contactile

PapillArray Tactile Sensor C++ Library for WINDOWS (v2.0)

Installation and Operation Manual

Document #: PTSC++WIN_2.0_MAN_NOV21

November, 2021

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 PapillArray Tactile 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 PapillArray Tactile Sensor Development Kit (v2.0) is a system of (up to) two PapillArray Tactile Sensor arrays and a Controller. Each PapillArray Tactile Sensor array can measure 3D displacement, 3D force, and vibration on each sensing element, as well as global 3D force, global 3D torque, the onset of slip, and friction. The Controller supplies power for (up to) two sensors and coordinates the simultaneous data acquisition from up to two PapillArray Tactile Sensors; i.e., coordinates sampling of the 9 pillars if one sensor is connected to the Controller, 18 pillars if two sensors are connected to the Controller. 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 PapillArray Tactile Sensor Development Kit (v2.0) are shown in Figure 1.1, connected to a laptop running the visualisation software.

Figure 1.1 – The PapillArray Tactile Sensor Development Kit (v2.0). Laptop not included.

This document is an installation and operation manual for the C++ Library for WINDOWS which was provided on the Contactile USB flash drive that was shipped with the Development Kit.

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 #PTS_2.0_SPEC_NOV21). 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 PapillArray Tactile Sensor C++ Library for WINDOWS (v2.0). 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 PapillArray Tactile Sensor Development Kit (v2.0). The Controller should be connected to the PapillArray Tactile Sensors, then the Controller should be connected via the micro USB port on the Controller to a PC running WINDOWS before you can use the C++ Library. For more information about connecting the sensors and powering on the Controller, refer to Document #PTSDK_2.0_MAN_NOV21.

3.2 End user licence agreement licence

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.

3.3 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++WIN*. To install the library, simply copy the entire contents of the *C++WIN* folder to a location on a PC running Windows.

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

Table 3.1 – Files in *C++WIN* folder

Sub Folder	File Name	File Description
Include	PTSDK_CPP_LIB.h	The header file containing DLL definitions
	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
	PTSDKPillar.h	The header file for the PTSDKPillar class
Lib	PTSDK_CPP_LIB.lib	For statically linking the library
DLL	PTSDK_CPP_LIB.dll	For dynamically linking the library during runtime
Example	USER_APP_EXAMPLE.vcproj and associated files	A Visual Studio 2019 project containing an example user program
	USER_APP_EXAMPLE.cpp	The example C++ code
	USER_APP_EXAMPLE.h	Header file for USER_APP_EXAMPLE.cpp

3.4 Library linking

The library must be linked in your software project. For detailed information about linking the library to your software project, consult the relevant documentation of the development environment being used.

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
MAX_NPILLAR	100	The maximum number of pillars in a sensor
CONTACT_THRESH	0.5	The minimum normal (Z) force for a pillar to be in contact
INELIGIBLE	-2	Pillar slip state: not in contact at slip detection start
CONTACT_AT_START	1	Pillar slip state: in contact from slip detection start
LOST_CONTACT	-1	Pillar slip state: has lost contact
TLOADING	2	Pillar slip state: is loaded tangentially
SLIPPED	3	Pillar slip state: has slipped
NOFRICTIONEST	-1	Value representing no friction estimate
SAMP_RATE_100	100	Constant representing 100 Hz sampling rate
SAMP_RATE_250	250	Constant representing 250 Hz sampling rate
SAMP_RATE_500	500	Constant representing 500 Hz sampling rate
SAMP_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 PapillArray Tactile Sensors connected
PTSDKSensor	Describes a PapillArray Tactile Sensor comprised of multiple pillars

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 two connected PapillArray Tactile 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 bool *isFlush*)

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] isFlush A flag indicating whether to flush the hardware input buffer if it contains too many bytes.

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.

bool setSamplingRate(IN const int *samplingRate*)

Description: Sets the sampling rate on the Controller.

Parameters: [in] samplingRate The sampling rate for the Controller: SAMP_RATE_100, SAMP_RATE_250, SAMP_RATE_500 or SAMP_RATE_1000

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

bool startSlipDetection(void)

Description: Starts the slip detection algorithms on the Controller. This should be called after a number of pillars of the sensor are already in contact, before tangential loading of the sensor.

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.

bool stopSlipDetection(void)

Description: Stops and resets the slip detection algorithms on the Controller.

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

4.3.2 PTSDKSensor class public functions

The PTSDKSensor is a class that describes a PapillArray Tactile 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 getAllDisplacements(OUT double *result*[NDIMENSION][MAX_NPILLAR])
Description: Gets the current X, Y and Z displacements of all pillars of this sensor.
Parameters: [out] Result The X, Y and Z displacements of all pillars.

void getAllForces(OUT double *result*[NDIMENSION][MAX_NPILLAR])
Description: Gets the current X, Y and Z forces of all pillars of this sensor.
Parameters: [out] Result The X, Y and Z forces of all pillars.

void getAllSlipStatus(OUT bool **isSlipDetectionActive*,
OUT bool **isRefPillarLoaded*,
OUT bool *contactStates*[MAX_NPILLAR],
OUT int *slipStates*[MAX_NPILLAR])

Description: Gets the slip state of all pillars of the sensor.

Parameters:			
	[out]	isSlipDetectionActive	True if slip detection is active, false otherwise.
	[out]	isRefPillarLoaded	True if the tangential force on the reference pillar has exceeded the threshold, false otherwise
	[out]	contactStates	For each pillar, true if the pillar normal force exceeds the threshold for contact, false otherwise.
	[out]	slipStates	The slip states of all pillars: INELIGIBLE if the pillar was not in contact when slip detection started, CONTACT_AT_START if the pillar was in contact when slip detection started, LOST_CONTACT if the pillar lost contact after slip detection started, TLOADING if the pillar is being loaded tangentially, of SLIPPED if the pillar has slipped.

double getFrictionEstimate(void)
Description: Gets the current friction estimate from this sensor.
Returns: The current friction estimate from this sensor; or NOFRICTIONESTIMATE if there is no friction estimate.

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.

void getGlobalTorque(OUT double *result*[NDIMENSION])
Description: Gets the global X,Y,Z torque acting on the sensor.
Parameters: [out] result The global X, Y and Z torque.

int getNPillar(void)
Description: Gets the number of pillars in this sensor.
Returns: The number of pillars in this sensor.

bool getPillarDisplacements(IN const int *pillarInd*, OUT double *result*[NDIMENSION])
Description: Gets the current X, Y and Z displacement of a pillar.
Parameters: [in] pillarInd The index of the pillar.
[out] result The X, Y and Z displacement of the pillar.
Returns: True if the pillar is valid; false otherwise.

bool getPillarForces(IN const int *pillarInd,* OUT double *result*[NDIMENSION])
Description: Gets the current X, Y and Z force on the pillar tip.
Parameters: [out] Result The current X, Y and Z force on the pillar tip.
Returns: True if the pillar index is valid; false otherwise.

bool getPillarForceAbs(IN const int *pillarInd,* OUT double **result*)
Description: Gets the current absolute X, Y, Z force on the pillar tip.
Parameters: [out] Result The current absolute X, Y, Z force on the pillar tip.
Returns: True if the pillar index is valid; false otherwise.

bool getPillarForceN(IN const int *pillarInd,* OUT double **result*)
Description: Gets the current normal (Z) force on the pillar tip.
Parameters: [out] Result The current normal (Z) force on the pillar tip.
Returns: True if the pillar index is valid; false otherwise.

bool getPillarForceT(IN const int *pillarInd,* OUT double **result*)
Description: Gets the current tangential (XY) force.
Parameters: [out] Result The current X, Y and Z force on the pillar tip.
Returns: True if the pillar index is valid; false otherwise.

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.

bool isSensorInContact(void)
Description: Gets whether the sensor is in contact.
Returns: True if at least one pillar is in contact; false otherwise.

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 two PapillArray Tactile 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 "stdafx.h"
#include <stdio.h>
#include <tchar.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. 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 a PTSDKSensor object for sensor connected to SEN0 port */
PTSDKSensor sen0 = PTSDKSensor();

/* Initialise a PTSDKSensor object for SEN1 sensor */
PTSDKSensor sen1 = PTSDKSensor();

/* Initialise the PTSDKListener object */
bool isLogging = true; // Create a log file
PTSDKListener listener = PTSDKListener(isLogging);

/* Add sensor 0 to the listener */
listener.addSensor(&sen0);

/* Add sensor 1 to the listener */
listener.addSensor(&sen1);
```

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[] = "\\\\.\\COM1";      // The name of the COM port to connect with
                                  // The additional '\' characters are required
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;
}
```

```
bool isFlush = true;       // Flag indicating whether to flush the
                           // hardware input buffer if it contains too many bytes
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 5.8, Example 5.9, and Example 5.10
}

/* 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 */
bool isFlush = false;      // Flag indicating whether to flush the
                           // hardware input buffer if it contains too many bytes
if(listener.connectAndStartListening(port, rate, parity, byteSize, isFlush) == 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 5.8, Example 5.9, and Example 5.10
}

/* 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 Setting the Controller sampling rate

By default, upon powering up, the Controller will default to a sample rate of 1000 Hz. The Controller sampling rate can be changed to 100, 250, 500 or 1000 Hz. An example of how to change the Controller sampling rate to 500 Hz is shown in Example 5.7.

Example 5.7 – Setting the Controller sampling rate

```
/* Set sampling rate */
if(listener.setSamplingRate(SAMP_RATE_500)){
	printf("main(): Successfully set the sampling rate to 500 Hz.\n");
}else{
	printf("main(): FAILED to set the sampling rate to 500 Hz.\n");
	return -1;
}
```

5.6 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. Examples of how to access different types of data are shown in Example 5.8, Example 5.9 and Example 5.10.

Example 5.8 – Accessing data from a whole 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");

/* Get XYZ displacements of all pillars of sensor 1 */
double allDisplacements[NDIM][MAX_NPILLAR];
sen1.getAllDisplacements(allDisplacements);
for(int pInd = 0; pInd < sen1.getNPillar(); pInd++){
	printf("S1_P%d", pInd);
	for(int dInd = 0; dInd < NDIM; dInd++){
		printf("\tD%d = %.3f\n", dInd, allDisplacements[dInd][pInd]);
	}
	printf("\n");
}
printf("\n");
```

Example 5.9 – Accessing data from a single pillar

```
/* Retreive and print the XYZ displacement of pillar 3 on sensor 0 */
int pInd = 3;
double displacement[NDIM];
sen0.getPillarDisplacements(pInd, displacement);
for(int dInd = 0; dInd < NDIM; dInd++) {
	printf("S0_P%d_D%d: %.3f\n", pInd, dInd, displacement[dInd]);
}

/* Retreive and print the XYZ force of pillar 5 on sensor 0 */
pInd = 5;
double force[NDIM];
sen0.getPillarForces(pInd,force);
for(int dInd = 0; dInd < NDIM; dInd++) {
	printf("S0_P%d_F%d: %.3f\n", pInd, dInd, force[dInd]);
}
```

Example 5.10 – Performing slip detection and estimating friction

```
/* Start slip detection
 * Only do this after a few pillars of each sensor are in contact
 * and before a tangential load is applied.
 * Not intended for slip detection in the presence of torsional loads.
 */
if(!listener.startSlipDetection()){
	printf("FAILED to start slip detection.\n");
	return -1;
}else{
	printf("Successfully started slip detection.\n");
}

/* Retrieve slip status and friction estimate
 * This would be implemented in a loop
 * Loop structure is dependent on single thread or multi-threaded application
 */

// Get slip states of all pillars in sensor 0
bool isSlipDetectionActive, isRefPillarLoaded, contactStates[MAX_NPILLAR];
int slipStates[MAX_NPILLAR];
sen0.getAllSlipStatus(&isSlipDetectionActive, &isRefPillarLoaded,
                          contactStates, slipStates);
for(int pInd = 0; pInd < sen0.getNPillar(); pInd++){
	printf("S0_P%d: ",pInd);
	switch(slipStates[pInd]){
		case INELIGIBLE:
			printf("was not in contact at slip detection start.\n");
			break;
		case CONTACT_AT_START:
			printf("in contact from slip detection start.\n");
			break;
		case LOST_CONTACT:
			printf("lost contact.\n");
			break;
		case TLOADING:
			printf("is being tangentially loaded\n");
			break;
		case SLIPPED:
			printf("slipped.\n");
	}
}
// Get the current friction estimate of sensor 0
double friction = sen0.getFrictionEstimate();
printf("S0: friction estimate = %.3f.\n\n",friction);

/* Stop slip detection */
if(!listener.stopSlipDetection()){
	printf("FAILED to stop slip detection.\n");
	return -1;
}else{
	printf("Successfully stopped slip detection.\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 is shown in Table 6.1

Table 6.1 – Data in log file

Data Order	Data Name	Data Description	
1	T_us	Timestamp in µs	Timestamps
2	S0_P0_DX	Sensor 0, pillar 0, X-axis displacement	Sensor 0, pillar 0, displacements
3	S0_P0_DY	Sensor 0, pillar 0, Y-axis displacement	
4	S0_P0_DZ	Sensor 0, pillar 0, Z-axis displacement	
5	S0_P0_FX	Sensor 0, pillar 0, X-axis force	Sensor 0, pillar 0, forces
6	S0_P0_FY	Sensor 0, pillar 0, Y-axis force	
7	S0_P0_FZ	Sensor 0, pillar 0, Z-axis force	
8	S0_P1_DX	Sensor 0, pillar 1, X-axis displacement	Sensor 0, pillar 1, displacements
9	S0_P1_DY	Sensor 0, pillar 1, Y-axis displacement	
10	S0_P1_DZ	Sensor 0, pillar 1, Z-axis displacement	
11	S0_P1_FX	Sensor 0, pillar 1, X-axis force	Sensor 0, pillar 1, forces
12	S0_P1_FY	Sensor 0, pillar 1, Y-axis force	
13	S0_P1_FZ	Sensor 0, pillar 1, Z-axis force	
⋮			

Data Order	Data Name	Data Description	
50	S0_P8_DX	Sensor 0, pillar 8, X-axis displacement	Sensor 0, pillar 8, displacements
51	S0_P8_DY	Sensor 0, pillar 8, Y-axis displacement	
52	S0_P8_DZ	Sensor 0, pillar 8, Z-axis displacement	
53	S0_P8_FX	Sensor 0, pillar 8, X-axis force	Sensor 0, pillar 8, forces
54	S0_P8_FY	Sensor 0, pillar 8, Y-axis force	
55	S0_P8_FZ	Sensor 0, pillar 8, Z-axis force	
56	S0_GG_FX	Sensor 0, global X-axis force	Sensor 0, global force
57	S0_GG_FY	Sensor 0, global Y-axis force	
58	S0_GG_FZ	Sensor 0, global Z-axis force	
59	S0_GG_TX	Sensor 0, global X-axis torque	Sensor 0, global torque
60	S0_GG_TY	Sensor 0, global Y-axis torque	
61	S0_GG_TZ	Sensor 0, global Z-axis torque	
62	S0_isSDActive	Sensor 0, is slip detection activated	Sensor 0 slip detection and friction estimate
63	S0_isRefLoad	Sensor 0, is the reference pillar tangentially loaded	
64	S0_P0_isInContact	Sensor 0, pillar 0, is in contact	
65	S0_P0_slipState	Sensor 0, pillar 0, slip state	
66	S0_P1_isInContact	Sensor 0, pillar 1, is in contact	
67	S0_P1_slipState	Sensor 0, pillar 1, slip state	
⋮			
80	S0_P8_isInContact	Sensor 0, pillar 8, is in contact	
81	S0_P8_slipState	Sensor 0, pillar 8, slip state	
82	S0_FrictionEst	Sensor 0, friction estimate	
83	S1_P0_DX	Sensor 1, pillar 0, X-axis displacement	Sensor 1, pillar 0, displacements
84	S1_P0_DY	Sensor 1, pillar 0, Y-axis displacement	
85	S1_P0_DZ	Sensor 1, pillar 0, Z-axis displacement	
86	S1_P0_FX	Sensor 1, pillar 0, X-axis force	Sensor 1, pillar 0, forces
87	S1_P0_FY	Sensor 1, pillar 0, Y-axis force	
88	S1_P0_FZ	Sensor 1, pillar 0, Z-axis force	
89	S1_P1_DX	Sensor 1, pillar 1, X-axis displacement	Sensor 1, pillar 1, displacements
90	S1_P1_DY	Sensor 1, pillar 1, Y-axis displacement	
91	S1_P1_DZ	Sensor 1, pillar 1, Z-axis displacement	
92	S1_P1_FX	Sensor 1, pillar 1, X-axis force	Sensor 1, pillar 1, forces
93	S1_P1_FY	Sensor 1, pillar 1, Y-axis force	
94	S1_P1_FZ	Sensor 1, pillar 1, Z-axis force	
…			
131	S1_P8_DX	Sensor 1, pillar 8, X-axis displacement	Sensor 1, pillar 8, displacements
132	S1_P8_DY	Sensor 1, pillar 8, Y-axis displacement	
133	S1_P8_DZ	Sensor 1, pillar 8, Z-axis displacement	

Data Order	Data Name	Data Description	
134	S1_P8_FX	Sensor 1, pillar 8, X-axis force	Sensor 1, pillar 8, forces
135	S1_P8_FY	Sensor 1, pillar 8, Y-axis force	
136	S1_P8_FZ	Sensor 1, pillar 8, Z-axis force	
137	S1_GG_FX	Sensor 1, global X-axis force	Sensor 1, global force
138	S1_GG_FY	Sensor 1, global y-axis force	
139	S1_GG_FZ	Sensor 1, global Z-axis force	
140	S1_GG_TX	Sensor 1, global X-axis torque	Sensor 1, global torque
141	S1_GG_TY	Sensor 1, global Y-axis torque	
142	S1_GG_TZ	Sensor 1, global Z-axis torque	
143	S0_isSDActive	Sensor 1, is slip detection activated	Sensor 1 slip detection and friction estimate
144	S0_isRefLoad	Sensor 1, is the reference pillar tangentially loaded	
145	S0_P0_isInContact	Sensor 1, pillar 0, is in contact	
146	S0_P0_slipState	Sensor 1, pillar 0, slip state	
147	S0_P1_isInContact	Sensor 1, pillar 1, is in contact	
148	S0_P1_slipState	Sensor 1, pillar 1, slip state	
		⋮	
161	S0_P8_isInContact	Sensor 1, pillar 8, is in contact	
162	S0_P8_slipState	Sensor 1, pillar 8, slip state	
163	S0_FrictionEst	Sensor 1, friction estimate	