

The log file is saved as comma-separated values (CSV) in ASCII text format. The order of the values and a description of the log file is described in Document #3DFDK_B1.0_MAN_JAN22.