

Tracker Interface 5.1.10

User's Guide

Table of Contents

| | | |
|------------|---|----------|
| 1 | INTRODUCTION | 1 |
| 1.1 | Changes | 1 |
| 1.1.1 | Changes since 1.10.1 | 1 |
| 1.1.2 | Changes since 1.5.0 | 2 |
| 1.1.3 | Changes since 1.6.0 | 2 |
| 1.1.4 | Changes since 1.7.0 | 2 |
| 1.1.5 | Changes since 1.7.2 | 2 |
| 1.1.6 | Changes since 1.7.3 | 2 |
| 1.1.7 | Changes since 1.8.0 | 2 |
| 1.1.8 | Changes since 1.9.0 | 2 |
| 1.1.9 | Changes since 1.10.0.3 | 3 |
| 1.1.10 | Changes since 1.10.0.4 | 3 |
| 1.1.11 | Changes since 2.0.0.1 | 3 |
| 1.1.12 | Changes since 2.1.0 | 3 |
| 1.1.13 | Changes since 2.2.0 | 3 |
| 1.1.14 | Changes since 2.3.0 | 3 |
| 1.1.15 | Changes since 3.0.0 | 4 |
| 1.1.16 | Changes since 4.0.0 | 4 |
| 1.1.17 | Changes since beta 4.0.1..... | 4 |
| 1.1.18 | Changes since 4.0.1 | 4 |
| 1.1.19 | Changes since 4.1.0 | 4 |
| 1.1.20 | Changes since 4.2.0 | 5 |
| 1.1.21 | Changes since b4.3.0 | 5 |
| 1.1.22 | Changes since 5.0.0 | 5 |
| 1.1.23 | Changes since 5.0.1 | 6 |
| 1.1.24 | Changes since 5.0.2 | 6 |
| 1.1.25 | Changes since 5.1.0 | 6 |
| 1.1.26 | Changes since 5.1.1 | 6 |
| 1.1.27 | Changes since 5.1.2 | 6 |
| 1.1.28 | Changes since 5.1.3 | 6 |
| 1.1.29 | Changes since 5.1.4 | 6 |
| 1.1.30 | Changes since 5.1.5 | 6 |
| 1.1.31 | Changes since 5.1.6 | 6 |
| 1.1.32 | Changes since 5.1.7 | 7 |
| 1.1.33 | Changes since 5.1.8 | 7 |
| 1.1.34 | Changes since 5.1.9 | 7 |
| 1.2 | Known Problems and Issues | 7 |
| 1.2.1 | Additional Known Problems and Issues with Tracker SDK 2.0.0.1 | 8 |
| 1.2.2 | Compatability Issues with 64bit Tracker SDK | 8 |
| 1.3 | Development Kit | 8 |
| 1.4 | Integrating with an Application | 9 |
| 1.4.1 | Java Specific | 10 |
| 1.4.2 | .NET Specific | 10 |
| 1.4.3 | Visual C++ Specific | 11 |
| 1.4.4 | Development..... | 11 |
| 1.4.5 | Execution..... | 11 |
| 1.4.6 | TrackCAM specifics | 11 |
| 1.4.7 | Six Dof Probe | 12 |

| | | |
|------------|--|-----------|
| 2 | CREATING THE INTERFACE..... | 14 |
| 3 | PROPERTIES | 14 |
| 3.1 | Blocking..... | 15 |
| 3.2 | Background Measurement Blocking..... | 16 |
| 3.3 | Measurement Event Rate..... | 17 |
| 3.4 | Background Measurement Event Rate..... | 17 |
| 3.5 | Exclusive Access..... | 18 |
| 4 | METHODS..... | 19 |
| 4.1 | Connect and Disconnect | 19 |
| 4.1.1 | Connect..... | 19 |
| 4.1.2 | Disconnect | 20 |
| 4.1.3 | Connected..... | 20 |
| 4.2 | Control..... | 20 |
| 4.2.1 | Toggle Sight Orientation | 21 |
| 4.2.2 | Home | 21 |
| 4.2.3 | New Home..... | 22 |
| 4.2.4 | Initialize..... | 22 |
| 4.2.5 | Initialize Smart | 23 |
| 4.2.6 | Measure Level | 24 |
| 4.2.7 | Measure Material Temperature..... | 25 |
| 4.2.8 | Measure External Temperature Sensor | 26 |
| 4.2.9 | Move..... | 26 |
| 4.2.10 | Search | 27 |
| 4.2.11 | Run Self Compensation (no User Interface)..... | 29 |
| 4.2.12 | Run Quick Compensation (no User Interface)..... | 30 |
| 4.2.13 | Run Angular Accuracy (no User Interface) | 32 |
| 4.2.14 | Run Angular Accuracy (no User Interface) with option to update the tracker compensation model | 33 |
| 4.2.15 | Application Results..... | 37 |
| 4.2.16 | Search Using Camera | 38 |
| 4.2.17 | Combo Search | 38 |
| 4.2.18 | Run Automated Compensation..... | 39 |
| 4.3 | Configuration and State..... | 41 |
| 4.3.1 | Distance Measurement | 42 |
| 4.3.2 | Motor State | 44 |
| 4.3.3 | Target Type | 45 |
| 4.3.4 | Tracking State..... | 47 |
| 4.3.5 | Target State..... | 48 |
| 4.3.6 | Backsight Orientation | 48 |
| 4.3.7 | Level Measurement Capability | 48 |
| 4.3.8 | Initialized..... | 49 |
| 4.3.9 | External Temperature Type Changeable | 49 |
| 4.3.10 | Change External Temperature Type | 49 |
| 4.3.11 | Get External Temperature Type | 51 |

| | | |
|------------|--|-----------|
| 4.3.12 | Number of External Temperature Sensors..... | 51 |
| 4.3.13 | Change Alarm Configuration | 51 |
| 4.3.14 | Get Alarm Configuration..... | 53 |
| 4.3.15 | Get Weather Information..... | 54 |
| 4.3.16 | Get Serial Number | 55 |
| 4.3.17 | Ready To Initialize | 55 |
| 4.3.18 | Get Angular Accuracy Error..... | 56 |
| 4.3.19 | Get ADM vs IFM Error | 56 |
| 4.3.20 | Change Air Temperature To Hardware | 57 |
| 4.3.21 | Change Air Pressure To Hardware | 58 |
| 4.3.22 | Change Humidity To Hardware..... | 58 |
| 4.3.23 | Change Air Temperature To Manual | 58 |
| 4.3.24 | Change Air Pressure To Manual | 58 |
| 4.3.25 | Change Humidity To Manual | 59 |
| 4.3.26 | Gesture State..... | 59 |
| 4.3.27 | Gestures Capable | 60 |
| 4.3.28 | Video Capable | 60 |
| 4.3.29 | Wireless Capable | 60 |
| 4.3.30 | Smart Warmup Wait State | 60 |
| 4.3.31 | Battery State | 61 |
| 4.3.32 | Battery Charge Remaining | 62 |
| 4.3.33 | Change Power Button State | 62 |
| 4.3.34 | Get Power Button State | 62 |
| 4.3.35 | Reboot | 63 |
| 4.3.36 | Follow Me Capable | 63 |
| 4.3.37 | Follow Me On..... | 63 |
| 4.3.38 | Change Follow Me State | 64 |
| 4.3.39 | Get Follow Me Search Radius..... | 64 |
| 4.3.40 | Change Follow me Search Radius | 64 |
| 4.3.41 | SixDof Capable | 65 |
| 4.3.42 | SixDof enabled | 65 |
| 4.3.43 | SixProbe Connected | 65 |
| 4.3.44 | SixProbe Name..... | 66 |
| 4.3.45 | SixProbe Battery Charge Remaining | 66 |
| 4.3.46 | Set Delay From Passive to Active Follow Me..... | 66 |
| 4.3.47 | Get Delay From Passive to Active Follow Me | 67 |
| 4.3.48 | SixProbe Battery State..... | 67 |
| 4.3.49 | SixProbe Battery State Change Event Supported | 67 |
| 4.3.50 | Wireless State | 68 |
| 4.3.51 | Check if capable of automatically reconnecting on communication glitch | 69 |
| 4.3.52 | Auto Compensation | 69 |
| 4.3.53 | Check if capable of generating tracker battery state change events..... | 69 |
| 4.3.54 | Probe Management | 70 |
| 4.4 | Data Collection..... | 79 |
| 4.4.1 | Foreground Measurement..... | 79 |
| 4.4.2 | Background Measurement | 81 |
| 4.4.3 | Measurement Point Data | 83 |
| 4.4.4 | Measurement Configuration | 86 |
| 4.5 | Tracker Application | 89 |
| 4.5.1 | Available Applications | 89 |
| 4.5.2 | Started Applications | 90 |
| 4.5.3 | Start Application Frame..... | 90 |
| 4.5.4 | Start Application That Is Always Top Window | 91 |
| 4.5.5 | Stop Application Frame..... | 91 |

| | | |
|-------------|--|------------|
| 4.5.6 | Modify Application | 91 |
| 4.5.7 | Get application exit status..... | 92 |
| 4.5.8 | Release the lock on the tracker | 92 |
| 4.6 | Tracker Asynchronous Messages..... | 93 |
| 4.6.1 | Start Asynchronous Messages | 93 |
| 4.6.2 | Stop Asynchronous Messages | 94 |
| 4.6.3 | Diagnostic History | 94 |
| 4.6.4 | Status History | 95 |
| 4.6.5 | Alarm History | 96 |
| 4.6.6 | Asynchronous Message Objects | 96 |
| 4.7 | Available Trackers | 101 |
| 4.7.1 | AvailableTracker | 102 |
| 4.8 | Miscellaneous Methods | 104 |
| 4.8.1 | Abort..... | 104 |
| 4.8.2 | Version Information | 104 |
| 4.8.3 | Busy | 105 |
| 4.8.4 | Busy For Current Thread..... | 105 |
| 4.8.5 | Measurement In Progress | 105 |
| 4.8.6 | Background Measurement In Progress | 106 |
| 4.8.7 | Sample Rate | 106 |
| 4.8.8 | Asynchronous Messages Started | 106 |
| 4.8.9 | Get IPAddress..... | 106 |
| 5 | EVENTS | 107 |
| 5.1 | Connection Event..... | 107 |
| 5.2 | Command Complete Event..... | 108 |
| 5.3 | Measure Data Event | 108 |
| 5.4 | Background Measure Data Event | 109 |
| 5.5 | Busy Event..... | 110 |
| 5.6 | Application Event | 111 |
| 5.7 | Diagnostic Event | 111 |
| 5.8 | Status Event | 112 |
| 5.9 | Change Event..... | 113 |
| 5.9.1 | RUN_AUTO_COMP | 115 |
| 5.10 | Alarm Event | 116 |
| 5.11 | Probe Button Event | 117 |
| 5.11.1 | Example:..... | 118 |
| 5.12 | Available Event..... | 119 |

| | | |
|------------|---|------------|
| 6 | .NET SPECIFIC ISSUES | 120 |
| 6.1 | First Instantiation Performs Loads DLL's..... | 120 |
| 6.2 | Namespace..... | 120 |
| 6.3 | Change the Installation Directory | 120 |
| 6.4 | JobjectArray | 120 |
| 6.5 | Exceptions | 121 |
| 7 | C++ SPECIFIC ISSUES..... | 121 |
| 7.1 | First Instantiation Performs Loads DLL's..... | 121 |
| 7.2 | Change the Installation Directory | 121 |
| 7.3 | Destroy Objects Passes as Parameters..... | 121 |
| 7.4 | Destroy Objects from Method Returns and Exceptions..... | 123 |
| 7.5 | Do Not Store or Destroy Event Pointer | 123 |
| 7.6 | TrkDrvObjectArray | 124 |
| 7.7 | TrkDrvException..... | 125 |
| 8 | TRACKER APPLICATIONS | 126 |
| 8.1 | General | 126 |
| 8.1.1 | Parameters | 126 |
| 8.2 | Tracker Pad | 128 |
| 8.2.1 | Parameters | 128 |
| 8.3 | Closure..... | 147 |
| 8.3.1 | Parameters | 148 |
| 8.4 | CompIT | 148 |
| 8.4.1 | FARO Tracker Version | 148 |
| 8.4.2 | 4xxx version | 149 |
| 8.5 | Startup Checks..... | 149 |
| 8.6 | MeasurePad..... | 149 |
| 8.6.1 | Starting the application..... | 149 |
| 8.6.2 | Starting parameters | 149 |
| 8.6.3 | Startup errors | 150 |
| 8.7 | HealthChecks | 151 |
| 8.7.1 | Starting parameters | 151 |
| 8.7.2 | Startup errors | 151 |

| | | |
|-------------|---------------------------------------|------------|
| 8.8 | Operational Checks | 151 |
| 8.8.1 | Starting parameters | 151 |
| 8.8.2 | Startup errors | 152 |
| 8.9 | Firmware Loader..... | 152 |
| 9 | TRACKER DRIVER DEMO | 152 |
| 9.1 | TrackerDriverDemo_VC++..... | 153 |
| 9.2 | Connect..... | 153 |
| 9.3 | Execute Command Sequence..... | 154 |
| 9.4 | Asynchronous messages | 155 |
| 9.5 | Abort Tracker Command..... | 157 |
| 9.6 | Handling Exceptions..... | 157 |
| 9.7 | Run Applications | 158 |
| 9.8 | Foreground Measurements..... | 159 |
| 9.9 | Background Measurements | 161 |
| 9.10 | Run Both Measurements..... | 163 |
| 9.11 | Listeners | 165 |
| 9.12 | Disconnect | 165 |
| 9.13 | TrackerDriverDemo_VCSharp..... | 165 |

1 Introduction

The tracker interface is a set of classes that can be used by an application to control a tracker and to collect data from it. The key class is called Tracker. All other classes in the interface support this class.

The interface itself is implemented in Java, so it can be used on various platforms. In addition, there are C++ and C# wrappers that can be used by applications built on the Windows platform.

The interface has three basic components:

- Properties
- Methods
- Events

Properties are items that are configurable in the interface itself. This includes things like if a method call returns before or after the tracker executes the command. It also includes things like event rates.

Methods are the functions that actually perform operations using the tracker. These functions may configure the tracker, read information from it, and command it to perform some operation.

Events are asynchronous signals that inform the application of something that has occurred. The application is responsible for creating the event handler and informing the interface as to what to call when the event occurs.

For all functions in the interface, errors are reported as exceptions. Each exception is an object that is typically derived from the TrackerException class. See the Tracker Reference Document for details on each exception. Functions define in this document will show typical exceptions that might be thrown but does not explain every possible exception.

If using the C++ or C# wrappers, it is possible that other types of exceptions will be thrown. Those exceptions are defined in the C++ and C# specific sections of this document.

1.1 Changes

1.1.1 Changes since 1.10.1

- This version of the SDK has new JAR files, DLLs and the JRE that is compatible with Windows VISTA OS (32 bit only).
- The .NET DLLs target Microsoft .NET 3.5 Framework. Users will need to install the 3.5 .NET Framework if not present on the target PC.
- When used on Windows 2000 OS, Service Pack 4 and security update KBB35732 are required.
- Does not work with Windows 2000 Chinese OS
- For C++ applications, a VC redistributable package is included in the SDK that is required to support VS 2008 library files (vcredist_x86.exe)
- This version of the SDK has compatibility issues with MCU firmware older than 4.4.0.1. Please see the section [#Known Problems and Issues](#) below
- MCU firmware revision 4.4.0.1 solves the above minor issues. This rev is now included in the SDK

1.1.2 Changes since 1.5.0

- Added a function to retrieve the serial number from the tracker.
- Added a function to determine if the tracker is ready to be initialized.
- Overloaded initialize() function with a boolean parameter.
- Overload initializeSmart() function with a boolean parameter.
- Overloaded initialized() function with a boolean parameter.
- Behavior of the move method in the simulator driver is corrected so that it no longer throws exception if the target is not present when the move finishes.
- Backsight method no longer throws NoResponseException if the tracker cannot acquire the target after the move. It now throws NoTarget exception as expected.

1.1.3 Changes since 1.6.0

- Added support for Camera

1.1.4 Changes since 1.7.0

- Added the .NET interface.

1.1.5 Changes since 1.7.2

- Renamed DLLs for the .NET interface to follow recommended naming convention.
- Signed the .NET DLLs with a strong name.

1.1.6 Changes since 1.7.3

- Added the ability to set the directory where .jar files, JRE directory, and bin directory, can be located.
- The Tracker.lib and TrackerUnicode.lib files are no longer compatible with VC++ 5.0. Files that are compatible with this version of VC++ will be included in the kit in the directory VC 5.0.

1.1.7 Changes since 1.8.0

- Added the ability to customize the Camera Drive dialog programmatically at startup and later
- Added the ability to get exit codes for applications via a method in the driver (currently only TrackerPad app supports it but the frame work is there to include other applications in the future)
- Added the ability to control the tracker alternately between co-operating applications through a new method (releaseLock) in the driver.
- Added a new method availableBkndMeasurements() so that calling application can know how many background measurements are available.
- Added a new method availableMeasurements() so that calling application can know how many foreground measurements are available.
- Added startSlew methods to move the tracker at specific velocities along azimuth and zenith axes. Velocities are specified in radians per second and time interval in seconds. Least allowed value for velocity is zero. A zero value for velocity will stop the movement along that axes.
- The stopSlew method stops the movement started by the startSlew command.

1.1.8 Changes since 1.9.0

- Added the ability to get application results for Pointing IT, ADM Checks and Accuracy Checks via the applicationResults(String appName) method where the appName can be one of the 3 possible values listed here, namely, "Pointing", "AngularAccuracy" and "ADM".

- The above method `applicationResults()` returns an object of type `ApplicationResults` (has time stamp and the result that can be extracted via the `getTime()` and `getResult()` methods respectively)
- Also the 3rd party developers could use the new method `isResultReportingCapable()` to find out if the MCU supports extracting the application results.

1.1.9 Changes since 1.10.0.3

- Fixed the linker issues when using the `TrackerUnicode.lib`.

1.1.10 Changes since 1.10.0.4

- Fixed the memory leak when using the C++ wrapper that returned `TrkDrvObjectArray` or `SMXObjectArray` instances.
 1. To extract the string object by calling the `getElement()` method on `TrkDrvObjectArray` or `SMXObjectArray` instance, one needs to cast the void object to `DriverStringPointer` instance.
- Fixed C# DLLs not working under Chinese & Japanese operating systems.

1.1.11 Changes since 2.0.0.1

- The new driver adds support for the new 7/8 inch windowed SMR. The interface is similar to that of 1.5 inch SMR.
-

1.1.12 Changes since 2.1.0

- The new SDK resolved the compatability issues caused by last release for the Unicode DLL.
- The new driver handles the occasional windows XP bug that corrupts the data. Data corruption is defined as bogus coordinates for the target measurement while the status indicates it as a valid measurement. This bug is fixed by ignoring the corrupt data and shipping the next non-corrupt data.

1.1.13 Changes since 2.2.0

- The new SDK added a few new API methods. Here is the list of methods and a brief explanation as to what they allow an SDK user to accomplish.
 1. `MPEResultsData checkAngularAccuracyError(SimplePointPairData fsBSMeasurements)`. See “Get Angular Accuracy Error” on page 56 for detailed information.
 2. `MPEResultsData checkADMAgainstIFM(SimplePointPairData admIFMMeasurements)`. See “Get ADM vs IFM Error” on page 56 for detailed information.
 3. `void runNoUISelfCompensation()` and related APIs. See “Run Self Compensation (no User Interface)” on page 29 for detailed information.

1.1.14 Changes since 2.3.0

- This latest 3.0.0 SDK includes the 64bit SDK as well as the 32bit version. The earlier SDK release which is 2.3.0 is a 32 bit version. All the 32 bit related files are moved into the 32bit folder and 64 bit related files are moved into the 64bit folder. There are 3 main differences between 32bit and 64bit versions.
 1. The JRE for the 64bit is different than that of 32bit.
 2. The Visual Studio redistributable is different between the 64bit and 32bit.
 3. All the interface DLLs, LIB files are different between the two revs.
- An important note: The JAR files are common between the 32bit and 64bit DLLs. In other words they are the same between the 32bit and 64bit. Infact the JAR files from the last 2.3.0 release are re-used in this release. All the API methods are the same. The users of `Tracker.h` (C++ and

Unicode SDK users) will find that this file too is the same. This implies that the users of the SDK only need to recompile their application(s) with the new 64bit interface DLLs and LIBs and use/deploy the new JRE. All the source code written will remain the same. This will make the applications be native 64bit compliant.

- A note of caution: As of this release, the 64bit SDK is developed and tested on Win 7. There might be incompatibility issues with other 64 bit OS.

1.1.15 Changes since 3.0.0

- Tracker SDK 4.0 version supports the new FARO Vantage Tracker. To connect to the Vantage Tracker, the tracker object to be constructed using the new TrackerVector keyword. An example is C++ is, new Tracker("TrackerVector").
- If wrong tracker keyword (example TrackerKeystone) is used to connect to the Vantage tracker, then, the API will throw a ConnectFailedException with the message "Tracker type specified and tracker being connected to are not same".
- New SDK methods have been added to change the air temperature, air pressure and humidity to hardware and manual.
- New SDK method to turn "Gesture Recognition" on or off is added. A C++ example would look like- changeGestureState(true) to turn on the Gesture Recognition and changeGestureState(false) to turn it off.
- A new SDK method is added to find out if current gesture recognition state is on or off. A C++ example would look like – gestureRecognitionOn(). This method returns true if the gesture recognition is on or returns false otherwise.
- Also in the 32 bit SDK folder two new demo programs are provided. The demo program TrackerDriverDemo_VC++2008 is compiled in Visual Studio 2008 and supports Vantage tracker.
- The TrackerDriverDemo_VCSsharp program is in C# and uses .net dlls. This demo will also support Vantage tracker.

1.1.16 Changes since 4.0.0

- Fixed 32 bit Unicode dlls tracker connection issue.
- Fixed MeasureLevel driver call.
- Tracker Index number can be used to connect to Vantage trackers. Tracker index number is the last four digits of the serial number. If the serial number is 1103700 the index number is 3700. A C++ example to connect to tracker using index number would be – trk->connect("3700", "user", "") assuming that the trk object is created. For Vantage tracker it is highly recommended to use index number to connect to tracker.

1.1.17 Changes since beta 4.0.1

- Added new SDK methods gesturesCapable(), videoCapable() and wirelessCapable(). These methods will return true if their respective hardware is enabled or will return false.
- Fixed the bug for Camera Drive where it occasionally locked up while closing the dialog.

1.1.18 Changes since 4.0.1

- Added new SDK method searchUsingCamera() supported by Vantage trackers only. This method commands the tracker to find the target closest in its field of view. For more information see details in the Control section below.
- Internal improvements to the tracker initialization related methods (initialize and initializeSmart).

1.1.19 Changes since 4.1.0

- Updated 64 bit jre to 1.7 version.

- Fixed bug in 64 bit Unicode dll to handle large memory access.

1.1.20 Changes since 4.2.0

- Added runNoUIQuickCompensation() and related APIs. See [Run Quick Compensation \(no User Interface\)](#) for detailed information.
- Added runNoUIAAC() and related APIs. See [Run Angular Accuracy \(no User Interface\)](#) for detailed information.
- Search method has been extended to support specifying timeout.
- Fixed bug in changeHumidityToHardware.
- Changed API calls (wirelessCapable, videoCapable, gesturesCapable) to return Boolean instead of throwing Feature Not Supported exception.
- Users need no longer specify the actual tracker type when creating the Tracker interface. Instead they can specify the IP address of the tracker or the last 4 digits of the Serial number. Based on the Tracker being connected to, the SDK constructs the appropriate Tracker type internally. This makes the 3rd party software oblivious to the tracker type they are connecting to and only provide the essential information. See [Creating the Interface](#) for more information.
- Added SDK method [smartWarmupWaitState](#) that will return the state of smart warmup progress on power up. The different wait states this API returns gives an idea of approximately how long it takes for the tracker to reach full thermal stability. Note that this API is supported for Vantage trackers only.
- Home method has been extended to check if target is present at the appropriate TMR. A new SDK method is added which can now return a success or failure of the home command based on the user preference to check for the presence of the target at the TMR. This API call is supported for Vantage trackers only. See [New Home](#) command for detailed information.
- Added new method trackerSetAndCheckInstallDir. This method will verify if the required Java SDK files (JRE folder and the JAR files) are present in the specified directory or not. Note that this API doesn't check for the presence of the DLL files.
- Camera Drive dialog for Vantage trackers is enhanced with bug fixes and new features that can be accessed when programmatically launched. See [Vantage Tracker](#) section for details.
- Added SDK method comboSearch that will use a combination of camera search and spiral search for finding targets.

1.1.21 Changes since b4.3.0

- Tracker SDK version 5.0 supports Vantage S/E tracker. To connect to the tracker, specify serial number or IP Address. To connect to the tracker using tracker keyword, the tracker object need to be constructed using the new TrackerCypher keyword. An example in C++ is, new Tracker("TrackerCypher").
- If wrong tracker keyword (example TrackerKeystone) is used to connect to the Vantage S/E tracker(s), then, the API will throw a ConnectFailedException with the message "Tracker type specified and tracker being connected to are not same".
- As Vantage S/E tracker can be used with Battery, API has been provided to determine batteryState and obtain remaining battery charge. The existing interface for Alarms can be used to get an event notification when the battery charge value goes below a setpoint.
- Additional API has been added to extend the support for running compit applications without the UI. Please see [sections](#) that describe the runNoUIXXX() methods.

1.1.22 Changes since 5.0.0

- A new filter called Interpolation filter is provided for Vantage S/E tracker(s) only.
- Bugs fixed for comboSearch(need 1.1.24 firmware and later), smartWarmupWaitState and didAACPassSinceInitialization API for Vantage S tracker.

1.1.23 Changes since 5.0.1

- A new feature Follow Me is available from this version of SDK for Vantage S/E tracker(s). Firmware version of 1.2.0 or later should be installed on trackers to access this feature. API is available to turn the feature on or off and find the targets within the specified radius.

1.1.24 Changes since 5.0.2

- Tracker SDK version 5.1.0 supports six degree of freedom probe feature. Trackers should be sixdof capable to use this feature. The API as explained in section [1.4.7](#) is available to change the target type to sixdof, probe management and obtain A, B, C angles of the probe orientation from the MeasurePointData.
- Fixed applicationResults throwing exception for “ADM” for Vantage and Vantage S/E trackers

1.1.25 Changes since 5.1.0

- Fixed probeAdpater API not returning serial number correctly.

1.1.26 Changes since 5.1.1

- Fixed bug in changing target type to 7/8 windowed target for Vantage S6/E, Vantage S/E and Vantage tracker(s).

1.1.27 Changes since 5.1.2

- Fixed bug of not generating event when probe button 4 is pressed for Vantage S6/E6 tracker(s).
- Fixed bug in Self Comp failing to save parameters on Windows 10 for FARO trackers.

1.1.28 Changes since 5.1.3

- Added API to [set](#) and [get](#) delay from passive to active follow me.

1.1.29 Changes since 5.1.4

- Added API to get [probeBatteryState](#) and generate an event when the state changes.
- Probe button events will not be generated to third party applications when buttons are pressed for java applications (For ex: Probe Comp). SDK 5.1.5 or later and MCU Firmware 2.6.1 or later is required for this fix.
- All the C++(32 bit and 64 bit), C++ Unicode(32 bit and 64 bit) and C#(32 bit and 64 bit) dlls are built using VS 2017. The demo programs(C++ and C#) are updated to VS 2017. The demo programs(C++ and C#) that were in VS 2008 are still available but are deprecated.
- Added API to find available trackers on the network.
- Added API to get ip address of the tracker.

1.1.30 Changes since 5.1.5

- The SDK is made JRE 6 complaint.
- Fixed bug that prevented connecting to the tracker due to internal probe management files.

1.1.31 Changes since 5.1.6

- Added API to turn [wireless on/off](#) for Vantage S/E and Vantage S6/E6 trackers.

- Added API to support Cat Eye target type for Vantage S/E and Vantage S6/E6 trackers.
- Improvements to find available trackers on the network faster.
- Improvements to connect API to throw better error messages that will inform the user the most likely cause of the failure.

1.1.32 Changes since 5.1.7

- Added API [trackerBatteryEventCapable](#) to determine if capable of generating battery events.
- Communications are automatically established when communication glitches occurs (more likely with WiFi). An API [autoReconnectCapable](#) method is provided to determine if this capability is available.
- Added a new event listener called [AvailableListener](#) that would generate an event when the tracker is available or not.
- SDK 5.1.8 provides Auto Compensation feature capability for Vantage S/E trackers and Vantage S6/E6 trackers.
 - Firmware 2.10.0 is required to take advantage of this feature. An API [autoCompCapable](#) is provided to determine if this feature is supported.
 - The compensation uses two predefined target locations to maintain tracker accuracy.
 - The feature supports both a no UI version and a UI version (similar to CompIT look and feel)
 - Calling [runAutomatedComp](#) API will launch a version without the UI.
 - To launch the UI version, user need to use `startApplicationFrame("CompIT", "-a AUTOMATED_COMP")`
 - Additionally, the tracker also detects and notifies if a compensation is required via a [RUN_AUTO_COMP](#) change event (only if users choose short warmup option during Startup Checks).
 - Note that the event will be cleared when the automated compensation is run.
 - If the application misses the event, an API [autoCompRequired](#) helps determine if the event was triggered earlier.

1.1.33 Changes since 5.1.8

- SDK 5.1.9 supports the new SixProbe 2.0 model for Vantage S6/E6 trackers. SixProbe 2.0 supports kinematic tips as well as improved acceptance angles for Pitch & Yaw. This latest SDK with Tracker firmware 2.11 version is needed to fully support this latest 2.0 model.
- The Probe Management Dialog which is part of the 2.11 firmware, handles probe compensation and probe checks as well as compatibility with SixProbe 1.0 and SixProbe 2.0.
- For the SDK integrators who use probe management dialog, there are no changes required, just need to redistribute the software with the latest 5.1.9 SDK.

1.1.34 Changes since 5.1.9

- Replacing Wifi keyword in error message with a more generic string.

1.2 Known Problems and Issues

- If a command complete listener is added and a tracker application is started, the listener fires on every command regardless of whether the parent application or a tracker application executes the command. Some of the tracker applications such as CompIT and Diagnostics perform checks to determine what type of hardware is installed in the tracker. These checks may generate errors, which CompIT and Diagnostics filter because it expects them. However, because the listener is added the parent application also sees these errors.
There are two possible workarounds:

- 1) Remove the command complete listener whenever a tracker application is started. Then when the application closes, add the command complete listener back in.
 - 2) Add a flag to the command complete listener that can be used to gate whether the command complete listener handles the response when the listener fires. The flag can be set when a tracker application is started and cleared when the application closes.
- The methods that start tracker applications may return before the started application is stable when blocking is turned on. This may be a problem if the parent application attempts to immediately access the driver after then return. The workaround is to put a sleep of a few seconds after returning from startApplicationFrame to allow the started application to become stable.
 - The PC Firewall must be disabled in order to get measurement data from the tracker.

1.2.1 Additional Known Problems and Issues with Tracker SDK 2.0.0.1

- The .NET DLLs are developed using the Visual Studio 2008. The DLLs might not be compatible with the Visual Studio 2003 & Visual Studio 2005.
- The Tracker Driver Demo program (described towards the end of this document) is based on Microsoft VC++ 5.0 IDE. It might work with VC++ 6.0, but will not be compatible with the newer .NET Visual Studio environments (e.g. VS 2008 cannot be used to recompile the Demo program's source code).
 - The Demo program's source code still accurately demonstrates how to use the Tracker SDK API. It is just the IDE (Integrated Development Environment) that is not compatible.
 - The EXE supplied in the Demo program works with the new Driver files (JRE, JARs, & DLLs) in this SDK except for Command->Demonstrations->Run Applications menu item.
- This version of the SDK has compatibility issues with MCU firmware older than 4.4.0.1. The following are the compatibility issues.
 - Clicking on SaveAs / Browse buttons might do nothing from HealthChecks or Diagnostics Java applications. The dialogs fail to open.
 - Diagnostics app when opened the first time might not have live data. Need to click the Refresh button to start having live data
 - Sometimes disabled buttons might have the look of enabled buttons. Clicking such buttons would not have any action.

1.2.2 Compatibility Issues with 64bit Tracker SDK

- The Tracker Camera functionality might be incompatible with the 64bit SDK as they are built using older Visual Studio IDE and older Java JRE. This would affect only the Camera Drive functionality (which is part of the TrackerPad Java application).

1.3 Development Kit

The development kit contains everything needed to interface to a tracker. These items are as follows:

- JAR files (Tracker.jar, Apps4xxx.jar, Ftp.jar, and Utility.jar) – These files are the executables for the tracker interface.
- C++ Wrapper (Tracker.h, Tracker.lib, Tracker.dll, TrackerUnicode.lib TrackerUnicode.dll) – These files allow a C++ application to access the tracker interface.
- C# Wrapper (FARO.DeviceInterface.dll, FARO.JNIBindings.dll) – These files allow a .NET application to access the tracker interface.
- Java Runtime Environment (jre directory) – All of the files in this directory make up the Java runtime environment (JRE) version 1.8 that was developed by Sun Microsystems.
- Tracker Reference Document (TrackerDoc.zip) – The files compressed in this zip file make up a reference document for all of the classes in the interface. The documentation is in HTML format, so any browser can be used to view it. The file index.html is the starting point.

- 4xxxx CompIT and 4xxx ADMComp and all related files in the \bin directory.
- Tracker simulator (AppsKeystoneSim.jar) and documentation (Tracker Simulator.rtf).
- Tracker firmware 2.7 (image.dat) – This is the first version of firmware for the Tracker 4000 / 4500 that will work with this interface.
- Example programs.
- Files to support Camera.

1.4 Integrating with an Application

The table below shows the files in the kit that are used at run time. The “Application Type Requirement” section, indicates the files that are required based the environment used for development. The “Feature Requirement” indicates what files are required for a particular feature to work properly.

To determine what files are needed, establish development and feature requirements. Any file that has a ‘*’ in a column matching your feature requirements must be included in the release version of the software.

| File or Directory | Application Type Requirement | | | Feature Requirement | | | |
|---|------------------------------|------|----------------------|---------------------|----------------------|----------|---|
| | Java | .NET | Unmanaged Visual C++ | FARO Laser Trackers | 4000 Series Trackers | TrackCAM | Vantage/Vantage S Or E/Vantage S6 or E6 |
| Tracker.jar | * | * | * | * | * | * | * |
| Ftp.jar | * | * | * | * | * | * | * |
| Utility.jar | * | * | * | * | * | * | * |
| Apps4xxx.jar | | | | | * | | |
| FARO.DeviceInterface.dll | | * | | | | | |
| FARO.JNIBindings.dll | | * | | | | | |
| Tracker.dll Or TrackerUnicode.dll | | | *1 | | | | |
| JRE directory and contents | 2 | * | * | | | | |
| Bin directory and contents | | | | | * | | |
| izmjnicom.dll | | | | | | * | |
| izmjnicomax.dll | | | | | | * | |
| VATDecoder.dll | | | | | | * | |
| VitaminCtrl.dll | | | | | | * | |
| izmcomjni.jar | | | | | | * | |
| Vitmain.jar | | | | | | * | |
| 21100527.LIC | | | | | | * | |

| | | | | | | | |
|--------------|--|--|--|--|--|---|--|
| 21100528.LIC | | | | | | * | |
| atl71.dll | | | | | | * | |

¹ Tracker.dll should be used in applications using 8-bit characters. TrackerUnicode.dll should be used in applications using 16-bit characters. A corresponding .lib file is provided for development.

² The JRE directory is not required for Java applications since it is assumed that it will already be present in the run time environment. However, the JRE directory can still be used for Java applications if desired.

1.4.1 Java Specific

1.4.1.1 Development

The tracker interface is added to a Java application by adding the JAR files from the development kit to the class path of the application. Classes can then be accessed by importing them or by referring to them by their package-qualified name.

1.4.1.2 Execution

The JAR files must be in class path at runtime.

1.4.2 .NET Specific

The .NET DLLs are developed using the Visual Studio 2005

1.4.2.1 Development

FARO.DeviceInterface.dll and FARO.JNIBindings.dll must be included in the project.

1.4.2.2 Execution

All required files and directories from the kit must be at the location specified by calling the following methods:

FARO.DeviceInterface.Cfg.setInstallDir(*pathname*)

If the path is set to an empty string or if this method is never called, the default location is the location where the .exe is located.

To specify the location of the required files and directories, the following method can also be used.

FARO.DeviceInterface.Cfg.setAndCheckInstallDir(*pathname*)

This method executes the setInstallDir() API internally. If successful, then this API checks the specified directory for the presence of the Java related files (like the JRE folder and the other .JAR files). If all the required files are found at the specified directory, the method returns true, otherwise returns false. Note that this API doesn't check for the presence of the DLL files.

The exceptions to this are the files related to the TrackCAM interface (see section 1.4.6).

1.4.3 Visual C++ Specific

1.4.4 Development

The interface is added to a Visual C++ application by performing the following steps:

- Put Tracker.h in the include path
- Include Tracker.h in any module that requires access to the interface
- Link with Tracker.lib if the application uses 8 bit character strings. Link with TrackerUnicode.lib if the application uses Unicode character strings.

1.4.5 Execution

All required files and directories from the kit must be at the location specified by calling the following method:

```
trackerSetInstallDir( pathname )  
trackerSetAndCheckInstallDir( pathname )
```

If the path is set to an empty string or if this method is never called, the default location is the location where the .exe is located.

To specify the location of the required files and directories, the following method can also be used.

```
trackerSetAndCheckInstallDir( pathname )
```

This method executes the setInstallDir() API internally. If successful, then this API checks the specified directory for the presence of the Java related files (like the JRE folder and the other .JAR files). If all the required files are found at the specified directory, the method returns true, otherwise returns false. Note that this API doesn't check for the presence of the DLL files.

The exceptions to this are the files related to the TrackCAM interface for keystone trackers(see section 1.4.6).

1.4.6 TrackCAM specifics

The tracker camera is a feature of Keystone trackers that allows the end user to drive the tracker to targets as viewed through a camera. Section 1.4 shows the files required for the interface. The table below shows the installation path for files that are not in the typical location used by the Tracker Development Kit. Note that some files must be registered.

| File Name | Register | Installation Path |
|-----------------|----------|---------------------|
| izmjnicom.dll | NO | C:\Windows\System32 |
| izmjnicomax.dll | YES | C:\Windows\System32 |
| VATDecoder.dll | YES | C:\Windows\System32 |
| VitaminCtrl.dll | YES | C:\Windows\System32 |
| 21100527.LIC | NO | C:\Windows\System32 |
| 21100528.LIC | NO | C:\Windows\System32 |
| atl71.dll | NO | C:\Windows\System32 |

After putting the files in the appropriate directories, only those files that are marked as YES in the table should be registered using regsvr32.exe. Thee DLL's are registered using regsvr32 file path + filename. If the file path is "C:\Windows\System32" and the filename is "VatDecoder.dll" then the command is regsvr32.exe "C:\Windows\System32\VatDecoder".

1.4.7 Six Dof Probe

- The six degree of freedom probe is a feature of Vantage S6/E6 trackers. SDK version 5.1.0 and later is required for SixProbe 1.0. SDK version 5.1.9 and later is required for SixProbe 2.0
- SixProbe 2.0 supports kinematic tip and wider acceptance angles for Pitch and Yaw orientations. SixProbe 1.0 has a maximum compounded acceptance of 18 degrees, while the SixProbe 2.0, allows upto 25 degrees (compounded).
- To determine if a tracker is capable of this feature or not, a method sixdof capable is available as explained in the section [4.3.41](#).
- To be able to use the sixprobe, sixdof feature must be enabled. By default sixdof feature is disabled. To enable the feature and check if it is enabled, please see section [4.3.42](#).
- One of the key features of the sixdof capable tracker is automatic recognition of the sixdof probe after connecting to the tracker.
- To connect the tracker to the sixprobe,
 - Make sure sixdof feature is enabled.
 - Hold the sixprobe in the tracker cameras field of view and lock the beam onto the sixprobe.
 - Press on any one of the four buttons located on the sixprobe for the tracker to start connecting to it.
 - While the sixprobe is connecting to the tracker, a blue led will be blinking on the sixprobe. When the led turn solid blue, sixprobe is connected to the tracker.
- To determine if it is connected, please see section [4.3.43](#).
- Third party applications can use change listener event as stated in section [5.9](#) to monitor the sixprobe connect and disconnect changes.
 - If a listener was added and asynchronous messages were started as stated in section [4.6.1](#), then a change event is generated informing that the sixprobe is connected.
 - Once connected, the sixprobe will remain connected until the user turns the power off on the sixprobe or if enable sixdof is disabled.
 - When disconnected, a change event is generated informing that the sixprobe is disconnected.
- Once connected, the tracker automatically switches to sixdof target type.
- From now on, when ever the user switches between sixprobe and other targets (SMR or mirror), tracker will automatically recognize the sixprobe and switch to sixdof target type.
- Third party applications can again use change listener event as stated in section [5.9](#) to monitor these target type change events.
 - If a listener was added and asynchronous messages were started as stated in section [4.6.1](#), then a target type change event is generated.
 - The event contains the information of the target type on which the event occurred.
 - Once the target type is switched to sixdof, it will remain so until the user locks the beam on to a different target. For ex: if there is a beam break, the target type will still be in sixdof.
 - When the beam is now locked onto a target other than sixprobe, the target type is switched to last known target.
 - For ex: If the last known target was 1.5" SMR and if the tracker is locked onto 1.5" SMR, the target type will now switch to 1.5" SMR. If the last known target was 1.5" SMR and if the tracker is locked onto 0.5" SMR, the target type will switch to 1.5" SMR as this is the last known target type.
- After connecting to the sixprobe, the tracker might be in an invalid state (i.e green lights blinking).
- See below steps on how to make the tracker valid and set an active tip for SixProbe 1.0
 - To use sixprobe to take valid measurements, users need to activate the tip that will be used and compensate it.
 - This can done using the Probe Management dialog as explained in section [8.2.1.4.1](#) from Tracker Pad. Third party applications can launch the probe management dialog or create the interface as explained in section [4.3.42](#).

- If a tip is not created, it needs to be created and compensated. The compensation will determine the tip vector of the attached tip.
 - When a tip is selected on the probe management dialog or if the changeProbe API is called informing the SDK the tip being used, this action is called activating the tip. The probe that is activated is called active tip.
 - When tip is activated and has been compensated, the tip vector is applied and the green leds on the tracker will go solid. The tracker is ready to give valid measurements.
 - When a tip is activated and has not been compensated, measurements will be invalid and the green leds on the tracker will start blinking indicating that the users have to compensate the tip.
 - When the tracker is power cycled or sixprobe disconnected and connected back, the compensation is invalidated. All the active tips are made inactive. When users connect the sixprobe to the tracker, green lights will be blinking indicating the users will have to activate a tip. If the activated tip is already compensated, then the tracker is ready to give valid measurements. If not users will have to compensate the tip.
- See below steps on how to make the tracker valid with SixProbe 2.0
 - To use SixProbe 2.0 to take valid measurements, users need to connect a kinematic tip to the sixprobe. If the tip has already been compensated, then it is ready to use. If not, the tip needs to be compensated.
 - This can be done using the Probe Management dialog as explained in section [8.2.1.4.2](#) from Tracker Pad. Third party applications can launch the probe management dialog or create the interface as explained in section [4.3.42](#).
 - If kinematic tip has not been connected, connect it and run the compensation. The compensation will determine the tip vector of the attached tip.
 - When tip has been compensated, the tip vector is applied and the green leds on the tracker will go solid. The tracker is ready to give valid measurements.
 - If a compensated kinematic tip is connected, the tip vector is applied and the green leds on the tracker will go solid and is ready to take measurements. If a non compensated kinematic tip is connected, green leds on the tracker will start blinking indicating that a compensation is required.
 - Depending on the kinematic tip connected is previously compensated or not, the SixProbe2.0 would automatically provide valid/invalid measurements.
 - Also there is no need to manually activate a tip. Attaching a kinematic tip, automatically activates that tip.
- Third party applications can use change listener to monitor the probe change event as explained in section [5.9](#). The event will contain the information of the tip that has been activated. This event can be used by third party applications to dynamically display the diameter.
- When users collect measurements with a sixprobe
 - The measurement data will contain the azimuth, zenith and distance values of the tip of the sixprobe. The measurement data will also contain the rotation A, B, C angles to determine the sixprobe orientation as explained in section [4.4.3.5](#).
 - If the target type is switched to SMR, measurement data will contain the azimuth, zenith and distance values of the center of the SMR.
 - To determine if the measurement data is sixdof or not an API is available as explained in section [4.4.3.4](#).
- The sixprobe consists of four buttons. Events are generated when these buttons are pressed and released. A Probe button listener is available for the third party software applications to configure these events as per their requirements as explained in section [5.11](#). For ex: Button 1 press can be configured to start a measurement and Button 1 release can be configured to stop the measurement.
- The charge remaining on the sixprobe battery can be obtained using the API in section [4.3.44](#). The state of the sixprobe battery can be obtained using API in section [4.3.45](#). A change event for the sixprobe battery is also generated as explained in section [4.3.45](#).
- The serial number of the sixprobe connected can be obtained using the API in section [4.3.45](#).

2 Creating the Interface

For any application to access a tracker, the first thing that must be done is to create a tracker object. One tracker object can be used for multiple trackers as long as an application only connects to one tracker at a time and each tracker is of the same type. If simultaneous connections are required, one tracker object must be created for each tracker.

The constructor for the tracker object takes one string as a parameter. The value of the string can be either the type of tracker or the IP address of the tracker or the last 4 digits of the tracker serial number.

Let us look at how to specify each of these 3 different kinds of input for creating the interface object. First kind of input is the Tracker type. This is the old way and forces the user to know up-front the kind of tracker they are connecting to.

Currently there are 3 Tracker types.

- Tracker4xxx – Indicates that the interface will be for a Tracker 4000 or 4500.
 - TrackerKeystone – Indicates that the interface will be for a FARO tracker.
 - TrackerVector – Indicates that the interface will be for a Vantage tracker.
- The disadvantage of creating this kind of interface object is that the users of the SDK need to keep up with the new Tracker types. If FARO creates a newer Tracker type, then one would need to know about it.

The Second type of input to the interface object could be the IP Address of the tracker.

This could also be challenging especially if the tracker is in wireless mode as the IP address is sort of hidden from the user.

This brings us to final type of input for creating the interface object. It is the last 4 digits of the tracker serial number. This is also the simplest input to create the interface object because any user can look this number up on the tracker(it is printed on label attached at the back of the tracker).

Below is an example of how the tracker interface was created before and how it can be created now.

Before,

```
Tracker trk = new Tracker("TrackerVector");  
trk.connect(ipadd, username, pswd);
```

Now,

```
Tracker trk = new Tracker(ipaddress or serial number);  
trk.connect(ipaddress or serial number, username, pswd);
```

Important NOTE:

For Vantage and FARO trackers, ipaddress and serial number can be specified.
For 4xx tracker only ipaddress should be specified.

Please note that "Tracker4xx", "TrackerKeystone" and "TrackerVector" can still be used to create tracker instance.

3 Properties

The interface itself has a few items that can be configured. These items are as follows:

- Blocking
- Background Measure Blocking
- Measurement Event Rate
- Background Measurement Event Rate
- Exclusive Access

3.1 Blocking

The blocking mode is changed by the `setBlocking` method and controls when certain methods will return. The current mode can be obtained by calling the `getBlocking` method. Blocking can be set at any time as long as no tracker command or measurement is currently in progress. The signatures are shown below:

| Method Signature | Language |
|---|----------|
| <code>void setBlocking(boolean state);</code> | Java |
| <code>void setBlocking(bool state);</code> | C++ |
| <code>void setBlocking(bool state);</code> | C# |

| Method Signature | Language |
|-------------------------------------|----------|
| <code>boolean getBlocking();</code> | Java |
| <code>bool getBlocking();</code> | C++ |
| <code>bool getBlocking();</code> | C# |

When blocking is set to true, methods will return after a tracker operation is completed or when a failure occurs. This includes any failure that the tracker may encounter. This mode makes implementing multi-step operations with the tracker simple. However, if these methods are called within the user interface thread, the user interface freezes until the tracker completes the operation. Long sequences of tracker commands or a tracker command that can potentially take a long time to execute (like search) should be called from a thread other than the user interface thread when blocking is enabled.

The example in Figure 1 demonstrates how setting the blocking mode to true can be used to write simple code to perform a series of operations with a tracker.

```
try
{
    Tracker      trk;

    trk = new Tracker("Tracker4xxx");

    trk.setBlocking(true);  /* Each of the following methods will not return until the tracker */
                           /* completes the command.                                     */

    trk.connect("128.128.128.100", "user", "");
    trk.home(false);
    trk.move(0.0, 1.57, 1.0, false);
    trk.backsight();
    trk.disconnect();
}
catch( TrackerException  e )
{
    /* Perform error handling */
}
```

Figure 1

When blocking is set to false, methods will return as soon as a command is sent to tracker or if an error occurs while attempting to send a command to a tracker. In other words, if a command is successfully sent to the tracker, the method returns before the tracker completes the operation.

The example in Figure 2 demonstrates how setting the blocking mode to false makes code similar to Figure 1 fail.

```
try
{
    Tracker      trk;

    trk = new Tracker("Tracker4xxx");

    trk.setBlocking(false); /* The following methods may not complete successfully. */

    /* The connect method will probably return without throwing an exception. */
    trk.connect("128.128.128.100", "user", "");

    /* The home method could throw an exception if the connection to the tracker failed or if the */
    /* connection operation did not complete yet. */
    trk.home(false);

    /* The disconnect method could throw an exception if the home command is still in */
    /* progress. */
    trk.disconnect();
}
catch( TrackerException e )
{
    /* Perform error handling */
}
```

Figure 2

The code in Figure 2 could possibly be made to work by inserting calls to the sleep method before each call to the tracker interface. The delay specified would have to be long enough to allow the previous tracker operation to complete. However, this is not provide the most efficient execution and would still not provide any indication as to whether the tracker was able to complete the previous operation successfully.

The proper solution involves the use of command complete listeners which are described in section

3.2 Background Measurement Blocking

The background measurement blocking mode is changed by the setBkndMeasureBlocking method. The current mode can be obtained by calling the getBkndMeasureBlocking method. Background measure blocking can be set at any time as long as no tracker command is currently in progress and controls when methods will return. The signatures are shown below:

| Method Signature | Language |
|---|----------|
| void setBkndMeasureBlocking(boolean state); | Java |
| void setBkndMeasureBlocking (bool state); | C++ |
| void setBkndMeasureBlocking (bool state); | C# |

| Method Signature | Language |
|------------------------------------|----------|
| boolean getBkndMeasureBlocking (); | Java |
| bool getBkndMeasureBlocking (); | C++ |
| bool getBkndMeasureBlocking (); | C# |

When blocking is set to true, the readBkndMeasurePointData method will not return until the specified number of measurements are received. When set to false, this method will return successfully if the specified number of records have already been received or throw an exception if they have not.

3.3 Measurement Event Rate

The tracker interface provides a mechanism that will generate an event whenever measurement data is received from the tracker. Calling startMeasure or startMeasurePoint starts measurements. The tracker will then send data whenever the trigger criteria are met. Measure event listeners are called when measurement events occur.

The default operation is to generate an event for each observation received from the tracker. Calling setMeasureEventRate changes the behavior of the interface so that it only generates events after receiving the specified number of operations. For example, setMeasureEventRate(5) will cause the interface to generate one measure event for every 5 observations it receives.

The current setting is obtained by calling getMeasureEventRate. The signatures are shown below.

| Method Signature | Language |
|---|----------|
| void setMeasureEventRate (int numPoints); | Java |
| void setMeasureEventRate (int numPoints); | C++ |
| void setMeasureEventRate (int numPoints); | C# |

| Method Signature | Language |
|-----------------------------|----------|
| int getMeasureEventRate (); | Java |
| int getMeasureEventRate (); | C++ |
| int getMeasureEventRate (); | C# |

3.4 Background Measurement Event Rate

The background measurement event is similar to the measurement event except that startBkndMeasure or startBkndMeasurePoint is used to start the data collection. The setBkndMeasureEventRate changes the setting, and getBkndMeasureEventRate returns the current setting.

The signatures are shown below:

| Method Signature | Language |
|---|----------|
| void setBkndMeasureEventRate (int numPoints); | Java |
| void setBkndMeasureEventRate (int numPoints); | C++ |
| void setBkndMeasureEventRate (int numPoints); | C# |

| Method Signature | Language |
|---------------------------------|----------|
| int getBkndMeasureEventRate (); | Java |
| int getBkndMeasureEventRate (); | C++ |
| int getBkndMeasureEventRate (); | C# |

3.5 Exclusive Access

When multiple threads share the same Tracker object, exclusive access can be used to “lock out” other threads. By obtaining exclusive access to the interface, that thread can continue to use the tracker as always, but any other thread that attempts to use the interface will see an `InterfaceBusyException`.

To obtain exclusive access, `setExclusiveAccess` must be called. If exclusive access is granted to the thread, the function will simply return. If exclusive access is not granted, an `InterfaceBusyException` will be thrown. Exclusive access may not be granted for the following reasons:

- The interface is busy because the current thread already has a command or foreground measurement in progress.
- The interface is busy executing a command or foreground measurement for another thread.
- Exclusive access has already been granted to another thread.

The `setExclusiveAccess` function has the following signature:

| Method Signature | Language |
|---|----------|
| <code>void setExclusiveAccess();</code> | Java |
| <code>void setExclusiveAccess();</code> | C++ |
| <code>void setExclusiveAccess();</code> | C# |

When exclusive access is granted, a `BusyEvent` is generated to inform all threads about the change. Note, the thread that is granted exclusive access will also see the busy event if it is using a busy event listener. This thread must filter the event.

The interface will remain busy for other threads until exclusive access is cleared. This is done by calling `clearExclusiveAccess`. If the thread calling this function has exclusive access, the exclusive access will be cleared. This method will throw an `InterfaceBusyException` if a thread other than the calling thread has exclusive access.

The `clearExclusiveAccess` method has the following signature:

| Method Signature | Language |
|---|----------|
| <code>void clearExclusiveAccess();</code> | Java |
| <code>void clearExclusiveAccess();</code> | C++ |
| <code>void clearExclusiveAccess();</code> | C# |

When exclusive access is cleared, a `BusyEvent` is generated to inform all threads that the interface is no longer busy.

The exclusive access state can be checked with two different functions. The first function checks if any thread has exclusive access. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>boolean exclusiveAccessSet();</code> | Java |
| <code>bool exclusiveAccessSet();</code> | C++ |
| <code>bool exclusiveAccessSet();</code> | C# |

This function returns true if any thread has exclusive access. It returns false if no thread has exclusive access.

The other function checks if the calling thread has exclusive access. It has the following signature:

| Method Signature | Language |
|--|----------|
| boolean exclusiveAccessForCurrentThread(); | Java |
| bool exclusiveAccessForCurrentThread(); | C++ |
| bool exclusiveAccessForCurrentThread(); | C# |

4 Methods

Methods are used to perform various operations that are related to the tracker itself. The methods can be divided up into the following categories:

- Connect and Disconnect
- Control
- Configuration and State
- Data Collection
- Tracker Application
- Tracker Asynchronous Messages

4.1 Connect and Disconnect

4.1.1 Connect

Once a tracker object is created, a connection must be established to the tracker itself before most of the other methods in the Tracker class can be called. This is done using the connect method. This method has the following signature:

| Method Signature | Language |
|---|----------|
| void connect(String ipAddress, String userID, String password); | Java |
| void connect(char ipAddress[], char userID, char password); | C++ |
| void connect(String ipAddress, String userID, String password); | C# |

The IP address is the address of the tracker. The user ID is the name used by the tracker to determine the access level to the tracker and can be specified as “user” for all applications. The password is the password associated with the user ID to gain access to the tracker. The user ID “user” has no password, so an empty string can be passed for this parameter (“”).

Any method that requires an established connection will throw `NotConnectedException` if the connect method has not been called.

Restrictions:

- Cannot be called if a connection is already established

Possible Events:

- Connect – if the disconnect succeeds
- Command Complete – to indicate that the command was successful or failed.
- Busy – generated twice. The first time will indicate that the command busy state is true. The second time will indicate that the command busy state is false.

4.1.2 Disconnect

The connection to a tracker can be broken by calling the disconnect method. This method has the following signature:

| Method Signature | Language |
|--------------------|----------|
| void disconnect(); | Java |
| void disconnect(); | C++ |
| void disconnect(); | C# |

Once a connection is broken by calling the disconnect command, a connection can be established with the same tracker or with another tracker by calling the connect method. Note: if the disconnect method is called when no connection is established, it will not throw an exception.

Calling connect and disconnect will generate command complete events to indicate that the method completed. In addition, if the connect or disconnect succeeds, a connect event will be generated.

Restrictions:

- Cannot be called while another command is in progress
- Cannot be called while a tracker application is running.
- Cannot be called if another thread has exclusive access.

Possible Events:

- Connect – if the disconnect succeeds
- Command Complete – to indicate that the command was successful or failed.
- Busy – generated twice. The first time will indicate that the command busy state is true. The second time will indicate that the command busy state is false.

4.1.3 Connected

The connected method returns true if a connection has been established with the tracker. It has the following signature:

| Method Signature | Language |
|----------------------|----------|
| boolean connected(); | Java |
| bool connected(); | C++ |
| bool connected(); | C# |

4.2 Control

Control methods cause the tracker to perform some type of action. Each of these methods is affected by the setting of the blocking flag.

The control methods are as follows:

- Toggle Sight Orientation
- Home
- New Home
- Initialize
- Initialize Smart
- Measure Level
- Measure Material Temperature
- Measure External Temperature Sensor
- Move
- Search

- Run Self Compensation (no UI)
- Run Quick Compensation (no UI)
- Run Angular Accuracy (no UI)
- Run Automated Compensation

The following restrictions apply to all control methods:

- A connection must be established with a tracker, otherwise `NotConnectedException` is thrown.
- A command may not already be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise `InterfaceBusyException` is thrown.

All of the following events are generated when the control methods are called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

4.2.1 Toggle Sight Orientation

The `toggleSightOrientation` method commands the tracker to perform a plunge and reverse based on its current position. If the target is present before performing the plunge and reverse, the tracker will search for the target after completing its move if necessary. If no target is present before the plunge and reverse, the tracker will not search for a target after the move completes.

The method has the following signature:

| Method Signature | Language |
|---|----------|
| <code>void toggleSightOrientation();</code> | Java |
| <code>void toggleSightOrientation();</code> | C++ |
| <code>void toggleSightOrientation();</code> | C# |

This method will typically fail for the following reasons:

- A target that was present in front sight cannot be located in back sight. A `NoTargetException` will be thrown.
- The motors are off. A `MotorStateException` will be thrown.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: `DISTANCE_DEVICE_DROPOUT`, `MANUAL POSITION CHANGE`, `NEED DISTANCE RESET`, and `NO TARGET`.
- Change – Generated if asynchronous messages have been started. Will see a code for `SIGHT_ORIENTATION`.

4.2.2 Home

The home method commands the tracker to point to the home location on the tracker. The tracker uses the current target type setting to determine which home position to use.

The method has the following signature:

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| | |
|---------------------------------|------|
| void home(boolean backsight); | Java |
| void home(bool backsight); | C++ |
| void home(bool backsight); | C# |

The backsight parameter determines the orientation in which the tracker will point to the home location. If backsight is set to false, the tracker will point to the home location in frontsight. If the backsight flag is set to true, the tracker will point to the home location in backsight.

This method will typically fail for the following reasons:

- The motors are off. A MotorStateException will be thrown.

Note: no error is generated if a target has not been placed at the home position.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: DISTANCE_DEVICE_DROPOUT, MANUAL POSITION CHANGE, NEED DISTANCE RESET, and NO TARGET.

4.2.3 New Home

The new home method does exactly like the original home method except it now returns a success or failure of the home based on whether there is a target at the home location or not. The user can add an extra parameter to the home command to check for the presence of the target.

This method blocks regardless of blocking being on or off.

The method has the following signature:

| Method Signature | Language |
|--|----------|
| boolean home(boolean backsight, boolean checkIfTargetPresent); | Java |
| bool home(bool backsight, bool checkIfTargetPresent); | C++ |
| bool home(bool backsight, bool checkIfTargetPresent); | C# |

The backsight parameter works just like in the original home command.

If the checkIfTargetPresent parameter is set to false, then this new home command behaves in the same way as the old home command. The command returns true because it fails silently when there is no target at the home location.

But if the new checkIfTargetPresent parameter is set to true, then the return value depends upon whether there is a target at the home location. If there is a target, then home is executed successfully and the return value is true. If there is no target, then the return value is false.

This new home command can be used to catch the silent home command failures because of no target at the home.

NOTE: The rest of the behavior is same as the old home command. So for example this new home method can throw general tracker exceptions (like MotorStateException etc) and generates the same asynchronous events.

4.2.4 Initialize

The initialize method commands the tracker to run its initialization sequence. This includes running the wakeup sequence which turns the motors on and finds physical stops and index locations. In addition, the tracker may run other operations such as taking dark level readings.

This method must be called if the tracker has not been initialized since it was booted. Once initialized, the application is not required to call this method again even if the application disconnects from the tracker and then connects again.

The method has the following signature:

| Method Signature | Language |
|-----------------------------------|----------|
| void initialize(boolean minimum); | Java |
| void initialize(bool minimum); | C++ |
| void initialize(bool minimum); | C# |

The minimum parameter indicates the requirements for the tracker to run the initialization sequence. If set to true, the tracker must meet the minimum requirements for initialization. If set to false, the tracker must meet all requirements before starting initialization (similar to the Startup Checks application).

The following is the equivalent of initialize(true)

| Method Signature | Language |
|--------------------|----------|
| void initialize(); | Java |
| void initialize(); | C++ |
| void initialize(); | C# |

This method will typically fail for the following reasons:

- If the tracker is not ready to be initialized. A tracker exception will be thrown. Please see section 4.3.17 for more detail on the criteria to determine the readiness.
- Tracker hardware failure in the motors or encoders. A MotorStateException will be thrown.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: MANUAL POSITION CHANGE, and MOTOR_POSITION_INACCURATE.

4.2.5 Initialize Smart

If the MCU firmware supports the “smart initialize” feature, the initialize smart method commands the tracker to run its smart initialization sequence. Otherwise, it commands the tracker to run its initialization sequence. The smart initialization sequence allow the tracker skip steps that are not necessary to complete initialization. The initializeSmart method will significantly reduce initialization time in cases where the tracker was already initialized.

The method has the following signature:

| Method Signature | Language |
|--|----------|
| void initializeSmart(boolean minimum); | Java |
| void initializeSmart(bool minimum); | C++ |
| void initializeSmart(bool minimum); | C# |

The minimum parameter works the same as specified in section 4.2.4.

Or the following , which is the equivalent of initializeSmart(true);

| Method Signature | Language |
|-------------------------|----------|
| void initializeSmart(); | Java |
| void initializeSmart(); | C++ |

| | |
|-------------------------|----|
| void initializeSmart(); | C# |
|-------------------------|----|

This method will typically fail for the following reasons:

- If the tracker is not ready to be initialized. A tracker exception will be thrown. Please see section 4.3.17 for more detail on the criteria to determine the readiness.
- Tracker hardware failure in the motors or encoders. A `MotorStateException` will be thrown.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: MANUAL POSITION CHANGE, and MOTOR_POSITION_INACCURATE.

4.2.6 Measure Level

The measure level method commands the tracker to run its level measurement sequence and return the results.

The method has the following signature:

| Method Signature | Language |
|-----------------------------|----------|
| LevelData measureLevel(); | Java |
| LevelData * measureLevel(); | C++ |
| LevelData measureLevel(); | C# |

The following restrictions are in addition to those for control methods:

- Device must be installed in the tracker; otherwise the method will throw `UnsupportedFeatureException`.

This method will typically fail for the following reasons:

- Motors are off. A `MotorStateException` is thrown.
- Tracker tilted more than the level sensor can measure. An `InternalDeviceFailureException` is thrown.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: MANUAL POSITION CHANGE and MEASURING.

If the blocking flag is set to true, the method will return the level data if the method completes successfully. If the blocking flag is set to false, this method will return null since the tracker will still be performing the operation. The level data can be obtained after a command complete event is generated indicating that the command completed successfully. The data is read by calling the `readBufferedLevel` method, which has the following signature:

| Method Signature | Language |
|----------------------------------|----------|
| LevelData readBufferedLevel(); | Java |
| LevelData * readBufferedLevel(); | C++ |
| LevelData readBufferedLevel(); | C# |

The `LevelData` class has three methods that return the rotation of the tracker about its X, Y, and Z axes with respect to gravity. The unit of measure is radians. The methods have the following signatures:

| Method Signature | Language |
|------------------|----------|
| double getRX(); | Java |
| double getRX(); | C++ |
| double getRX(); | C# |

| Method Signature | Language |
|------------------|----------|
| double getRY(); | Java |
| double getRY(); | C++ |
| double getRY(); | C# |

| Method Signature | Language |
|------------------|----------|
| double getRZ(); | Java |
| double getRZ(); | C++ |
| double getRZ(); | C# |

4.2.7 Measure Material Temperature

The measure material temperature method commands the tracker to read its material temperature sensors and return the average reading.

The method has the following signature:

| Method Signature | Language |
|--|----------|
| MaterialTemperatureData measureMaterialTemperature(); | Java |
| MaterialTemperatureData * measureMaterialTemperature (); | C++ |
| MaterialTemperatureData measureMaterialTemperature(); | C# |

This method will typically fail for the following reasons:

- No temperature sensors are being used by the tracker. An InternalDeviceFailureException will be thrown.
- One or more temperature sensors are not reading properly. An InternalDeviceFailureException will be thrown.

If the blocking flag is set to true, the method will return the material temperature data if the method completes successfully. If the blocking flag is set to false, this method will return null since the tracker will still be performing the operation. The material temperature data can be obtained after a command complete event is generated indicating that the command completed successfully. The data is read by calling the readBufferedMaterialTemperature method, which has the following signature:

| Method Signature | Language |
|---|----------|
| MaterialTemperatureData readBufferedMaterialTemperature(); | Java |
| MaterialTemperatureData * readBufferedMaterialTemperature (); | C++ |
| MaterialTemperatureData readBufferedMaterialTemperature(); | C# |

The MaterialTemperatureData class has methods that return the average temperature and the number of sensors used for the measurement. The unit of measure for the temperature is degrees Celsius. The signatures for these methods are shown below:

| Method Signature | Language |
|-----------------------|----------|
| double temperature(); | Java |
| double temperature(); | C++ |
| double temperature(); | C# |

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| | |
|-------------------|------|
| int numSensors(); | Java |
| int numSensors(); | C++ |
| int numSensors(); | C# |

4.2.8 Measure External Temperature Sensor

Each of the external temperature sensors can be read on an individual basis by using the measureExternalTempSensor method.

The method has the following signature:

| Method Signature | Language |
|---|----------|
| double measureExternalTempSensor (int sensorNum); | Java |
| double measureExternalTempSensor (int sensorNum); | C++ |
| double measureExternalTempSensor (int sensorNum); | C# |

This method will typically fail for the following reasons:

- The specified temperature sensor is 0 or greater than the number of sensors on the tracker. An exception will be thrown.
- The specified sensor is not reading properly. An InternalDeviceFailureException will be thrown.

If the blocking flag is set to true, the method will return the temperature data if the method completes successfully. If the blocking flag is set to false, this method will return 0.0 since the tracker will still be performing the operation. The temperature can be obtained after a command complete event is generated indicating that the command completed successfully. The data is read by calling the readBufferedExternalTempSensor method, which has the following signature:

| Method Signature | Language |
|---|----------|
| double readBufferedMaterialTemperature(); | Java |
| double readBufferedMaterialTemperature(); | C++ |
| double readBufferedMaterialTemperature(); | C# |

The unit of measure for the temperature is degrees Celsius. This method will fail if no data is available.

4.2.9 Move

The move method commands the tracker to point to the tracker to the specified location. If a target is found at this location and tracker is turned on, the tracker will track to the center of the target.

The method has the following signature when spherical coordinates are used:

| Method Signature | Language |
|---|----------|
| void move(double azimuth, double zenith, double distance, boolean isRelative); | Java |
| void move(double azimuth, double zenith, double distance, bool isRelative); | C++ |
| void move(double azimuth, | C# |

| | | |
|--------|------------|----|
| double | zenith, | |
| double | distance, | |
| bool | isRelative |); |

The method has the following signature when Cartesian coordinates are used:

| Method Signature | Language |
|---|----------|
| void move (double x, double y, double z, boolean isBacksight, boolean isRelative); | Java |
| void move (double x, double y, double z, bool isBacksight, bool isRelative); | C++ |
| void move (double x, double y, double z, bool isBacksight, bool isRelative); | C# |

This method will typically fail for the following reasons:

- The motors are off. A `MotorStateException` is thrown.
- An invalid location is given (such as $x=0$, $y=0$, $z=0$)

Note: no error is generated if a target has not been found at the new location.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: `DISTANCE_DEVICE_DROPOUT`, `MANUAL POSITION CHANGE`, and `NO_TARGET`.

The move method that uses spherical coordinates takes two angles and a distance as parameters. The unit of measure for the azimuth and zenith angles is radians. The unit of measure for the distance is meters. An additional parameter is used to indicate if the move is absolute or relative. If `isRelative` is set to false, the azimuth, zenith, and distance, are used as the actual location to point the tracker. If `isRelative` is set to true, these same parameters are added to the current location to obtain the new location where the tracker should point

The move method that uses Cartesian coordinates takes X, Y, and Z as parameters as well a flag to indicate if the tracker should point in backsight or frontsight. The unit of measure for X, Y, and Z is meters. If the `isBacksight` flag is set to false, the tracker points to the specified location in frontsight. If the `is` flag is set to true, its points to the specified location in backsight. An additional parameter is used to indicate if the move is absolute or relative. If `isRelative` is set to false, the X, Y, and Z, are used as the actual location to point the tracker. If `isRelative` is set to true, these same parameters are added to the current location to obtain the new location where the tracker should point.

4.2.10 Search

The search method commands the tracker to run a spiral search from its current location to find a target. If a target is found at this location and tracking is on, the tracker will track to the center of the target.

The method has the following signature:

| Method Signature | Language |
|-------------------------------|----------|
| void search(double radius); | Java |
| void search(double radius); | C++ |
| void search(double radius); | C# |

Or the following can be used to specify an approximate distance to the target.

| Method Signature | Language |
|---|----------|
| void search(double radius, double distance); | Java |
| void search(double radius, double distance); | C++ |
| void search(double radius, double distance); | C# |

Or the following can be used to specify the amount of time to search for the target.

| Method Signature | Language |
|---|----------|
| void search(double radius, int timeout); | Java |
| void search(double radius, int timeout); | C++ |
| void search(double radius, int timeout); | C# |

| Method Signature | Language |
|---|----------|
| void search(double radius, double distance , int timeout); | Java |
| void search(double radius, double distance, int timeout); | C++ |
| void search(double radius, double distance, int timeout); | C# |

Note that if the method with the distance parameter is used, it is assumed the distance is within a few centimeters of the true distance to the target. With this assumption, the tracker may search for the target faster. If the distance is not specified, the tracker may search more conservatively using the current working distance. Even with the more conservative search, the working distance must be within about 20% of the actual distance to the target.

The unit of measure for the radius parameter is meters. The tracker will search until a target is found or until the spiral reaches the specified radius. If no distance is specified, the current radial distance to the target is used. If a distance is specified, the tracker sets its working distance to that value and then runs the spiral search. The unit of measure for distance is meters.

If timeout is not specified, the tracker searches for the target for a default timeout (typically long). If timeout is specified, the tracker searches for the target for a maximum time as specified in the timeout parameter. The unit for time is milliseconds. The minimum value for the time out is 1000 milli-seconds. A tracker exception is raised if a smaller than this minimum value is used. If a target could not be found within the specified time, then a NoTargetException is thrown.

This method will typically fail for the following reasons:

- The motors are off. `MotorStateException` is thrown.
- No target is found. `NoTargetException` is thrown.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: `DISTANCE_DEVICE_DROPOUT`, `MANUAL POSITION CHANGE`, `SEARCHING_FOR_TARGET`, and `NO_TARGET`.

4.2.11 Run Self Compensation (no User Interface)

Instead of opening CompIT application and then choosing the Self Compensation application, the SDK now provides a way to run the Self Compensation procedure automatically with out the UI and update the tracker parameters. Self Compensation takes about 5 minutes. Two application events are generated when this API is called. The first event says the CompIT application is running and the second event is generated after the compensation is done. The caller of this API should listen to these two application events to know that the Self Compensation application started and exited.

Note: No commands should be issued to the tracker once this API is started. The API caller should wait till the application closing event is received before issuing any tracker command.

The method has the following signature:

| Method Signature | Language |
|---|----------|
| <code>void runNoUISelfCompensation()</code> | Java |
| <code>void runNoUISelfCompensation()</code> | C++ |
| <code>void runNoUISelfCompensation()</code> | C# |

Tracker firmware revisions 5.5.0 and above supports this API. Running this API on older firmware will throw an `UnsupportedFeatureException()`.

To avoid getting the `UnsupportedFeatureException()`, a new API method is added to check if this feature is supported or not as described below.

4.2.11.1 Check if Self Compensation without UI is supported

The method has the following signature:

| Method Signature | Language |
|--|----------|
| <code>boolean isSelfCompensationNoUICapable()</code> | Java |
| <code>boolean isSelfCompensationNoUICapable()</code> | C++ |
| <code>boolean isSelfCompensationNoUICapable()</code> | C# |

This method returns true if the Tracker has the firmware that supports the Self Compensation with out the UI. In otherwords this method returns true for firmware revisions 5.5.0 and above. Otherwise this API returns false.

The API caller can make use of this method to appropriately change their process or UI to include the no UI Self Compensation.

4.2.11.2 Abort the no UI Self Compensation

The method has the following signature:

| Method Signature | Language |
|-------------------------------------|----------|
| boolean abortNoUISelfCompensation() | Java |
| boolean abortNoUISelfCompensation() | C++ |
| boolean abortNoUISelfCompensation() | C# |

This method allows the API caller to abort the no UI Self Compensation that is started via the runNoUISelfCompensation() API.

It returns true if the command is successfully submitted to the running Self Compensation application. It returns false if the Self Compensation is started from the standard CompIT UI.

When the no UI Self Compensation is aborted, an application event is generated. This is the same event that would have been generated if the no UI Self Compensation ran to completion.

4.2.11.3 Check results of the no UI Self Compensation

The method has the following signature:

| Method Signature | Language |
|---|----------|
| ApplicationResults applicationResults(String appName) | Java |
| ApplicationResults applicationResults(String appName) | C++ |
| ApplicationResults applicationResults(String appName) | C# |

This method is supported since rev 1.10 of this Tracker SDK. The appName now can be “Self Compensation”. Please see 1.1.8 for more information.

The above method applicationResults() returns an object of type ApplicationResults (has time stamp and the result that can be extracted via the getTime() and getResult() methods respectively).

The getTime() method has the time when the no UI Self Compensation closed or aborted.

If the no UI Self Compensation ran successfully, then the result string will be “Pass”. Otherwise the result string will have the reason for failure.

For example if no UI Self Compensation is aborted via the abortNoUISelfCompensation () API, then the result string will be “Application is aborted by the user.”

4.2.12 Run Quick Compensation (no User Interface)

Quick Compensation can be launched through CompIT application for Vantage trackers. The UI that pops up takes most of the screen space. Third party applications can now avoid bringing up the UI but still run a Quick Compensation that can update the tracker parameters using this new API. The procedure takes about a minute. Two application events are generated when this API is called. The first event says the CompIT application is running and the second event is generated after the compensation is done. The caller of this API should listen to these two application events to know that the Quick Compensation application started and exited.

Note: No commands should be issued to the tracker once this API is started. The API caller should wait till the application closing event is received before issuing any tracker command.

The method has the following signature:

| Method Signature | Language |
|----------------------------------|----------|
| void runNoUIQuickCompensation() | Java |
| void runNoUIQuickCompensation() | C++ |
| void runNoUIQuickCompensation () | C# |

Vantage firmware revisions 1.1.7 and above supports this API. Running this API on older firmware will throw an `UnsupportedFeatureException()`.

To avoid getting the `UnsupportedFeatureException()`, a new API method is added to check if this feature is supported or not as described below.

Previous versions of Quick Compensation required the user to start at the home location and then move the target to the desired location. The compensation is enhanced to check at the launch to see if the tracker is locked on a valid target. If yes, the compensation is executed at this location. Also at the end of the compensation, the tracker is left locked on the same location (previously the tracker used to go to the home location).

No UI Quick Compensation will not run if the target is too close to the tracker. For example, home location is considered too close and even if the tracker is aimed at this location, the `runNoUIQuickCompensation` will fail with an exception (the details of which would show that the target is too close).

This API call is supported only for Vantage trackers.

4.2.12.1 Check if Quick Compensation without UI is supported

The method has the following signature:

| Method Signature | Language |
|---|----------|
| <code>boolean isQuickCompensationNoUICapable()</code> | Java |
| <code>boolean isQuickCompensationNoUICapable()</code> | C++ |
| <code>boolean isQuickCompensationNoUICapable()</code> | C# |

This method returns true if the Tracker has the firmware that supports the Quick Compensation without the UI. In other words, this method returns true for Vantage firmware revisions 1.1.7 and above. Otherwise, this API returns false.

The API caller can make use of this method to appropriately change their process or UI to include the no UI Quick Compensation.

4.2.12.2 Abort the no UI Quick Compensation

The method has the following signature:

| Method Signature | Language |
|---|----------|
| <code>boolean abortNoUIQuickCompensation()</code> | Java |
| <code>boolean abortNoUIQuickCompensation()</code> | C++ |
| <code>boolean abortNoUIQuickCompensation()</code> | C# |

This method allows the API caller to abort the no UI Quick Compensation that is started via the `runNoUIQuickCompensation()` API.

It returns true if the command is successfully submitted to the running Quick Compensation application. It returns false if the Quick Compensation is started from the standard CompIT UI.

When the no UI Quick Compensation is aborted, an application event is generated. This is the same event that would have been generated if the no UI Quick Compensation ran to completion.

4.2.12.3 Check results of the no UI Quick Compensation

The method has the following signature:

| Method Signature | Language |
|--|----------|
| ApplicationResults applicationResults(String appName) | Java |
| ApplicationResults* applicationResults(String appName) | C++ |
| ApplicationResults applicationResults(String appName) | C# |

This method is supported since SDK 1.1.8. However No UI Quick Compensation results can be obtained only from this release of SDK. The appName is “QuickComp” to obtain the results.

The above method applicationResults() returns an object of type ApplicationResults (has time stamp, details and the result that can be extracted via the getTime(), getDetails() and getResult() methods respectively).

The getTime() method will return the time of the last successful run.

The getResult() method will return if the compensation is a Pass or a Fail. If the no UI Quick Compensation ran successfully, the result string will be “Pass” else will be “Fail”.

The getDetails() method will return details of why the compensation is a pass or a run. For Quick Compensation if the result is a pass, there will not be any details. However if the compensation is a fail, it could be for one of the following reasons.

- If the tracker is not locked on to a valid target or
- If the tracker is too close to the target (at home location) or
- If the user cancelled the compensation using [abortNoUIQuickCompensation\(\)](#) API.
- For example if no UI Quick Compensation is aborted via the abortNoUIQuickCompensation () API, then the result will be “Fail” and the details string will be “Failed (User aborted the NO-UI Compensation).”
- If there is a beam break during measurements
- If the log file does not has write permissions

A valid target location is when tracker is pointing on a target and green light is solid.

Also, Users can use the [applicationExitCode\(\)](#) API to obtain the status of the application running.

4.2.13 Run Angular Accuracy (no User Interface)

Angular Accuracy Checks can be launched only through CompIT application. Third party applications that do not wish to do so can launch Angular Accuracy Checks without UI from SDK. Two application events are generated when this API is called. The first event says the CompIT application is running and the second event is generated after the compensation is done. The caller of this API should listen to these two application events to know that the Angular Accuracy application started and exited.

Note: No commands should be issued to the tracker once this API is started. The API caller should wait till the application closing event is received before issuing any tracker command.

The method has the following signature: This method can be used when users want to measure just the current target location.

| Method Signature | Language |
|-------------------|----------|
| void runNoUIAAC() | Java |
| void runNoUIAAC() | C++ |
| void runNoUIAAC() | C# |

Or this signature could be used when user wants to do Angular Accuray Checks at multiple locations.

| Method Signature | Language |
|------------------------------|----------|
| void runNoUIAAC(String data) | Java |
| void runNoUIAAC(String data) | C++ |
| void runNoUIAAC(String data) | C# |

To specify a single point location, Azimuth, Zenith and Distance of that location should be specified.

For ex: runNoUIAAC (“Az, Ze, D”) where Az, Ze, D are Azimuth, Zenith and Distance of the point location.

To specify multiple point locations, each point location should be separated by semi colon.

For ex: runNoUIAAC (“Az1, Ze1, D1;Az2, Ze2, D2”) where Az, Ze, D are Azimuth, Zenith and Distance of the point location.

Vnatage firmware revisions 1.1.7 and above supports this API. Running this API on older firmware will throw an UnsupportedOperationException().

To avoid getting the UnsupportedOperationException(), a new API method is added to check if this feature is supported or not as described below.

This API call is supported only for Vantage trackers.

4.2.13.1 Check if Angular Accuracy without UI is supported

The method has the following signature:

| Method Signature | Language |
|----------------------------|----------|
| boolean isAACNoUICapable() | Java |
| boolean isAACNoUICapable() | C++ |
| boolean isAACNoUICapable() | C# |

This method returns true if the Tracker has the firmware that supports the Angular Accuracy with out the UI. In otherwords this method returns true for Vantage firmware revisions 1.1.7 and above. Otherwise this API returns false.

The API caller can make use of this method to appropriately change their process or UI to include the no UI Angular Accuracy.

4.2.14 Run Angular Accuracy (no User Interface) with option to update the tracker compensation model

This is an extension of the no UI AAC API. This new API allows lot of flexibility to the user by allowing certain options to be specified during execution of the accuracy checks.

30

Note: No commands should be issued to the tracker once this API is started. The API caller should wait till the application closing event is received before issuing any tracker command. The method has the following signature: This method can be used when users want to measure just the current target location.

| Method Signature | Language |
|---|----------|
| void runNoUIAACWithCompensationOption (String data, boolean cartesianCoordinates, boolean searchForTarget, boolean compensateIfOutOfTolerance, boolean usePreviousAACMeasurements) | Java |
| void runNoUIAACWithCompensationOption (String data, boolean cartesianCoordinates, boolean searchForTarget, boolean compensateIfOutOfTolerance, boolean usePreviousAACMeasurements) | C++ |
| void runNoUIAACWithCompensationOption (String data, boolean cartesianCoordinates, boolean searchForTarget, boolean compensateIfOutOfTolerance, boolean usePreviousAACMeasurements) | C# |

Similar to the runNoUIAAC() API this API also supports one or more target locations to be specified. And similar to the runNoUIAAC variant, executing this method also generates the Application events (one has to register to receive these application events). To avoid getting the `UnsupportedFeatureException()`, a new API method is added to check if this feature is supported or not as described in the next section. The user can specify the target coordinates in Cartesian system. The first flag should be turned on to signify that the target data is x,y,z data in meters. If multiple targets are specified in the data parameter all the targets must be specified in the same Cartesian system. Each target data additionally can specify optional parameters. These include the azimuth and zenith tolerances (specified in radians) when using the combo-search to find the target. Also each front sight and back sight measurement (the pair is called a two-face measurement) that is taken is evaluated for repeatability. The way the repeatability check is done is by comparing two subsequent two-face measurements to be within a certain tolerance. If the second and first set of measurements are not within tolerance, then another set of measurements is taken to evaluate the third and the second sets and so on. This is continued till the maximum repeatability criteria is reached. The internal default is to repeat the tolerance checks for 10 subsequent measurements. The user can optionally specify this repeatability max criteria as part of each target.

Note: When multiple target locations are specified, the user can specify optional parameters for some or all the targets and each target can have different options.

Here is an example of providing these optional parameters.

Example 1: 0, 1.57, 3 implies azimuth angle is 0 radians, zenith is 1.57 radians and distance is 3 meters. Because there are no additional parameters internal default are used for the optional parameters.

Example 2: 1.1, 2.2, 3.3 with CartesianCoordinates flag turned on. This implies x,y,z coordinates in meters and internal defaults for the optional parameters.

Example 3: 0, 1.57, 3, 0.02, 0.01 This implies az, ze and distance format with azimuth tolerance set to 20 milli-radians and zenith tolerance of 10 milli-radians. The missing repeatability max criteria is set to internal default.

Example 4: 0, 1.57, 3, 0.02, 0.01, 5 This implies all the optional parameters are explicitly specified. The last number specifies that a maximum of 5 repeatability checks should be made to test for the stability of the target, failing which the target is deemed invalid and the result of the check would be a fail.

Example 5: 0, 1.57, 3, 0.02, 0.01, 5 ; 1.57, 1.57, 5 ; 1.57, 1.78, 10, 0.05, 0.01 Here 3 targets are specified (separated by semi-colons). The first target specified all the optional parameters, the second target specified none (so using all the internal defaults) and 3rd target specified only the combo-search options.

Also the user could specify if the tracker should search for a target at the specified location using the combo-search or not. Typically to save time, a user might turn off the search option. The second flag is used to turn this option on or off. It is not recommended to turn this option off unless the user is sure that the target locations being specified are accurate and that the tracker can lock on them indicated by a solid green light on the tracker. The third flag is the most important option. It is the option to use the some or all AAC

measurements for computing a new compensation model for the tracker if the AACs failed with the current compensation model. Note that the new model is only computed if the accuracy of one or more target locations is above the tracker MPE specification when evaluated using the current/existing model. If this flag is off, then this new API behaves the same as the simpler variant, namely, runNoUIAAC(String data). The last option is to use “all” the previous measurements (all previous measurements imply all those AAC measurements that were taken without disconnecting from the tracker and as long as they are within a certain time period) for evaluation if a new model needs to be computed. Consider this scenario. Imagine that the user is checking one target at a time with the compensation option turned on. Imagine that the first two points passed but the 3rd one failed. Because of the compensation option, a new compensation model will be calculated based off the 3rd target location and if this new model passed, the tracker is uploaded with it. But what if this new model is not appropriate for the first two points? Remember that those first two points were evaluated with the old compensation model. It puts the burden on the user to re-measure these first two points again because the tracker has been updated with a new compensation model. But if this option is turned on, when the new model is calculated it automatically re-evaluates the current and “all” previous measurements to make sure that all of them pass the accuracy specification before updating the tracker. Note that this last option is only applied if the previous compensation option is turned on too. Also this option makes sense if the user is evaluating one target at a time as opposed to measuring/evaluating all available targets at once.

Running this API on older firmware will throw an UnsupportedOperationException(). To avoid getting the UnsupportedOperationException(), a new API method is added to check if this new feature is supported or not as described below. This API call is supported only for Vantage trackers. NOTE: It is better to use this new API with compensation flag turned on, than calling the separate runNoUIAAC and runNoUIQuickComp APIs, for two reasons. 1. The compensation is on par with a Quick Comp or better (usually the more the target locations measured, the better is the comp compared to the Quick Comp which uses a single target location). 2. It is faster because the user does not need to re-measure/re-check the AAC after a Quick Comp.

4.2.14.1 Check if No UI Angular Accuracy with compensation option is supported

The method has the following signature:

| Method Signature | Language |
|--|----------|
| boolean isNoUIAACWithCompensationOptionSupported() | Java |
| boolean isNoUIAACWithCompensationOptionSupported() | C++ |
| boolean isNoUIAACWithCompensationOptionSupported() | C# |

This method returns true if the Tracker has the firmware that supports the no UI Angular Accuracy with the compensation option. It returns true only for Vantage firmware revisions 1.1.10 and above. Otherwise this API returns false

4.2.14.2 Abort the no UI Angular Accuracy

The method has the following signature:

| Method Signature | Language |
|-------------------------|----------|
| boolean abortNoUIAAC() | Java |
| boolean abortNoUIAAC () | C++ |
| boolean abortNoUIAAC () | C# |

This method allows the API caller to abort the no UI Angular Accuracy that is started via the [runNoUIAACQ\(\)](#) API.

It returns true if the command is successfully submitted to the running Angular Accuracy application. It returns false if the Angular Accuracy is started from the standard CompIT UI. When the no UI Angular Accuracy is aborted, an application event is generated. This is the same event that would have been generated if the no UI Angular Accuracy ran to completion.

4.2.14.3 Check results of the no UI Angular Accuracy

The method has the following signature:

| Method Signature | Language |
|--|----------|
| ApplicationResults applicationResults(String appName) | Java |
| ApplicationResults* applicationResults(String appName) | C++ |
| ApplicationResults applicationResults(String appName) | C# |

This method is supported since SDK rev 1.1.8. However Angular Accuracy results can be obtained only from this release of SDK. To get the results for Angular Accuracy Checks, use “AngularAccuracy” as the parameter to the API. Please refer to section 1.1.8 for more information.

The return value for this API is an object of type “ApplicationResults”. This object has publicAPI that allows the user to obtain the time stamp of when the application ran and the detailed results of the run. Below is the public API for the ApplicationResults object.

The getTime() method will return the time of the last successful run.

The getResults() method will return if the compensation is a Pass or a Fail. If Angular Accuracy ran successfully, the result string will be “Pass” else will be “Fail”. If all specified points are measured and all the measurements had error less than MPE, then the result string will be set to “Pass”. Otherwise the result string will be set to “Fail”.

Here are some more reasons for a failed result.

- If the tracker is not looking at valid target location on the launch of the app and no points are specified in the custom parameter list
- If the tracker could not find a valid target at even one of the specified locations when multiple points are specified
- If the user cancelled the compensation using [abortNoUIAngularAccuracy\(\)](#) API
 - The value of the details string in such a case would be “Failed (User aborted the NO-UI Compensation).”
- If CompITLog is in-accessible. Note that even during the No UI mode, the log file is updated similar to the full UI mode.
- If the beam is broken multiple times (upto 3 times) during a front-sight or back-sight measurement, the application considers it as a missed target and proceeds to the next target location in the list.

The getDetails() method will return detailed results that lead to the over all Pass/Fail result.

For all successfully measured points, the details are presented as explained below.

- For a single point, the format of the string returned would be MPE, error as %MPE.
- For multiple points the details of each measurement are separated by semi-colon. So the format of the string returned would be MPE, error as %MPE; MPE, error as %MPE and so on.
-

Note: However if one or more points are not measured, then the results would have an empty semi-colon in the order the points are measured. For example if 3 points are specified and 2nd point is not found then the details will be in the format MPE, error as %MPE;;MPE, error as %MPE.

Note: A target location will said to be valid when the tracker is aimed on a target with a solid green light.

[applicationExitCode\(\)](#) API provides the different state of an application like never run state, currently running state, closed state etc.

4.2.15 Application Results

Application Results are returned as ApplicationResults objects. The object has several methods for obtaining information about the results.

4.2.15.1 GetTime

This method will return the time in seconds of the last successful run. It has the following signature:

| Method Signature | Language |
|------------------|----------|
| long getTime(); | Java |
| long getTime(); | C++ |
| long getTime(); | C# |

4.2.15.2 GetResults

This method will return if the compensation is a Pass or a Fail. The result string will be “Pass” else will be “Fail”. It has the following signature:

| Method Signature | Language |
|----------------------|----------|
| String getResults(); | Java |
| String getResults(); | C++ |
| String getResults(); | C# |

For the reasons the result could be a pass or a fail, please see Section 4.2.12.3 for Angular Accuracy Checks and Section 4.2.11.3 for Quick Compensation.

4.2.15.3 GetDetails

This method will return detailed results that lead to the over all Pass/Fail result of the compensation that is run.

| Method Signature | Language |
|-----------------------|----------|
| String getDetails(); | Java |
| String getDetails(); | C++ |
| String getDetailss(); | C# |

For the details about Angular Accuracy Checks please see Section 4.2.12.3 and for Quick Compensation please see Section 4.2.11.3.

4.2.16 Search Using Camera

The search using camera method commands the tracker to search for the closest target in the center of its field of view. If a target is found at this location and tracking is on, tracker will track to the center of the target. This method blocks regardless of blocking being on or off. It blocks for a maximum of 10 seconds. The abort command will not cancel this command.

The method has the following signature:

| Method Signature | Language |
|----------------------------|----------|
| void searchUsingCamera (); | Java |
| void searchUsingCamera (); | C++ |
| void searchUsingCamera (); | C# |

The method will fail for the following reasons:

- If there are no target(s) in its field of view, an exception is thrown.
- At the end of the search, if tracker fails to lock on a target, an exception is thrown.

4.2.17 Combo Search

The combo search method commands the tracker to search for the closest target in the center of its field of view. If a target is found at this location and tracking is on, tracker will track to the center of the target. If target is found but is not locked onto the target, then spiral search is executed to find the target. This method blocks regardless of blocking being on or off. The abort command will not cancel this command.

Combo Search supports the below combinations to search for target.

The method has the following signature.

| Method Signature | Language |
|----------------------|----------|
| void comboSearch (); | Java |
| void comboSearch (); | C++ |
| void comboSearch (); | C# |

Or the following can be used if the tracker needs to aims to the specified azimuth, zenith and distance location and search for the target closest it its field of view. If the target is found but is not locked onto a target, spiral search is executed if the searchIfNeeded parameter is set to true. If the tracker finds and locks onto the target it is ensured that it is found within the tolerance specified in the azTol and zeTol parameters.

| Method Signature | Language |
|---|----------|
| void comboSearch (double azAbsRads, double zeAbsRads, double distmeters, double azTol, double zeTol, boolean runRegularSearchIfNeeded); | Java |
| void comboSearch (double azAbsRads, double zeAbsRads, double distmeters, double azTol, double zeTol, boolean runRegularSearchIfNeeded); | C++ |
| void comboSearch (double azAbsRads, double zeAbsRads, double distmeters, double azTol, double zeTol, boolean runRegularSearchIfNeeded); | C# |

Or the following can be used where the tracker needs to aims at the specified azimuth, zenith and distance location and search for the target closest it its field of view. If the target is found but is not locked onto a target, spiral search is executed if the searchIfNeeded parameter is set to true.

| Method Signature | Language |
|--|----------|
| void comboSearch(double azAbsRads, double zeAbsRads, double distmeters, boolean runRegularSearchIfNeeded); | Java |
| void comboSearch(double azAbsRads, double zeAbsRads, double distmeters, boolean runRegularSearchIfNeeded); | C++ |
| void comboSearch(double azAbsRads, double zeAbsRads, double distmeters, boolean runRegularSearchIfNeeded); | C# |

Or the following can be used if the tracker needs to search for a target that is closest to the center of camera's field of view. If the target is found but is not locked onto a target, spiral search is executed if the searchIfNeeded parameter is set to true. If the tracker finds the target and is locked onto the target it is ensured that it is found within the tolerance specified in the azTol and zeTol parameters.

| Method Signature | Language |
|--|----------|
| void comboSearch (double azTol, double zeTol, boolean runRegularSearchIfNeeded); | Java |
| void comboSearch (double azTol, double zeTol, boolean runRegularSearchIfNeeded); | C++ |
| void comboSearch (double azTol, double zeTol, boolean runRegularSearchIfNeeded); | C# |

All these methods will fail for the following reasons:

- If there are no target(s) in its field of view, an exception is thrown.
- At the end of the search, if tracker fails to lock on a target, an exception is thrown.

4.2.18 Run Automated Compensation

The runAutomatedComp method runs the compensation on two predefined target locations. SDK 5.1.8 and later only support this API. Firmware version 2.10.0 and later is required and tracker should be Vantage S/E or Vantage S6/E6 type to run the compensation. Below is the signature of the method.

| Method Signature | Language |
|-------------------------|----------|
| int runAutomatedComp(); | Java |
| int runAutomatedComp(); | C++ |
| int runAutomatedComp(); | C# |

These locations can be learnt by launching the Auto Compensation from CompIT or by calling this same procedure using startApplicationFrame("CompIT", "-a AUTOMATED_COMP").

If the blocking flag is set to true, the method will return compensation result. If the blocking flag is set to false, this method will return -1 since the tracker will still be performing the operation. The result can be obtained after a command complete event is generated indicating that the command completed successfully. The result is read by calling the readBufferedAutoCompResult method, which has the following signature.

| Method Signature | Language |
|----------------------------------|----------|
| int readBufferedAutoCompResult() | Java |
| int readBufferedAutoCompResult() | C++ |
| int readBufferedAutoCompResult() | C# |

The readBufferedAutoCompResult method is available from SDK 5.1.8 and later and is supported for Vantage S/E or Vantage S6/E6 trackers only. It will throw an UnsupportedOperationException when connected to a tracker that does not support this feature.

The result values can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|--|----------|
| 1 | AutomatedCompResult.PASS | Java |
| | AutomatedCompResult::PASS | C++ |
| | AutomatedCompResult.PASS | C# |
| 2 | AutomatedCompResult.FAILED_TO_FIND_TARGET_1 | Java |
| | AutomatedCompResult::FAILED_TO_FIND_TARGET_1 | C++ |
| | AutomatedCompResult.FAILED_TO_FIND_TARGET_1 | C# |
| 3 | AutomatedCompResult.FAILED_TO_FIND_TARGET_2 | Java |
| | AutomatedCompResult::FAILED_TO_FIND_TARGET_2 | C++ |
| | AutomatedCompResult.FAILED_TO_FIND_TARGET_2 | C# |
| 4 | AutomatedCompResult.LEARN_TARGETS_PROCEDURE_NOT_RUN | Java |
| | AutomatedCompResult::LEARN_TARGETS_PROCEDURE_NOT_RUN | C++ |
| | AutomatedCompResult.LEARN_TARGETS_PROCEDURE_NOT_RUN | C# |
| 5 | AutomatedCompResult.NO_TARGET_AT_HOME | Java |
| | AutomatedCompResult::NO_TARGET_AT_HOME | C++ |
| | AutomatedCompResult.NO_TARGET_AT_HOME | C# |
| 6 | AutomatedCompResult.OUT_OF_SPEC | Java |
| | AutomatedCompResult::OUT_OF_SPEC | C++ |
| | AutomatedCompResult.OUT_OF_SPEC | C# |

If the method returns 2 (FAILED_TO_FIND_TARGET_1), 3(FAILED_TO_FIND_TARGET_2) or 4(LEARN_TARGETS_PROCEDURE_NOT_RUN) values it is recommended to launch the user interface version that would run the learn targets procedure and then run the compensation mentioned above startApplicationFrame(“CompIT”, “-a AUTOMATED_COMP”).

Below is an example in c++ on how to launch the compensation.
Example:

```
C++
void launchRunAutomatedComp()
{
    //This method should be called within blocking mode to get the return value.
    //If run in non blocking mode, the cmd will return -1.
    //store the blocking state
    boolean block = trk->getBlocking();
    //turn the blocking mode on
    trk->setBlocking(true);
    int result = trk->runAutomatedComp();
    trk->setBlocking(block); //restore blocking state

    if(result == AutomatedCompResult::FAILED_TO_FIND_TARGET_1 ||
       result == AutomatedCompResult::FAILED_TO_FIND_TARGET_2 ||
       result == AutomatedCompResult::LEARN_TARGETS_PROCEDURE_NOT_RUN)
    {
        //Launching the user interface version will run the learn targets procedure and will also
        //run the compensation
        startApplicationFrame(“CompIT”, “-a AUTOMATED_COMP”);
    }
}
```

4.3 Configuration and State

Configuration and state methods control how the tracker operates or determine the current state of the tracker. These methods cannot be aborted using the abort method. Also, they do not generate command complete events and they are not affected by the setting of the blocking flag. Each of these methods returns after the tracker completes the operation.

The configuration and state methods are as follows:

- admCapable
- admOnlyCapable
- admScanCapable
- interferometerCapable
- changeDistanceMeasureMode
- distanceMeasureMode
- changeMotorState
- motorsOn
- changeTargetType
- targetType
- changeTrackingState
- trackingOn
- targetPresent
- targetLocationValid
- backsightOrientation
- levelCapable
- initialized
- externalTempTypeChangeable
- externalTempType
- changeExternalTempType
- numExternalTempSensors
- alarmCfg
- changeAlarmCfg
- availableTargetType
- getWeatherInfo
- readyToInitialize
- gesturesCapable
- videoCapable
- wirelessCapable
- thermalStabilityWaitState
- batteryState
- batteryChargeRemaining
- changePowerButtonState
- powerButtonState
- reboot
- isFollowMeCapable
- isFollowMeOn
- changeFollowMeState
- getFollowMeSearchRadius
- changeFollowMeSearchRadius
- probeBatteryChargeRemaining
- getDelayFromPassiveToActiveFollowMe
- setDelayFromPassiveToActiveFollowMe
- probeBatteryChargeRemaining
- probeBatteryState

- wirelessOn
- changeWirelessState
- autoReconnectCapable
- autoCompCapable
- autoCompRequired
- trackerBatteryEventCapable
- sixProbe2Capable
- probeModelType
- kinematicAdapterConnectedToSixProbe2

The following restrictions apply to all configuration and state methods:

- A connection must be established with a tracker, otherwise `NotConnectedException` is thrown.
- A command may not already be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise `InterfaceBusyException` is thrown.

All of the following events are generated when the configuration and state methods are called:

- **Busy** – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

4.3.1 Distance Measurement

There are four methods related to how the tracker measures distance. Each is described in the following subsections.

4.3.1.1 ADM Capable

The `admCapable` method is used to determine if the tracker has ADM hardware installed. If this method returns true, the hardware is installed. If it returns false, the hardware is not installed.

| Method Signature | Language |
|------------------------------------|----------|
| <code>boolean admCapable();</code> | Java |
| <code>bool admCapable();</code> | C++ |
| <code>bool admCapable();</code> | C# |

4.3.1.2 ADM Only Capable

the `admOnlyCapable` method is used to determine if the tracker can run in ADM only mode. If this method returns true, the ADM hardware is installed and the tracker is capable of running in ADM only mode. Otherwise, this method returns false.

| Method Signature | Language |
|--|----------|
| <code>boolean admOnlyCapable();</code> | Java |
| <code>bool admOnlyCapable();</code> | C++ |
| <code>bool admOnlyCapable();</code> | C# |

4.3.1.3 ADM Scan Capable

This method is used to determine if the tracker can be used to measure a moving target. It will return true if ADM is installed and both MCU and ADM hardware support velocity compensation, otherwise it will return false.

Note that if this function returns false, the tracker will still allow scanning with ADM. However, the radial distance measurement will have large errors.

| Method Signature | Language |
|---------------------------|----------|
| boolean admScanCapable(); | Java |
| bool admScanCapable(); | C++ |
| bool admScanCapable(); | C# |

4.3.1.4 Interferometer Capable

The interferometerCapable method is used to determine if the tracker has interferometer hardware installed. If this method returns true, the hardware is installed. If it returns false, the hardware is not installed.

| Method Signature | Language |
|----------------------------------|----------|
| boolean interferometerCapable(); | Java |
| bool interferometerCapable(); | C++ |
| bool interferometerCapable(); | C# |

4.3.1.5 Change Distance Measure Mode

The method to change distance measure mode is used to configure how the tracker should use its installed distance measurement hardware. It has the following signature:

| Method Signature | Language |
|---|----------|
| void changeDistanceMeasureMode(DistanceMode mode); | Java |
| void changeDistanceMeasureMode(DistanceMode *mode); | C++ |
| void changeDistanceMeasureMode(DistanceMode mode); | C# |

The mode parameter is an object that was created from one of the DistanceMode subclasses. The DistanceMode class is abstract, so a DistanceMode object can not be created. The classes that can be used are:

- ADMOnly
- Interferometer Only
- InterferometerSetByADM

If an ADMOnly object is passed to the changeDistanceMode method, the tracker will only use the ADM hardware to measure the distance. Note: this mode of operation is not available on 4000 series trackers.

If an InterferometerOnly object is passed to the changeDistanceMode method, the tracker will only use the interferometer to measure the distance.

If an InterferometerSetByADM object is passed to the changeDistanceMode method, the tracker will use the interferometer to measure the distance. When the beam is broken ADM will be used to set the distance for the interferometer. Note: On the 4000 series trackers, ADM only runs after calling the move or search method.

This method can fail for the following reasons:

- The required hardware is not installed in the tracker for the specified mode. Will throw `UnsupportedFeatureException`.
- The tracker does not support the specified mode of operation. Will throw `UnsupportedFeatureException`.

4.3.1.6 Get Current Distance Measure Mode

The `distanceMeasureMode` method is used to obtain the current distance measure mode from the tracker. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>DistanceMode distanceMeasureMode();</code> | Java |
| <code>DistanceMode * distanceMeasureMode();</code> | C++ |
| <code>DistanceMode distanceMeasureMode();</code> | C# |

See section 4.3.1.5 for an explanation of the `DistanceMode` class and its subclasses.

4.3.2 Motor State

Two methods relate to motor state. One method changes the state and the other provides the current state.

4.3.2.1 Change Motor State

The method to change the motor state is used to turn the motors on and off. It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>void changeMotorState(boolean on);</code> | Java |
| <code>void changeMotorState(bool on);</code> | C++ |
| <code>void changeMotorState(bool on);</code> | C# |

If `true` is passed for the `on` parameter, the motors will be turned on. If `false` is passed for the `on` parameters, the motors will be turned off. No error is generated if the motors are already in the requested state.

This method can fail for the following reasons:

- The motors have not been initialized since the tracker was booted. Throws `MotorStateException`.
- The motors failed on the tracker. Throws `InternalDeviceFailedException`.

The following events are typically generated in addition to those normally generated by control methods:

- Status – Generated if asynchronous messages have been started. The following codes are typically encountered: `MOTORS_OFF`.

4.3.2.2 Get Current Motor State

The `motorsOn` method is used to determine the current state of the motors. It has the following signature:

| Method Signature | Language |
|----------------------------------|----------|
| <code>boolean motorsOn();</code> | Java |
| <code>bool motorsOn();</code> | C++ |
| <code>bool motorsOn();</code> | C# |

This method will return `true` if the motors are on and `false` if the motors are off.

4.3.3 Target Type

There are three methods related to target type. One sets the target type, one returns the current setting and the other returns a list of target type the tracker supports. Each is described in the following subsections.

4.3.3.1 Change Target Type

The method to change the target type informs the tracker about the target that is being used. The main reason for informing the tracker about the target type is so that it knows how to follow the target. In addition, target type is used for certain operations such as home and anything that involves ADM. Having the incorrect target type may impact the accuracy of a measurement.

| Method Signature | Language |
|--|----------|
| void changeTargetType(TargetType type); | Java |
| void changeTargetType(TargetType *type); | C++ |
| void changeTargetType(TargetType type); | C# |

The type parameter is an object that was created from one of the TargetType subclasses. The TargetType class is abstract, so a TargetType object cannot be created. The classes that can be used are:

- MirrorTargetType
- SMRTargetType
- WindowedSMRTargetType
- SixDofTargetType
- CateyeTargetType

Passing a MirrorTargetType object to the changeTargetType method informs the tracker that a flat mirror is being used as the target. The constructor for the mirror target type has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| MirrorTargetType(); | Java |
| MirrorTargetType(); | C++ |
| MirrorTargetType(); | C# |

Passing an SMRTargetType object to the changeTargetType method informs the tracker that an SMR is being used as the target. When an SMRTargetType object is created, the sphere diameter in meters must be provided. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------------------|----------|
| SMRTargetType(double diameter); | Java |
| SMRTargetType(double diameter); | C++ |
| SMRTargetType(double diameter); | C# |

Passing a WindowedSMRTargetType object to the changeTargetType method informs the tracker that a windowed SMR is being used as the target. When a WindowedSMRTarget object is created, the sphere diameter in meters must be provided. Currently, only 1 1/2 inch windowed SMR is supported. The user can only change the target type to the windowed SMR if the MCU firmware supports it, otherwise an exception will be thrown. The constructor has the following signature:

| Constructor Signature | Language |
|--|----------|
| WindowedSMRTargetType (double diameter); | Java |
| WindowedSMRTargetType (double diameter); | C++ |
| WindowedSMRTargetType (double diameter); | C# |

The Windowed SMR has a radial offset in certain distance modes that the tracker compensates out. This offset can be found by calling the `getRadiusOffset` method in the `WindowedSMRTargetType` class. It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>double getRadiusOffset(DistanceMode mode);</code> | Java |
| <code>double getRadiusOffset(DistanceMode *mode);</code> | C++ |
| <code>double getRadiusOffset(DistanceMode mode);</code> | C# |

Passing a `CateyeTargetType` object to the `changeTargetType` method informs the tracker that a cateye is being used as the target. When a `CateyeTargetType` object is created, the sphere diameter in meters and radius offset in meters must be provided. The user can only change the target type to the cateye if the tracker is Vantage S/E or Vantage S6/E6 and firmware $\geq 2.6.4$ or later, otherwise an exception will be thrown. The constructor has the following signature:

| Constructor Signature | Language |
|---|----------|
| <code>CateyeTargetType (double offset, double diameter);</code> | Java |
| <code>CateyeTargetType (double offset, double diameter);</code> | C++ |
| <code>CateyeTargetType (double offset, double diameter);</code> | C# |

4.3.3.2 Get Current Target Type

The `targetType` method is used to obtain the current target being used by the tracker. It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>TargetType targetType();</code> | Java |
| <code>TargetType * targetType();</code> | C++ |
| <code>TargetType targetType();</code> | C# |

See section 4.3.3.1 for an explanation of the `TargetType` class and its subclasses.

4.3.3.3 Get Target Type List

The `availableTargetType` method is used to obtain a list of target type the tracker supports.

If the MCU firmware doesn't support the windowed target type, the following list of target type will be returned upon invoking the method:

- 1 ½ inch SMR
- Mirror
- ½ inch SMR
- 7/8 inch SMR
- User defined SMR

If the MCU firmware does support the windowed target type, an extra target type will be added to the above list before returning to the user.

- Windowed 1 1/2 inch SMR

If the tracker is Vantage S6 or E6 an extra target type will be added to the above list before returning to the user.

- SixDof target

If the tracker is Vantage S, E, S6 or E6, firmware $\geq 2.6.4$ and later only an extra target type will be added to the above list before returning to the user.

- Cateye

| Method Signature | Language |
|---|----------|
| TargetType[] availableTargetType(); | Java |
| TrkDrvObjectArray *availableTargetType(); | C++ |
| JsonObjectArray availableTargetType(); | C# |

4.3.4 Tracking State

Two methods relate to tracking state. One method changes the state and the other provides the current state.

4.3.4.1 Change Tracking State

The method to change the tracking state is used to turn the tracking on and off. It has the following signature:

| Method Signature | Language |
|---|----------|
| void changeTrackingState(boolean on); | Java |
| void changeTrackingState(bool on); | C++ |
| void changeTrackingState(bool on); | C# |

If true is passed for the on parameter, the tracking will be turned on. If false is passed for the on parameters, the tracking will be turned off. No error is generated if the current tracking state matches the requested state.

The following events are typically generated in addition to those normally generated by control methods:

- Change – Generated if asynchronous messages have been started. The following codes are typically encountered: TRACKING_MODE.

4.3.4.2 Get Current Tracking State

The trackingOn method is used to determine the current tracking state. It has the following signature:

| Method Signature | Language |
|-----------------------|----------|
| boolean trackingOn(); | Java |
| bool trackingOn(); | C++ |
| bool trackingOn(); | C# |

This method will return true if tracking is on and false if tracking is off.

4.3.5 Target State

Two methods are used to determine if the tracker has acquired a target and if the location information for the target is valid. These methods are described in the following subsections:

4.3.5.1 Target Present

The targetPresent method is used to determine if the tracker has detected a target in the laser beam path. It has the following signature:

| Method Signature | Language |
|--------------------------|----------|
| boolean targetPresent(); | Java |
| bool targetPresent(); | C++ |
| bool targetPresent(); | C# |

The method will return true if a target has been detected and false if no target has been detected.

4.3.5.2 Target Location Valid

The targetLocationValid method is used to determine if the tracker has detected a target in the laser beam path, and all information related to that position is valid. It has the following signature:

| Method Signature | Language |
|--------------------------------|----------|
| boolean targetLocationValid(); | Java |
| bool targetLocationValid(); | C++ |
| bool targetLocationValid(); | C# |

This method returns true if a target has been detected and all information about the target is valid (angular and distance information is accurate). This method returns false otherwise.

4.3.6 Backsight Orientation

The backsightOrientation method is used to determine how the tracker is pointed. It has the following signature:

| Method Signature | Language |
|---------------------------------|----------|
| boolean backsightOrientation(); | Java |
| bool backsightOrientation(); | C++ |
| bool backsightOrientation(); | C# |

The method returns true if the tracker is pointing in backsight (zenith angle is < 0). It returns false if it is pointing in front sight (zenith angle ≥ 0).

4.3.7 Level Measurement Capability

The levelCapable method is used to determine if the tracker has the hardware installed to measure the tilt of the tracker with respect to gravity. It has the following signature:

| Method Signature | Language |
|-------------------------|----------|
| boolean levelCapable(); | Java |

| Method Signature | Language |
|----------------------|----------|
| bool levelCapable(); | C++ |
| bool levelCapable(); | C# |

This method returns true if the level hardware is installed. It returns false if the level hardware is not installed.

4.3.8 Initialized

The initialized method is used to determine if a tracker has been initialized since it was booted. It has the following signature:

| Method Signature | Language |
|---------------------------------------|----------|
| boolean initialized(boolean minimum); | Java |
| bool initialized(bool minimum); | C++ |
| bool initialized(bool minimum); | C# |

It also has the following signature, which is the equivalent of initialized(true)

| Method Signature | Language |
|------------------------|----------|
| boolean initialized(); | Java |
| bool initialized(); | C++ |
| bool initialized(); | C# |

For a minimum check, the following items are checked:

- The motor state is either ON or OFF.
- The dark level has been performed.
- The azimuth and zenith encoder status doesn't have ERROR or NOT_INITIALIZED bit set.

For a full check, in addition to the minimum check, an extra item is checked also.

- The zero correct has been performed.

It will return true if it passes all of the checks, and false if it has not.

4.3.9 External Temperature Type Changeable

To determine if the temperature type can be specified by software the externalTempTypeChangeable method is called. It has the following signature:

| Method Signature | Language |
|---------------------------------------|----------|
| boolean externalTempTypeChangeable(); | Java |
| bool externalTempTypeChangeable(); | C++ |
| bool externalTempTypeChangeable(); | C# |

The method returns true if the type can be specified through the changeExternalTempType method. If it returns false, the tracker hardware determines the type of sensor on its own.

4.3.10 Change External Temperature Type

If the external temperature type can be modified, it is done using the changeExternalTempType method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void changeExternalTempType(int sensorNum, ExtTempType type); | Java |
| void changeExternalTempType(int sensorNum, ExtTempType *type); | C++ |
| void changeExternalTempType(int sensorNum, ExtTempType type); | C# |

The sensor number is greater than 0 and less than or equal to the number of sensors that are available in the tracker.

The type parameter is an object that was created from one of the ExtTempType subclasses. The ExtTempType class is abstract, so an ExtTempType object cannot be created. The classes that can be used are:

- AirExtTemp
- MaterialExtTemp
- NullExtTemp

If an AirExtTemp object is passed to the changeExternalTempType method, the tracker will use the measured temperature to calculate the wavelengths for the laser installed in the tracker. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| AirExtTemp(); | Java |
| AirExtTemp(); | C++ |
| AirExtTemp(); | C# |

If a MaterialExtTemp object is passed to the changeExternalTempType method, the tracker will use the sensor to measure material temperature. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| MaterialExtTemp(); | Java |
| MaterialExtTemp(); | C++ |
| MaterialExtTemp(); | C# |

If a NullExtTemp object is passed to the changeExternalTempType method, the tracker will not use the sensor. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| NullExtTemp(); | Java |
| NullExtTemp(); | C++ |
| NullExtTemp(); | C# |

-

This method can fail for the following reasons:

- The specified sensor does not exist in the tracker. It will throw an UnsupportedOperationException.
- The tracker does not support the setting of the temperature type because it is determined automatically. It will throw an UnsupportedOperationException.

4.3.11 Get External Temperature Type

To determine if the type of temperature sensor for a given sensor number, the `externalTempType` method is used. It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>ExtTempType externalTempType(int sensorNum);</code> | Java |
| <code>ExtTempType * externalTempType(int sensorNum);</code> | C++ |
| <code>ExtTempType externalTempType(int sensorNum);</code> | C# |

See section 4.3.10 for an explanation of the `ExtTempType` class and its subclasses.

This method can fail for the following reasons:

- The specified sensor does not exist in the tracker. It will throw an `UnsupportedFeatureException`.

4.3.12 Number of External Temperature Sensors

To determine the number of external temperature sensors available on the tracker, the `numExternalSensors` method is used. Its signature is as follows:

| Method Signature | Language |
|--|----------|
| <code>int numExternalTempSensors();</code> | Java |
| <code>int numExternalTempSensors();</code> | C++ |
| <code>int numExternalTempSensors();</code> | C# |

The number returned is the number of sensors that is available.

4.3.13 Change Alarm Configuration

4.3.13.1 ION/Vantage

Each external temperature sensor can have an alarm associated with it. The alarm for each sensor is configured using the `changeAlarmCfg` method. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>void changeAlarmCfg(int alarmNum AlarmType type);</code> | Java |
| <code>void changeAlarmCfg(int alarmNum, AlarmType *type);</code> | C++ |
| <code>void changeAlarmCfg(int alarmNum AlarmType type);</code> | C# |

The alarm number is greater than 0 and less than or equal to the number of sensors that are available in the tracker.

The type parameter is an object that was created from one of the `AlarmType` subclasses. The `AlarmType` class is abstract, so an `AlarmType` object cannot be created. The classes that can be used are:

- `DeviationAlarm`
- `HighAlarm`
- `LowAlarm`

- NullAlarm

If a DeviationAlarm object is passed to the changeAlarmCfg method, the tracker will generate an alarm whenever the measured temperature differs from the set point by the given deviation. Both the set point and deviation are specified in degrees Celsius when the object is created. The constructor has the following signature:

| Constructor Signature | Language |
|--|----------|
| DeviationAlarm(double setPoint, double deviation); | Java |
| DeviationAlarm(double setPoint, double deviation); | C++ |
| DeviationAlarm(double setPoint, double deviation); | C# |

If a HighAlarm object is passed to the changeAlarmCfg method, the tracker will generate an alarm whenever the measured temperature rises above the set point. The set point is specified in degrees Celsius when the object is created. The constructor has the following signature:

| Constructor Signature | Language |
|-------------------------------|----------|
| HighAlarm(double setPoint); | Java |
| HighAlarm(double setPoint); | C++ |
| HighAlarm(double setPoint); | C# |

If a LowAlarm object is passed to the changeAlarmCfg method, the tracker will generate an alarm whenever the measured temperature falls below the set point. The set point is specified in degrees Celsius when the object is created.

The constructor has the following signature:

| Constructor Signature | Language |
|------------------------------|----------|
| LowAlarm(double setPoint); | Java |
| LowAlarm(double setPoint); | C++ |
| LowAlarm(double setPoint); | C# |

If a NullAlarm object is passed to the changeAlarmCfg method, the specified alarm is disabled. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| NullAlarm(); | Java |
| NullAlarm(); | C++ |
| NullAlarm(); | C# |

The changeAlarmCfg method can fail for the following reasons:

- The specified sensor does not exist in the tracker. It will throw an UnsupportedOperationException.

4.3.13.2 Vantage S/E, Vantage S6/E6

The signature of the API remains same to configure an alarm for Vantage S/E or Vantage S6/E6 tracker. Below are the alarms that can be configured for these trackers. To specify the alarmNumber, a class called AlarmIndex has been provided which consists of a constant for each of the different alarms that can be triggered. This constant must be provided as the parameter for the alarmNumber.

The alarm numbers can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|---------------------------|----------|
| 0 | AlarmIndex.BATTERY_1 | Java |
| | AlarmIndex:: BATTERY_1 | C++ |
| | AlarmIndex. BATTERY_1 | C# |
| 1 | AlarmIndex. BATTERY_2 | Java |
| | AlarmIndex:: BATTERY_2 | C++ |
| | AlarmIndex. BATTERY_2 | C# |
| 2 | AlarmIndex. EXT_TEMP_1 | Java |
| | AlarmIndex:: EXT_TEMP_1 | C++ |
| | AlarmIndex. EXT_TEMP_1 | C# |
| 3 | AlarmIndex.PROBE_BATTERY | Java |
| | AlarmIndex::PROBE_BATTERY | C++ |
| | AlarmIndex.PROBE_BATTERY | C# |

The interface to configure High Alarm, Low Alarm and Null Alarm remains same.

Please Note:

- Deviation Alarm is not supported for Vantage S/E or Vantage S6/E6 tracker.
- EXT_TEMP_1 is associated with the external temperature sensor that is attached to the tracker itself.
- When configuring alarm for tracker battery or probe battery, the setpoint should be an integer value. If a double value is supplied it will be converted to integer and used.
- The alarm for probe battery is available from SDK version 5.1.0 and for Vantage S6/E6 trackers.

Below is an example how a low alarm for a set point of 20% can be configured for battery 1.

```
trk->changeAlarmCfg(AlarmIndex.BATTERY_1, new LowAlarm(20));
```

The changeAlarmCfg method for Vantage S/E or Vantage S6/E6 tracker can fail for the following reasons:

- The specified alarm number is not supported. It will throw an UnsupportedOperationException.
- If the battery or temp sensor does not exist when configuring the alarm.
- If tracker is not sixdof capable or if probe is not connected to the tracker when configuring alarm for probe battery.

4.3.14 Get Alarm Configuration

4.3.14.1 ION/Vantage

To determine the type of alarm for a given alarm number, the alarmCfg method is used. It has the following signature:

| Method Signature | Language |
|---------------------------------------|----------|
| AlarmType alarmCfg(int alarmNumber); | Java |
| AlarmType *alarmCfg(int alarmNumber); | C++ |
| AlarmType alarmCfg(int alarmNumber); | C# |

See section 4.3.13 for an explanation of the AlarmType class and its subclasses.

This method can fail for the following reasons:

- The specified alarm number does not exist in the tracker. It will throw an `UnsupportedFeatureException`.

4.3.14.2 Vantage S/E, Vantage S6/E6

The signature of the API remains same. To get the alarm configuration an alarm number must be specified. To specify the alarmNumber, a class called `AlarmIndex` has been provided which consists of a constant for each alarm. This constant must be provided as the parameter for the alarmNumber.

To determine the type of alarm configured for `BATTERY_1`.

```
AlarmType type = trk->alarmCfg(AlarmIndex.BATTERY_1);
LowAlarm* lowAlarm = dynamic_cast<LowAlarm*>(type);
if(lowAlarm)
{
    double setPoint = lowAlarm->getSetPoint();
}
else
{
    HighAlarm* highAlarm = dynamic_cast<HighAlarm*>(type);
    if(highAlarm)
    {
        double setPoint = highAlarm->getSetPoint();
    }
    else
    {
        NullAlarm* nullAlarm = dynamic_cast<NullAlarm*>(type);
        if(nullAlarm)
        {
            // ...
        }
    }
}
```

The method for Vantage S/E or or Vantage S6/E6 tracker can fail for the following reasons:

- The specified alarm number is not supported. It will throw an `UnsupportedFeatureException`.

4.3.15 Get Weather Information

This method is used to return air temperature, air pressure, and humidity to the user. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>WeatherInformation getWeatherInfo();</code> | Java |
| <code>WeatherInformation *getWeatherInfo();</code> | C++ |
| <code>WeatherInformation getWeatherInfo();</code> | C# |

This method can fail for the following reasons:

- If the status of air temperature is not valid or not in the manual mode.
- If the status of air pressure is not valid or not in the manual mode.
- If the status of humidity is not valid or not in the manual mode.

WeatherInformation is a class that has the following public methods to return the weather information to a user:

The air temperature in degrees C is obtained using the following method:

| Method Signature | Language |
|-----------------------------|----------|
| double getAirTemperature(); | Java |
| double getAirTemperature(); | C++ |
| double getAirTemperature(); | C# |

The air pressure in mmHg is obtained using the following method:

| Method Signature | Language |
|--------------------------|----------|
| double getAirPressure(); | Java |
| double getAirPressure(); | C++ |
| double getAirPressure(); | C# |

The humidity as a percentage is obtained using the following method:

| Method Signature | Language |
|-----------------------|----------|
| double getHumidity(); | Java |
| double getHumidity(); | C++ |
| double getHumidity(); | C# |

4.3.16 Get Serial Number

This method is used to return the serial number of the tracker. It has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| String getSerialNumber(); | Java |
| char* getSerialNumber(); | C++ |
| String getSerialNumber(); | C# |

4xxx driver always returns “4xxx” as the serial number and simulator driver always returns “Simulator” as the serial number.

- Keystone driver reads the appropriate information from the tracker and constructs a string representing the serial number of the tracker based on the serial number format

4.3.17 Ready To Initialize

This method is used to determine if the tracker is ready to be initialized. User can specify if minimum readiness (pass in “true” as the parameter) or full readiness (pass in “false” as the parameter) is required to determine the tracker state. It has the following signature:

| Method Signature | Language |
|--|----------|
| boolean readyToInitialize(boolean minimum) | Java |
| bool readyToInitialize(bool minimum) | C++ |
| bool readyToInitialize(bool minimum) | C# |

4xxx driver checks the following items regardless of the parameter passed in. If both items pass the check, a value of true is returned. Otherwise, a value of false is returned.

- Shutter is open

- Laser is on and locked.

Simulator driver always returns a value of true.

Keystone driver behaves different depending upon the parameter passed in. For a minimum readiness, the method returns a value of true if either of the following requirements is met. In any other cases, a value of false will be returned.

- If the tracker is IFM capable, the laser is on and locked.
- If the tracker is only ADM capable.

For a full readiness (similar to Startup Checks), the method returns a value of true if either of the following requirements is met. In any other cases, a value of false will be returned.

- The smart warm up is done and if the tracker is IFM capable, the laser is on and locked.
- The smart warm up is done and if the tracker is only ADM capable.

4.3.18 Get Angular Accuracy Error

This method can be used to return the angular accuracy error for a given target location. Angular accuracy error is also known as back sight error. The result from this method is the same as that obtained from running the Angular Accuracy Checks application from the standard CompIT UI. It has the following signature:

| Method Signature | Language |
|--|----------|
| MPEResultsData checkAngularAccuracyError(SimplePointPairData fsBSMeasurements) | Java |
| MPEResultsData checkAngularAccuracyError(SimplePointPairData fsBSMeasurements) | C++ |
| MPEResultsData checkAngularAccuracyError(SimplePointPairData fsBSMeasurements) | C# |

There are two new object types introduced in the SDK for this API.

- MPEResultsData – This class has all the information about the angular accuracy error. Error, MPE, error as percentage of MPE and pass/fail are all available as member access methods. Error and MPE are in meters and percentage is value between 0 and 1.
- SimplePointPairData – This class holds data for 2 measurements. It takes 6 doubles (the first three are azimuth, zenith and distance of front sight measurement and the next three are azimuth, zenith and distance of the same point in backsight orientation). Azimuth and zenith angles are in radians and distance in meters. Azimuth, zenith and distance are obtained from a MeasurePointData object.

4.3.19 Get ADM vs IFM Error

This method can be used to check the accuracy of ADM against IFM for a tracker that is equipped with both ADM and IFM hardware. Note that the result is not as accurate as when the ADM Checks application is run from the standard CompIT UI.

It has the following signature:

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| Method Signature | Language |
|---|----------|
| MPEResultsData checkADMAgainstIFM(SimplePointPairData admIFMMeasurements) | Java |
| MPEResultsData checkADMAgainstIFM(SimplePointPairData admIFMMeasurements) | C++ |
| MPEResultsData checkADMAgainstIFM(SimplePointPairData admIFMMeasurements) | C# |

There are two new object types introduced in the SDK for this API.

- MPEResultsData – This class has all the information about the ADM vs IFM error. Error, MPE, error as percentage of MPE and pass/fail are all available as member access methods. Error and MPE are in meters and percentage is value between 0 and 1.
- SimplePointPairData – This class holds data for 2 measurements. It takes 6 doubles (the first three are azimuth, zenith and distance from a measurement using ADM only mode and the next three are azimuth, zenith and distance of the same point using IFM only mode). Azimuth and zenith angles are in radians and distance in meters. Azimuth, zenith and distance are obtained from a MeasurePointData object.

4.3.20 Change Air Temperature To Hardware

Will change the air temperature input to hardware.

This means user gets the current air temperature given by the tracker when the method getAirTemperature() is used mentioned in section [4.3.15](#).

| Method Signature | Language |
|---|----------|
| double changeAirTemperatureToHardware (); | Java |
| double changeAirTemperatureToHardware (); | C++ |
| double changeAirTemperatureToHardware (); | C# |

```
//To get the current air temperature create the weather information
//object and get the temperature
WeatherInformation* info = trk->getWeatherInfo();
double temp = info->getAirTemperature();

//now change the air temperature to manual by setting the temperature
trk->changeAirTemperatureToManual(10);

//wait like 5 seconds to allow the tracker to change to manual

//Now get the air temperature from the weather info which should
//give the temperature set by the user
info = trk->getWeatherInfo();
temp = info->getAirTemperature();configuring alarm for probe battery

//can reset it to hardware
trk->changeAirTemperatureToHardware();
```


4.3.21 Change Air Pressure To Hardware

Will change the air pressure input to hardware.

This means user gets the current air pressure given by the tracker when the method `getAirPressure()` is used mentioned in section [4.3.15](#).

| Method Signature | Language |
|---|----------|
| <code>double changeAirPressureToHardware ();</code> | Java |
| <code>double changeAirPressureToHardware ();</code> | C++ |
| <code>double changeAirPressureToHardware ();</code> | C# |

4.3.22 Change Humidity To Hardware

Will change the humidity input to hardware.

This means user gets the current humidity given by the tracker when the method `getHumidity()` is used mentioned in section [4.3.15](#).

| Method Signature | Language |
|--|----------|
| <code>double changeHumidityToHardware ();</code> | Java |
| <code>double changeHumidityToHardware ();</code> | C++ |
| <code>double changeHumidityToHardware ();</code> | C# |

4.3.23 Change Air Temperature To Manual

The air temperature input is changed to manual and is set to the specified temperature. Units of temperature are degC.

- This means that the tracker will use the air temperature set by the user using this method.
- Also when the method `getAirTemperature()` is called mentioned in section 4.3.15, user will get the air temperature set by this method.
- The method will throw a `TrackerException` if the air temperature value is not within -20.0 to 60 degC.

| Method Signature | Language |
|--|----------|
| <code>double changeAirTemperatureToHardware (double airTemp);</code> | Java |
| <code>double changeAirTemperatureToHardware (double airTemp);</code> | C++ |
| <code>double changeAirTemperatureToHardware (double airTemp);</code> | C# |

4.3.24 Change Air Pressure To Manual

The air pressure input is changed to manual and is set to the specified pressure. Units of pressure are mmHG.

- This means that the tracker will use the air pressure set by the user using this method.
- Also when the method `getAirPressure()` is called mentioned in section [4.3.15](#), user will get the air pressure set by this method.
- The method will throw a `TrackerException` if the air pressure value is not within 225 to 900.0 mmHG

| Method Signature | Language |
|---|----------|
| double changeAirPressureToManual (double airPress); | Java |
| double changeAirPressureToManual (double airPress); | C++ |
| double changeAirPressureToManual (double airPress); | C# |

4.3.25 Change Humidity To Manual

The humidity input is changed to manual and is set to the specified humidity. Units of humidity are percent.

- This means that the tracker will use the humidity set by the user using this method.
- Also when the method getHumidity() is called mentioned in section [4.3.15](#), user will get the humidity set by this method.
- The method will throw a TrackerException if the humidity value is not within 0 to 100%

| Method Signature | Language |
|--|----------|
| double changeHumidityToManual (double humidity); | Java |
| double changeHumidityToManual (double humidity); | C++ |
| double changeHumidityToManual (double humidity); | C# |

4.3.26 Gesture State

Two methods relate to gestures. One method changes the state and the other provides the current state.

4.3.26.1 Change Gesture State

The method to change the gesture state is used to turn the gesture on and off. It has the following signature:

| Method Signature | Language |
|--|----------|
| void changeGestureState(boolean on); | Java |
| void changeGestureState(bool on); | C++ |
| void changeGestureState(bool on); | C# |

If true is passed for the on parameter, gestures will be turned on. If false is passed for the on parameters, the gestures will be turned off. No error is generated if the gestures are already in the requested state.

Gesture state is persistent through power cycles for Vantage S/E or Vantage S6/E6 trackers from 5.0.2 SDK version.

4.3.26.2 Get Current Gesture State

The gestureRecognitionOn method is used to determine the current state of gestures. It has the following signature:

| Method Signature | Language |
|----------------------------------|----------|
| boolean gestureRecognitionOn (); | Java |
| bool gestureRecognitionOn (); | C++ |
| bool gestureRecognitionOn (); | C# |

This method will return true if the gestures are on and false if the gestures are off.

4.3.27 Gestures Capable

The gesturesCapable method is used to determine if the tracker has gestures enabled are not. If this method returns true, gestures are enabled. If it returns false, then gestures are not enabled.

| Method Signature | Language |
|----------------------------|----------|
| boolean gesturesCapable(); | Java |
| bool gesturesCapable(); | C++ |
| bool gesturesCapable(); | C# |

4.3.28 Video Capable

The videoCapable method is used to determine if the tracker has video enabled are not. If this method returns true, video is enabled. If it returns false, then video is not enabled.

| Method Signature | Language |
|-------------------------|----------|
| boolean videoCapable(); | Java |
| bool videoCapable(); | C++ |
| bool videoCapable(); | C# |

4.3.29 Wireless Capable

The wirelessCapable method is used to determine if the tracker has wireless enabled are not. If this method returns true, wireless is enabled. If it returns false, then wireless is not enabled.

| Method Signature | Language |
|----------------------------|----------|
| boolean wirelessCapable(); | Java |
| bool wirelessCapable(); | C++ |
| bool wirelessCapable(); | C# |

4.3.30 Smart Warmup Wait State

This method will return the state of smart warmup progress on power up. The different wait states this API returns give an idea of approximately how long it takes for the tracker to complete the smart warmup (tracker will be close to being at thermal equilibrium with its surroundings).

Note that this API call is supported for Vantage trackers only. As this API depends on the smart warmup progress, there should be at least one external air temperature sensor plugged into the MCU.

| Method Signature | Language |
|------------------------------|----------|
| int smartWarmupWaitState (); | Java |
| int smartWarmupWaitState (); | C++ |
| int smartWarmupWaitState (); | C# |

This method returns the following values:

- 0 = Done or Aborted. This indicates that the tracker is no longer doing smart warmup either because the tracker reached the thermal equilibrium or if the user aborted the smart warmup by waking up the motors
- 1 = Extended. This indicates that tracker is going to take >45 minutes to complete smart warmup.
- 2 = Long. This indicates that the tracker is going to take >20 minutes and <= 45 minutes to complete smart warmup.
- 3 = Short. This indicates that the tracker is going to take >5 minutes and <= 20 minutes to complete smart warmup.

- 4 = Quick. This indicates that the tracker is going to take <= 5 minutes to complete smart warmup.

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|--------------------------------|----------|
| 0 | SmartWarmupWaitState.Done | Java |
| | SmartWarmupWaitState::Done | C++ |
| | SmartWarmupWaitState.Done | C# |
| 1 | SmartWarmupWaitState.Extended | Java |
| | SmartWarmupWaitState::Extended | C++ |
| | SmartWarmupWaitState.Extended | C# |
| 2 | SmartWarmupWaitState.Long | Java |
| | SmartWarmupWaitState::Long | C++ |
| | SmartWarmupWaitState.Long | C# |
| 3 | SmartWarmupWaitState.Short | Java |
| | SmartWarmupWaitState::Short | C++ |
| | SmartWarmupWaitState.Short | C# |
| 4 | SmartWarmupWaitState.Quick | Java |
| | SmartWarmupWaitState::Quick | C++ |
| | SmartWarmupWaitState.Quick | C# |

4.3.31 Battery State

Battery pack consists of two batteries. The state for each battery can be obtained by providing the battery number.

| Method Signature | Language |
|--------------------------------------|----------|
| int batteryState(int batteryNumber); | Java |
| int batteryState int batteryNumber); | C++ |
| int batteryState(int batteryNumber); | C# |

The batteryNumber parameter is the number of the battery on the battery pack whose state needs to be obtained.

The following are the states of the battery that are returned.

- 0 = Accurate: Battery is present and is in accurate state.
- 1 = Inaccurate: Battery is present but is not in accurate state. So the charge value returned cannot be trusted.
- 2 = Warning: Battery has been depleted to a very low level.
- 3 = Critical: Battery has been depleted to such a low level that battery stops discharging. The red light on the battery pack starts blinking.
- 4 = Not_Present: Battery is not present in the battery pack or battery is in error state.

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|--------------------------|----------|
| 0 | BatteryState.ACCURATE | Java |
| | BatteryState::ACCURATE | C++ |
| | BatteryState.ACCURATE | C# |
| 1 | BatteryState.INACCURATE | Java |
| | BatteryState::INACCURATE | C++ |
| | BatteryState.INACCURATE | C# |
| 2 | BatteryState.WARNING | Java |

| Value | Reference | Language |
|-------|---------------------------|----------|
| | BatteryState::WARNING | C++ |
| | BatteryState.WARNING | C# |
| 3 | BatteryState.CRITICAL | Java |
| | BatteryState::CRITICAL | C++ |
| | BatteryState.CRITICAL | C# |
| 4 | BatteryState.NOT_PRESENT | Java |
| | BatteryState::NOT_PRESENT | C++ |
| | BatteryState.NOT_PRESENT | C# |

The method can fail for the following reasons:

- If the specified battery number is not 1 or 2, then an `UnsupportedFeatureException` is thrown.
- If the battery is in error, an `IntenalDeviceFailure` exception is thrown.

4.3.32 Battery Charge Remaining

The charge remaining in each battery can be obtained using this method.

| Method Signature | Language |
|---|----------|
| <code>int batteryChargeRemaining(int batteryNumber);</code> | Java |
| <code>int batteryChargeRemaining(int batteryNumber);</code> | C++ |
| <code>int batteryChargeRemaining(int batteryNumber);</code> | C# |

The `batteryNumber` parameter is the number of the battery whose charge needs to be obtained.

The method can fail for the following reasons:

- If the specified battery number is not 1 or 2, then an `UnsupportedFeatureException` is thrown.
- If battery is not present, an `IntenalDeviceFailure` exception is thrown.
- If battery is in error, an `IntenalDeviceFailure` exception is thrown.

4.3.33 Change Power Button State

By default, user needs to press the power button to turn the tracker on after plugging the tracker's power cable into AC power. This behavior can be changed using change power state method.

It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>void changePowerButtonState(boolean state);</code> | Java |
| <code>void changePowerButtonState(bool state);</code> | C++ |
| <code>void changePowerButtonState(bool state);</code> | C# |

If `true` is passed for the state parameter, the tracker will be turned on automatically without the need to press the power button after plugging the power cable to AC power. If `false` is passed, then the tracker will return to its default behaviour. No error is generated if the current power state matches the requested state.

4.3.34 Get Power Button State

This method is used to determine the current power button state. It has the following signature:

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| Method Signature | Language |
|-----------------------------|----------|
| boolean powerButtonState(); | Java |
| bool powerButtonState(); | C++ |
| bool powerButtonState(); | C# |

This method will return true if power button state is on and false if off.

4.3.35 Reboot

This is a very advanced API method which should be used with extreme caution and only in very rare cases. Calling this API will disconnect from the tracker, power cycle and establish the connection back to the tracker. So users should make sure that they stop all the tracker communications like measurements before calling this API. After the command is finished, user will have to take care of initializing tracker and restarting the measurements.

The method has the following signature:

| Method Signature | Language |
|------------------|----------|
| void reboot(); | Java |
| void reboot(); | C++ |
| void reboot(); | C# |

Note: In some cases, the connection to the tracker might be lost after the reboot. In this case the application must disconnect first and then try reconnecting to the tracker.

4.3.36 Follow Me Capable

This method is available from SDK version 5.0.2 for Vantage S/E or or Vantage S6/E6 trackers. This method can be used to check if tracker is Follow Me capable. If the method returns true, the tracker supports the feature. If false, feature is not supported.

The method has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| void isFollowMeCapable(); | Java |
| void isFollowMeCapable(); | C++ |
| void isFollowMeCapable(); | C# |

4.3.37 Follow Me On

This method is available from SDK version 5.0.2 for Vantage S/E or Vantage S6/E6 trackers . This method is used to check if Follow Me is on or not. If Follow Me is on, the method returns true else false.

The method has the following signature:

| Method Signature | Language |
|----------------------|----------|
| void isFollowMeOn(); | Java |
| void isFollowMeOn(); | C++ |
| void isFollowMeOn(); | C# |

4.3.38 Change Follow Me State

This method is available from SDK version 5.0.2 for Vantage S/E trackers or Vantage S6/E6 trackers. This method is used to turn the Follow Me feature on or off.

The method has the following signature:

| Method Signature | Language |
|--|----------|
| void changeFollowMeState(boolean state); | Java |
| void changeFollowMeState(bool state); | C++ |
| void changeFollowMeState(bool state); | C# |

Follow Me state change is persistent through power cycles.

This method can fail for the following reasons.

It will throw an unsupported feature exception if connected to a tracker that does not support the feature.

4.3.39 Get Follow Me Search Radius

This method is available from SDK version 5.0.2 for Vantage S/E trackers or Vantage S6/E6 trackers. This method is used to get the search radius of Follow Me.

The method has the following signature:

| Method Signature | Language |
|------------------------------------|----------|
| double getFollowMeSearchRadius (); | Java |
| double getFollowMeSearchRadius (); | C++ |
| double getFollowMeSearchRadius (); | C# |

The method will return the search radius in meters.

This method can fail for the following reasons.

It will throw an unsupported feature exception if connected to a tracker that does not support the feature.

4.3.40 Change Follow me Search Radius

This method is available from SDK version 5.0.2 for Vantage S/E trackers or Vantage S6/E6 trackers. This method is used to set the search radius of targets for Follow Me.

The method has the following signature:

| Method Signature | Language |
|---|----------|
| void changeFollowMeSearchRadius(double searchRadius); | Java |
| void changeFollowMeSearchRadius(double searchRadius); | C++ |
| void changeFollowMeSearchRadius(double searchRadius); | C# |

The unit of measure for the searchRadius parameter is meters. The valid range for the search radius is 0 to 20 meters.

This method can fail for the following reasons.

- It will throw an unsupported feature exception if connected to a tracker that does not support the feature.
- If the search radius specified is out of range.

4.3.41 SixDof Capable

SDK 5.1.0 and later only support this method. The method returns true if the tracker is capable else will return false.

| Method Signature | Language |
|--------------------------|----------|
| boolean sixDofCapable(); | Java |
| bool sixDofCapable(); | C++ |
| bool sixDofCapable(); | C# |

4.3.42 SixDof enabled

4.3.42.1.1 SixDof Enabled

This method will return a boolean value if the sixdof feature has been enabled or not. This feature has to be enabled to use the probe.

| Method Signature | Language |
|--------------------------|----------|
| boolean sixdofEnabled(); | Java |
| bool sixdofEnabled (); | C++ |
| bool sixdofEnabled (); | C# |

4.3.42.1.2 Change SixDof Enable State

SDK 5.1.0 and later only support this method. This method is used to change the sixdof enable state on and off. It has the following signature:

| Method Signature | Language |
|--|----------|
| void changeSixDofEnableState(boolean); | Java |
| void changeSixDofEnableState (bool); | C++ |
| void changeSixDofEnableState (bool); | C# |

Enabled state is persistent through power cycles.

This method can fail for the following reasons.

- It will throw a Tracker Exception if the tracker is not sixdof capable.
- It will throw unsupported feature exception if connected to a tracker that does not support the feature.

4.3.43 SixProbe Connected

SDK 5.1.0 and later only support this method. This method will return a boolean value if the sixprobe is connected or not.

| Method Signature | Language |
|----------------------------------|----------|
| boolean probeAdapterConnected(); | Java |
| bool probeAdapterConnected(); | C++ |
| bool probeAdapterConnected(); | C# |

4.3.44 SixProbe Name

SDK 5.1.0 and later only support this method. This method will return the serial number of the sixprobe connected to the tracker.

| Method Signature | Language |
|------------------------|----------|
| String probeAdapter(); | Java |
| char* probeAdapter(); | C++ |
| String probeAdapter(); | C# |

The method will throw an exception in the following cases.

- If tracker is not six dof capable.
- If there is no sixprobe connected to the tracker.

Please refer to section [4.3.37.11](#) for an example.

4.3.45 SixProbe Battery Charge Remaining

SDK 5.1.0 and later only support this method. The charge remaining for the sixprobe battery can be obtained using this method.

| Method Signature | Language |
|-----------------------------------|----------|
| int probebatteryChargeRemaining() | Java |
| int probebatteryChargeRemaining() | C++ |
| int probebatteryChargeRemaining() | C# |

The charge returned is between 0-100 percent.

The method can fail for the following reasons:

- It will throw a tracker exception
 - If the tracker is not sixdof capable
 - If not connected to the sixprobe.
 - If the sixprobe battery status is in error state.
- It will throw unsupported feature exception if connected to a tracker that does not support the feature.

4.3.46 Set Delay From Passive to Active Follow Me

SDK 5.1.4 and later only support this method. This method can be used to set the delay from passive to active Follow Me when a stable target is detected.

This delay is the wait time between when the stable target is detected and before the tracker moves to the target. This applies only if the yellow lights are steadily blinking when the tracker is in Follow Me mode.

| Method Signature | Language |
|---|----------|
| setDelayFromPassiveToActiveFollowMe(double delay) | Java |
| setDelayFromPassiveToActiveFollowMe(double delay) | C++ |
| setDelayFromPassiveToActiveFollowMe(double delay) | C# |

- The unit for delay is seconds.
- The method will throw a TrackerException if the delay value is not within 0 to 5 seconds.

4.3.47 Get Delay From Passive to Active Follow Me

SDK 5.1.4 and later only support this method. The method will return the delay in seconds set using the `setDelayFromPassiveToActiveFollowMe` when stable target is detected. The default value is 2.5 seconds.

| Method Signature | Language |
|--|----------|
| double getDelayFromPassiveToActiveFollowMe() | Java |
| double getDelayFromPassiveToActiveFollowMe() | C++ |
| double getDelayFromPassiveToActiveFollowMe() | C# |

4.3.48 SixProbe Battery State

SDK 5.1.5 and later only support this method. The state of the probe battery can be obtained using this method.

| Method Signature | Language |
|-------------------------|----------|
| int probeBatteryState() | Java |
| int probeBatteryState() | C++ |
| int probeBatteryState() | C# |

The following are the states of the probe battery that are returned.

0 = Accurate. Battery is in accurate state and the charge remaining value can be trusted.

1 = Inaccurate. Battery is in accurate state which means the charge value is still in indeterminate state.

2 = Error. Battery is in error and charge value cannot be trusted.

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|-------------------------------|----------|
| 0 | ProbeBatteryState.ACCURATE | Java |
| | ProbeBatteryState::ACCURATE | C++ |
| | ProbeBatteryState.ACCURATE | C# |
| 1 | ProbeBatteryState.INACCURATE | Java |
| | ProbeBatteryState::INACCURATE | C++ |
| | ProbeBatteryState.INACCURATE | C# |
| 2 | ProbeBatteryState.WARNING | Java |
| | ProbeBatteryState::WARNING | C++ |
| | ProbeBatteryState.WARNING | C# |

The method can fail for the following reasons:

- It will throw a tracker exception
 - If the tracker is not sixdof capable
 - If not connected to the probe.
- It will throw unsupported feature exception if connected to a tracker that does not support the feature.

4.3.49 SixProbe Battery State Change Event Supported

SDK 5.1.5 and later only support this method. This event is generated only if the firmware version is 2.6.1 and later only. So an API is provided to determine this.

| Method Signature | Language |
|---|----------|
| boolean probeBatteryStateChangeEventSupported() | Java |
| bool probeBatteryStateChangeEventSupported() | C++ |
| bool probeBatteryStateChangeEventSupported() | C# |

A true indicates that the event is supported. A false indicates that it is not supported.

4.3.50 Wireless State

Below are the methods related to wireless. These methods are available only from SDK 5.1.7 and later only.

4.3.50.1 Check if wireless state change supported

This method will return true if firmware supports changing wireless state else it will return false. It is recommended to use this API to check if this feature is supported before checking if wireless is on or changing wireless state. It has the following signature:

| Method Signature | Language |
|---|----------|
| boolean wirelessStateChangeSupported() | Java |
| bool wirelessStateChangeSupported () | C++ |
| bool wirelessStateChangeSupported () | C# |

4.3.50.2 Change Wireless State

The method to change the wireless state is used to turn the wireless on and off. It has the following signature:

| Method Signature | Language |
|---|----------|
| void changeWirelessState(boolean state) | Java |
| void changeWirelessState (bool state) | C++ |
| void changeWirelessState (bool state) | C# |

If true is passed for the state parameter, wireless will be turned on. If false is passed for the state parameter, wireless will be turned off.

This method can fail for the following reasons.

It will throw an unsupported feature exception if connected to a tracker that does not support the feature.

4.3.50.3 Get Current Wireless State

The wirelessOn method is used to determine the current state of the wireless. It has the following signature:

| Method Signature | Language |
|-----------------------|----------|
| boolean wirelessOn(); | Java |
| bool wirelessOn(); | C++ |
| bool wirelessOn(); | C# |

This method will return true if wireless is on and false if wireless is off.

This method can fail for the following reasons.

It will throw an unsupported feature exception if connected to a tracker that does not support the feature.

4.3.51 Check if capable of automatically reconnecting on communication glitch

SDK 5.1.8 and later only support this API.

The `autoReconnectCapable` method will return true if firmware supports automatically establishing communications when communication glitches occurs (more likely with WLAN).

Firmware version 2.9.1 and later is required and tracker should be Vantage S/E or Vantage S6/E6 type to return true. In all other cases, it returns false.

It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>boolean autoReconnectCapable()</code> | Java |
| <code>bool autoReconnectCapable ()</code> | C++ |
| <code>bool autoReconnectCapable ()</code> | C# |

4.3.52 Auto Compensation

The below methods are provide if tracker is capable of auto compensation and if required to run. SDK 5.1.8 and later only supports this API.

4.3.52.1 Check if capable of auto compensation

The `autoCompCapable` method will return true if capable. Firmware version 2.10.0 and later is required and tracker should be Vantage S/E or Vantage S6/E6 type. In all other cases, it returns false.

It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>boolean autoCompCapable()</code> | Java |
| <code>bool autoCompCapable () ()</code> | C++ |
| <code>bool autoCompCapable ()</code> | C# |

4.3.52.2 Check if auto compensation required

The `autoCompRequired` method will return true if tracker detects that a compensation is required to maintain accuracy. Firmware version 2.10.0 and later is required and tracker should be Vantage S/E or Vantage S6/E6 type. In all other cases, it returns false.

It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>boolean autoCompRequired()</code> | Java |
| <code>bool autoCompRequired() ()</code> | C++ |
| <code>bool autoCompRequired()</code> | C# |

If this method returns true, it is recommended to launch the compensation. Please see section [4.2.18](#) on how to launch a compensation when this method returns true.

4.3.53 Check if capable of generating tracker battery state change events

SDK 5.1.8 and later only support this API.

The `trackerBatteryEventCapable` returns true if capable of generating tracker battery change events. Please see section [5.9](#) on how to capture the event.

Firmware version 2.0 and later is required and tracker should be Vantage S/E or Vantage S6/E6 type. In all other cases, it returns false.
It has the following signature.

| Method Signature | Language |
|--------------------------------------|----------|
| boolean trackerBatteryEventCapable() | Java |
| bool trackerBatteryEventCapable () | C++ |
| bool trackerBatteryEventCapable () | C# |

4.3.54 Probe Management

SDK 5.1.0 and later only support the below methods which are available to manage sixdof probes. Using this API, users can create custom user interface to manage tips. Managing tips includes add, delete, modify and activate. Please refer to section [4.3.37.10](#) for an example on managing tips. A sixprobe must be connected to the tracker to add a new tip and to change (activate) the tip state. A tip may or may not be connected to the sixprobe to get all the available tips, modify or delete tips(s).

4.3.54.1 Available Tips

This method will return the list of all the available tips for this tracker.

| Method Signature | Language |
|--|----------|
| Probes[] availableProbes(); | Java |
| TrkDrvObjectArray* availableProbes (); | C++ |
| JObjectArray availableProbes (); | C# |

Below are the tip types returned.

- 1) Standard
- 2) Custom
- 3) Kinematic (Returned if probe model is SixProbe 2.0 and if SDK is 5.1.9 and later)

Standard tips are like the default tip templates that come with every six dof capable tracker. These can only be used to clone new tips. These template tips cannot be modified or deleted or activated.

Custom tips are the tips that are created by the user. These tips can be added or removed or activated to be used as the current tips type.

Kinematic tips are the tips that can be automatically mounted and removed. These type of tips can be used with SixProbe 2.0 only.

This method will throw exception in the following cases:

- If tracker is not six dof capable.
- If there is no sixprobe connected to the tracker

Please refer to section [4.3.37.11](#) for an example of this method.

Please refer to section [4.3.37.10](#) for an explanation of Probe class and its methods.

4.3.54.2 Probe

This will return the active tip that is currently being used.

| Method Signature | Language |
|------------------|----------|
| Probe probe(); | Java |
| Probe probe(); | C++ |

| Method Signature | Language |
|------------------|----------|
| Probe probe(); | C# |

The method will throw an exception in the following cases.

- If tracker is not six dof capable.
- If there is no sixprobe connected to the tracker
- If there is no active tip.

Please refer to section [4.3.37.11](#) for an example of this method.

Please refer to section [4.3.37.10](#) for an explanation of Probe class and its methods.

4.3.54.3 Change Tip

This method is used to change the tip to be currently active or inactive.

If a tip is made active:

If it is compensated, the tip vector is used in the measurements. If the leds on the tracker are blinking they will go solid. The tracker is now ready to take measurements.

If it is not compensated, the green leds on the tracker will start blinking indicating that the user has to compensate the tip. Measurements will not be valid. If the tip is compensated then the tip vector is applied, the green leds on the tracker go solid indicating that the tracker is ready to give valid measurements.

If tip is made inactive, the behavior will be same as when not compensated.

So, whenever user switches the tip, they need to activate the tip being used via software for Six probe 1.0 only, else it will affect the accuracy of the measurements. Six Probe 2.0 automatically handles tip changes, no need to activate manually.

| Method Signature | Language |
|--|----------|
| void changeProbe(Probe probe, boolean active); | Java |
| void changeProbe(Probe probe, bool active); | C++ |
| void changeProbe(Probe probe, bool active); | C# |

The probe parameter is the tip to be made active or inactive.

If the active parameter is set to true, tip will be made active. If set to false, tip will be made inactive

The method will throw an exception in the following cases.

- If tracker is not six dof capable.
- If there is no sixprobe connected to the tracker.
- If this tip does not exist (never added).
- If the tip being activated is standard type.
- It will throw unsupported feature exception when connected to SixProbe 2.0 because the tip is automatically activated/deactivated when the kinematic tip is attached/detached in SixProbe 2.0. Note that the probe tip needs to be manually activated by calling this method in case of SixProbe 1.0

Please refer to section [4.3.37.11](#) for an example of this method.

Please refer to section [4.3.37.10](#) for an explanation of Probe class and its methods.

4.3.54.4 Add Tip

This method is used to add a new custom tip.

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| Method Signature | Language |
|-----------------------------|----------|
| void addProbe(Probe probe); | Java |
| void addProbe(Probe probe); | C++ |
| void addProbe(Probe probe); | C# |

The probe parameter is the new tip to be added.

The method will throw an exception in the following cases.

- If tracker is not six dof capable.
- If there is no sixprobe connected to the tracker.
- If the user is adding a tip which already exists (same name).
- If user is adding a tip with insufficient parameters (name is empty).
- It will throw unsupported feature exception when connected to SixProbe 2.0, because the new tip automatically gets added to the stored tips after a successfully completing the tip compensation.

Please refer to section [4.3.37.11](#) for an example of this method.

Please refer to section [4.3.37.10](#) for an explanation of Probe class and its methods.

4.3.54.5 Modify Tip

This method is used to modify an existing tip.

| Method Signature | Language |
|--------------------------------|----------|
| void modifyProbe(Probe probe); | Java |
| void modifyProbe(Probe probe); | C++ |
| void modifyProbe(Probe probe); | C# |

The probe parameter is the tip which needs to be modified. The tip parameters that can be modified are name and diameter.

The method will throw an exception in the following cases.

- If tracker is not six dof capable.
- If this tip does not exist (never added).
- If the user is modifying a standard tip.

Please refer to section [4.3.37.11](#) for an example of this method.

Please refer to section [4.3.37.10](#) for an explanation of Probe class and its methods.

4.3.54.6 Remove Tip

This method is used to remove an existing tip.

| Method Signature | Language |
|--------------------------------|----------|
| void removeProbe(Probe probe); | Java |
| void removeProbe(Probe probe); | C++ |
| void removeProbe(Probe probe); | C# |

The probe parameter is the tip which needs to be removed.

The method will throw an exception in the following cases.

- If tracker is not six dof capable.

- If this tip does not exist (never added).
- If the user is removing a standard tip type.
- If the tip is active.

Please refer to section [4.3.37.11](#) for an example of this method.

Please refer to section [4.3.37.10](#) for an explanation of Probe class and its methods.

4.3.54.7 Probe

Probe class can be used to create new tips, add and modify them. The following parameters can be specified when creating a probe and modify them. Tip name, Diameter and Length.

Tip information is also returned as Probe objects. The object has several methods for obtaining information about the probe as below.

4.3.54.7.1 Get Name

This method returns the name of the tip.

| Method Signature | Language |
|-------------------|----------|
| String getName(); | Java |
| char* getName(); | C++ |
| String getName(); | C# |

4.3.54.7.2 Get SixProbe Name

This method returns the name of the sixprobe this tip is associated with.

| Method Signature | Language |
|-------------------------------|----------|
| String getProbeAdapterName(); | Java |
| char* getProbeAdapterName(); | C++ |
| String getProbeAdapterName(); | C# |

4.3.54.7.3 Get Diameter

This method returns the diameter of the tip that has been set to in meters.

| Method Signature | Language |
|-----------------------|----------|
| double getDiameter(); | Java |
| double getDiameter(); | C++ |
| double getDiameter(); | C# |

4.3.54.7.4 Get Length

This method returns the length of the tip that has been set to in meters.

| Method Signature | Language |
|---------------------|----------|
| double getLength(); | Java |
| double getLength(); | C++ |
| double getLength(); | C# |

4.3.54.7.5 *Get X*

This method return the x value of the tip vector in meters.

| Method Signature | Language |
|------------------|----------|
| double getX(); | Java |
| double getX(); | C++ |
| double getX(); | C# |

4.3.54.7.6 *Get Y*

This method will return the y value of the tip vector in meters.

| Method Signature | Language |
|------------------|----------|
| double getY(); | Java |
| double getY(); | C++ |
| double getY(); | C# |

4.3.54.7.7 *Get Z*

This method will return the z value of the tip vector in meters.

| Method Signature | Language |
|------------------|----------|
| double getZ(); | Java |
| double getZ(); | C++ |
| double getZ(); | C# |

4.3.54.7.8 *Get Calibration Time*

This method will return the timestamp of the calibration in seconds

| Method Signature | Language |
|---------------------------------|----------|
| double getDateOfCalibration(); | Java |
| double getDateOfCalibration (); | C++ |
| double getDateOfCalibration (); | C# |

4.3.54.7.9 *Get Calibration Temp*

This method will return the temperature at the time of the calibration in degC.

| Method Signature | Language |
|--------------------|----------|
| double getTemp(); | Java |
| double getTemp (); | C++ |
| double getTemp (); | C# |

4.3.54.7.10 *Get Type*

This method will return whether the type of the tip is a Standard type or Custom type

| Method Signature | Language |
|------------------|----------|
| int getType(); | Java |
| int getType (); | C++ |
| int getType (); | C# |

It will return the following values.

0 – Standard (Tips of this type can be cloned only)

1 – Custom (These type of tips are created by users and can be modified or removed)

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|------------------|----------|
| 0 | Probe.STANDARD | Java |
| | Probe::STANDARD | C++ |
| | Probe.STANDARD | C# |
| 1 | Probe.CUSTOM | Java |
| | Probe::CUSTOM | C++ |
| | Probe.CUSTOM | C# |
| 2 | Probe.KINEMATIC | Java |
| | Probe::KINEMATIC | C++ |
| | Probe. KINEMATIC | C# |

4.3.54.7.11 *Get Active*

This method will return a boolean value. It will return true if tip is active else will return false.

| Method Signature | Language |
|-------------------|----------|
| boolean active(); | Java |
| bool active(); | C++ |
| bool active(); | C# |

4.3.54.7.12 *Get Compensated*

This method will return a boolean value. It will return true if tip is compensated else will return false.

| Method Signature | Language |
|------------------------|----------|
| boolean compensated(); | Java |
| bool compensated(); | C++ |
| bool compensated(); | C# |

4.3.54.7.13 *Check if error is supported*

This method will return a boolean value. It will return true if error is supported for this tip type else will return false.

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| Method Signature | Language |
|-----------------------------|----------|
| boolean isErrorSupported(); | Java |
| bool isErrorSupported(); | C++ |
| bool isErrorSupported(); | C# |

4.3.54.7.14 Get Error

This method will return an error value.

| Method Signature | Language |
|---------------------|----------|
| boolean getError(); | Java |
| bool getError(); | C++ |
| bool getError(); | C# |

4.3.54.7.15 Get Serial Number

This method will return the serial number of the kinematic tip if the tip is kinematic type.

| Method Signature | Language |
|----------------------------|----------|
| boolean getSerialNumber(); | Java |
| bool getSerialNumber(); | C++ |
| bool getSerialNumber(); | C# |

4.3.54.7.16 Set Name

This is the method is used to modify the name of the tip

| Method Signature | Language |
|-----------------------|----------|
| void setName(String); | Java |
| void setName(char*); | C++ |
| void setName(String); | C# |

4.3.54.7.17 Set Diameter

This method is to modify the diameter of the tip. Diameter has to be in meters.

| Method Signature | Language |
|---------------------------|----------|
| void setDiameter(double); | Java |
| void setDiameter(double); | C++ |
| void setDiameter(double); | C# |

4.3.54.7.18 Set Length

This method is to modify the length of the tip. Length has to be in meters.

| Method Signature | Language |
|-------------------------|----------|
| void setLength(double); | Java |
| void setLength(double); | C++ |
| void setLength(double); | C# |

4.3.54.8 Example

4.3.54.8.1 Java

```
//get all the probes
Probes[] probes = trk.availableProbes();
for(int i = 0; i < probes.length; i++){
    //get probe information
}

//To add a new probe
String adapterName = trk.probeAdapter();
Probe p = new Probe("test", adapterName, 1.0, 5.0); //"test" is the name,
1.0 is diameter of the probe and 5.0 is the length of the probe
trk.addProbe(p);

//To remove probe
trk.removeProbe(probes[0]);

//To change probe and activate
trk.changeProbe(probes[0], true);

//To modify probe
probes[0].setName("temp");
trk.modifyProbe(probes[0]); //This will modify the name of probe from
test to temp.
```

4.3.54.8.2 C++

```
//get all the probes
TrkDrvObjectArray* probes = trk->availableProbes();
int num = probes->numElements();
for(int i = 0; i < num; i++){
    Probe* probe = (Probe*)pDataArray->getElement(i);
    //get probe information
}

//To add a new probe
char* adapterName = trk->probeAdapter();
Probe* p = new Probe("test", adapterName, 1.0, 5.0); //"test" is the name,
1.0 is diameter of the probe and 5.0 is the length of the probe
trk->addProbe(p);

//To remove probe
Probe* p = (Probe*)pDataArray->getElement(0);
trk.removeProbe(p);

//To change probe and activate
Probe* p = (Probe*)pDataArray->getElement(0);
trk.changeProbe(p, true);

//To modify probe
Probe* p = (Probe*)pDataArray->getElement(0);
p->setName("temp");
trk.modifyProbe(p); //This will modify the name of probe from test to
temp.
```

4.3.54.8.3 C#

```
//get all the probes
JSONObject probes = trk.availableProbes();
probes = new Probe[array.numElements];
for(int i = 0; i < array.numElements; i++){
    probes[i] = (Probe)array.getElement(i);
    //get probe information
}

//To add a new probe
String adapterName = trk.probeAdapter();
Probe p = new Probe("test", adapterName, 1.0, 5.0); //"test" is the name,
1.0 is diameter of the probe and 5.0 is the length of the probe
trk.addProbe(p);

//To remove probe
Probe p = (Probe)array.getElement(0);
trk.removeProbe(p);

//To change probe and activate
Probe p = (Probe)array.getElement(0);
trk.changeProbe(p, true);

//To modify probe
Probe p = (Probe)array.getElement(0);
p.setName("temp");
trk.modifyProbe(p); //This will modify the name of probe from test to
temp.
```

4.3.54.9 Is Six Probe 2.0 capable

SDK 5.1.9 and later only support this method. Firmware version 2.11 and later only support SixProbe 2.0. The method returns true if the tracker is capable else will return false.

| Method Signature | Language |
|-----------------------------|----------|
| boolean sixProbe2Capable(); | Java |
| bool sixProbe2Capable(); | C++ |
| bool sixProbe2Capable(); | C# |

4.3.54.10 Get sixprobe model type

SDK 5.1.9 and later only support this method. The method returns the model type of the sixprobe the tracker is currently connected to.

| Method Signature | Language |
|-----------------------|----------|
| int probeModelType(); | Java |
| int probeModelType(); | C++ |
| int probeModelType(); | C# |

If Firmware version 2.11 and later the values returned are 0 if not connected to any probe or if tracker is not sixdof capable, 1 if connected to SixProbe 1.0 and 2 if connected to SixProbe 2.0.

If Firmware version is below 2.11, the values returned are 0 if not connected to any probe or if tracker is not sixdof capable, 1 if connected to SixProbe 1.0 or if connected to SixProbe 2.0. This is because firmware version 2.11 is required for SixProbe 2.0.

| Value | Reference | Language |
|-------|-------------------------------|----------|
| 0 | ProbeModelType.NONE | Java |
| | ProbeModelType::NONE | C++ |
| | ProbeModelType.NONE | C# |
| 1 | ProbeModelType.SIX_PROBE_1_0 | Java |
| | ProbeModelType::SIX_PROBE_1_0 | C++ |
| | ProbeModelType.SIX_PROBE_1_0 | C# |
| 2 | ProbeModelType.SIX_PROBE_2_0 | Java |
| | ProbeModelType::SIX_PROBE_2_0 | C++ |
| | ProbeModelType.SIX_PROBE_2_0 | C# |

4.3.54.11 Is kinematic tip connected to Six probe 2.0

SDK 5.1.9 and later only support this method. The method returns true if a kinematic tip is connected to SixProbe 2.0 else will return false.

| Method Signature | Language |
|---|----------|
| boolean kinematicAdapterConnectedToSixProbe2(); | Java |
| bool kinematicAdapterConnectedToSixProbe2 (); | C++ |
| bool kinematicAdapterConnectedToSixProbe2 (); | C# |

4.4 Data Collection

The data collection methods are used to collect target data from the tracker. There are two mechanisms for collecting data from the tracker:

- Foreground measurement
- Background measurement

Both mechanisms can collect data in various ways, and both can generate events.

4.4.1 Foreground Measurement

When a foreground measurement is in progress, control methods and configuration / state methods cannot be called until the measurement is stopped. In addition, when a foreground measurement is in progress, the red light on the tracker is used.

4.4.1.1 Start

A foreground measurement is started using the startMeasurePoint method that has the following signature:

| Method Signature | Language |
|---|----------|
| void startMeasurePoint(MeasureCfg cfg); | Java |
| void startMeasurePoint(MeasureCfg *cfg); | C++ |
| void startMeasurePoint(MeasureCfg cfg); | C# |

If blocking is enabled, this method will return after the measurement has started. If blocking is disabled, this method will return after the command has been submitted to the tracker. A command complete event will be generated to indicate that the measurement is started.

The `cfg` parameter is a `MeasureCfg` object that defines how data should be collected. This class will be explained in more detail in section 4.4.4.

Restrictions:

- A connection must be established with a tracker, otherwise `NotConnectedException` is thrown.
- A command may not already be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- A foreground measurement cannot already be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise `InterfaceBusyException` is thrown.

The following events are typically generated:

- **Command Complete** – Indicates that the command finished and in also indicates whether it succeeded or failed.
- **Busy** – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy and that the measure state is busy. Note: the first event is not generated for each method if exclusive access is granted to a thread.
- **MeasureDataAvailable** – Generated for data that is received. The number of events per measurements received is control by the data event rate property. See section 3.3.
- **Status** – Generated if asynchronous messages are started. Typical codes encountered are as follows: `MEASURING`.

4.4.1.2 Read Data

Data from a foreground measurement is read using the `readMeasurePointData` method that has the following signature:

| Method Signature | Language |
|---|----------|
| <code>MeasurePointData[] readMeasurePointData(int numRecs);</code> | Java |
| <code>TrkDrvObjectArray * readMeasurePointData(int numRecs);</code> | C++ |
| <code>ObjectArray readMeasurePointData(int numRecs);</code> | C# |

The following can be used to obtain one object at a time:

| Method Signature | Language |
|---|----------|
| <code>MeasurePointData readMeasurePointData();</code> | Java |
| <code>MeasurePointData * readMeasurePointData();</code> | C++ |
| <code>MeasurePointData readMeasurePointData();</code> | C# |

If the `readMeasurePointData` method is called with no parameters, one `MeasureData` object will be returned. If a number of records is specified, that number of `MeasureData` objects is returned in an array.

If blocking is enabled, the method will not return until the specified number of records has been received. If blocking is disabled, the method will return if the number of records has been received. It will throw an exception if the specified number of records has not been received.

This method will also throw an exception if a measurement has not been started.

See section 4.4.3 for an explanation of the MeasurePointData class.

4.4.1.3 Stop

The stopMeasurePoint method stops a point measurement that was previously started by the startMeasurePoint method. It has the following signature:

| Method Signature | Language |
|--------------------------|----------|
| void stopMeasurePoint(); | Java |
| void stopMeasurePoint(); | C++ |
| void stopMeasurePoint(); | C# |

When a measurement is stopped, any buffered data is lost.

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.

The following events are typically generated:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy and that the measure state is not busy. Note: the first event is not generated for each method if exclusive access is granted to a thread.
- Status – Generated if asynchronous messages are started. Typical codes encountered are as follows: MEASURING.

4.4.2 Background Measurement

When a background measurement is in progress, control methods, configuration / state methods, and foreground measurement methods can be called. The red light on the tracker will not indicate that a measurement is in progress for a background measurement.

4.4.2.1 Start

A background measurement is started using the startMeasurePoint method that has the following signature:

| Method Signature | Language |
|---|----------|
| void startBkndMeasurePoint(MeasureCfg cfg); | Java |
| void startBkndMeasurePoint(MeasureCfg *cfg); | C++ |
| void startBkndMeasurePoint(MeasureCfg cfg); | C# |

If blocking is enabled, this method will return after the measurement has started. If blocking is disabled, this method will return after the command has been submitted to the tracker. A command complete event will be generated to indicate that the measurement is started.

The cfg parameter is a MeasureCfg object that defines how data should be collected. This class will be explained in more detail in section 4.4.4.

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.

- A command may not already be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- A background measurement cannot already be in progress when the method is called, otherwise `InterfaceBusyException` is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise `InterfaceBusyException` is thrown.

The following events are typically generated:

- **Command Complete** – Indicates that the command finished and in also indicates whether it succeeded or failed.
- **Busy** – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy and that the measure state is busy. Note: the first event is not generated for each method if exclusive access is granted to a thread.
- **MeasureDataAvailable** – Generated for data that is received. The number of events per measurements received is control by the data event rate property. See section 3.3.

4.4.2.2 Read Data

Data from a background measurement is read using the `readBkndMeasurePointData` method that has the following signature:

| Method Signature | Language |
|---|----------|
| <code>MeasurePointData[] readBkndMeasurePointData(int numRecs);</code> | Java |
| <code>TrkDrvObjectArray * readBkndMeasurePointData(int numRecs);</code> | C++ |
| <code>JsonObjectArray readBkndMeasurePointData(int numRecs);</code> | C# |

The following can be used to obtain one object at a time:

| Method Signature | Language |
|---|----------|
| <code>MeasurePointData readBkndMeasurePointData();</code> | Java |
| <code>MeasurePointData * readBkndMeasurePointData();</code> | C++ |
| <code>JsonObjectArray readBkndMeasurePointData();</code> | C# |

If the `readBkndMeasurePointData` method is called with no parameters, one `MeasureData` object will be returned. If a number of records is specified, that number of `MeasureData` objects is returned in an array.

If blocking is enabled, the method will not return until the specified number of records has been received. If blocking is disabled, the method will return if the number of records has been received. It will throw an exception if the specified number of records has not been received.

This method will also throw an exception if a background measurement has not been started.

See section 4.4.3 for an explanation of the `MeasurePointData` class.

4.4.2.3 Stop

The `stopBkndMeasurePoint` method stops a point measurement that was previously started by the `startBkndMeasurePoint` method. It has the following signature:

| Method Signature | Language |
|---|----------|
| <code>void stopBkndMeasurePoint();</code> | Java |

| Method Signature | Language |
|------------------------------|----------|
| void stopBkndMeasurePoint(); | C++ |
| void stopBkndMeasurePoint(); | C# |

When a background measurement is stopped, any buffered data is lost.

Restrictions:

- A connection must be established with a tracker, otherwise `NotConnectedException` is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise `InterfaceBusyException` is thrown.

The following events are typically generated:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy and that the measure state is not busy. Note: the first event is not generated for each method if exclusive access is granted to a thread.

4.4.3 Measurement Point Data

Point data is returned as `MeasurePointData` objects. The object has several methods for obtaining information about the target.

4.4.3.1 Status

The status method returns information concerning the accuracy of the point data. It has the following signature:

| Method Signature | Language |
|------------------|----------|
| int status(); | Java |
| int status(); | C++ |
| int status(); | C# |

This method returns the following values:

- 0 = Data accurate. This indicates that the point data is valid for this observation. `MeasurePointData.DATA_ACCURATE` can be used in Java applications to test for this status.
- 1 = Data inaccurate. This indicates that the tracker has detected a target in the beam path, but the information about the target is inaccurate. This may be due to a bad encoder reading or an inaccurate radial distance due to a beam break. `MeasurePointData.DATA_INACCURATE` can be used in Java applications to test for this status.
- 2 = Data Error. This indicates that the tracker did not detect a target in the beam path when the observation was taken. `MeasurePointData.DATA_ERROR` can be used in Java applications to test for this status.

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|---|----------|
| 0 | <code>MeasurePointData.DATA_ACCURATE</code> | Java |
| | <code>MeasurePointData::DATA_ACCURATE</code> | C++ |
| | <code>MeasurePointData.DATA_ACCURATE</code> | C# |
| 1 | <code>MeasurePointData.DATA_INACCURATE</code> | Java |

| Value | Reference | Language |
|-------|-----------------------------------|----------|
| | MeasurePointData::DATA_INACCURATE | C++ |
| | MeasurePointData.DATA_INACCURATE | C# |
| 2 | MeasurePointData.DATA_ERROR | Java |
| | MeasurePointData::DATA_ERROR | C++ |
| | MeasurePointData.DATA_ERROR | C# |

If tracker is sixdof capable and target type is set to sixdof, the status returned includes the status of the probe orientation angles (rotation A, rotation B and rotation C).

In all the other cases, the status returned contains status of the azimuth, zenith and distance values.

4.4.3.2 Time

The time method returns a time stamp that tracker assigned to the data. It has the following signature:

| Method Signature | Language |
|------------------|----------|
| double time(); | Java |
| double time(); | C++ |
| double time(); | C# |

The unit of measure for the time stamp is in seconds. It represents the number of seconds that have elapsed since January 1, 1970 00:00:00 GMT.

4.4.3.3 Position

The position of the target is obtained using the azimuth, zenith, and distance methods. They have the following signatures:

| Method Signature | Language |
|-------------------|----------|
| double azimuth(); | Java |
| double azimuth(); | C++ |
| double azimuth(); | C# |

| Method Signature | Language |
|------------------|----------|
| double zenith(); | Java |
| double zenith(); | C++ |
| double zenith(); | C# |

| Method Signature | Language |
|--------------------|----------|
| double distance(); | Java |
| double distance(); | C++ |
| double distance(); | C# |

The azimuth and zenith methods return the two angles for the target location. The unit of measure is radians. The values are normalized to the range of $\pm\pi$.

The distance method returns the radial distance to the target. The unit of measure is meters.

Note, if the status indicates that the data is in error, all three of these methods will return 0.0.

4.4.3.4 Data Type

This method is available from SDK 5.1.0 version. This method returns true if the position of the target location is of the probe tip. This is returned only if tracker is sixdof capable and target type is set to sixdof target type. In all other cases this method returns false as the position of target location is the center of the SMR.

The method has the following signature.

| Method Signature | Language |
|-----------------------|----------|
| boolean sixDofData(); | Java |
| bool sixDofData (); | C++ |
| bool sixDofData (); | C# |

4.4.3.5 Probe orientation

From SDK version 5.1.0, MeasurePointData consists of the methods to obtain the probe orientation. The methods are rotationA, rotationB and rotationC.

They have the following signatures:

| Method Signature | Language |
|----------------------|----------|
| double rotationA(); | Java |
| double rotationA (); | C++ |
| double rotationA (); | C# |

| Method Signature | Language |
|---------------------|----------|
| double rotationB(); | Java |
| double rotationB(); | C++ |
| double rotationB(); | C# |

| Method Signature | Language |
|---------------------|----------|
| double rotationC(); | Java |
| double rotationC(); | C++ |
| double rotationC(); | C# |

The rotation A, B and C angles returned are the Euler angles in radians.

These methods will fail for the following reason:

- If the tracker is not sixdof capable, an Unsupported feature exception will be thrown.

4.4.3.6 Statistics

If statistics information is requested along with the measurement, it is obtained by calling the statistics method. See section to see how to configure a measurement with statistics. The statistics method has the following signature:

| Method Signature | Language |
|-----------------------------------|----------|
| MeasurePointStats statistics(); | Java |
| MeasurePointStats * statistics(); | C++ |
| MeasurePointStats statistics(); | C# |

The object returned is a class that is derived from MeasurePointStats.

4.4.3.6.1 Standard Deviation

Standard deviation information is contained in the MeasurePointStdDev class. The standard deviation information is retrieved by calling the following methods:

| Method Signature | Language |
|-------------------------|----------|
| double azimuthStdDev(); | Java |
| double azimuthStdDev(); | C++ |
| double azimuthStdDev(); | C# |

| Method Signature | Language |
|------------------------|----------|
| double zenithStdDev(); | Java |
| double zenithStdDev(); | C++ |
| double zenithStdDev(); | C# |

| Method Signature | Language |
|--------------------------|----------|
| double distanceStdDev(); | Java |
| double distanceStdDev(); | C++ |
| double distanceStdDev(); | C# |

The azimuthStdDev method returns the standard deviation of the azimuth axis measurement in radians. The zenithStdDev method returns the standard deviation of the zenith axis measurement in radians. The distanceStdDev method returns the standard deviation of the distance measurement in meters.

4.4.4 Measurement Configuration

The measurement configuration for a measurement is specified when the measurement is started using the MeasureCfg class.

The constructor for this class is as follows:

| Method Signature | Language |
|--|----------|
| MeasureCfg(int samplesPerObservation, Filter filter, StartTrigger startTrigger, ContinueTrigger continueTrigger); | Java |
| MeasureCfg(int samplesPerObservation, Filter *filter, StartTrigger *startTrigger, ContinueTrigger *continueTrigger); | C++ |
| MeasureCfg(int samplesPerObservation, Filter filter, StartTrigger startTrigger, ContinueTrigger continueTrigger); | C# |

4.4.4.1 Samples Per Observation

The number of samples per observation specifies how many samples the tracker will take before processing the data. The samples are taken at the sample rate of the tracker.

For Vantage S/E or Vantage S6/E6 tracker, the number of samples per observation should be set to 1 if the filter is set to Interpolation filter and continue trigger set to External continue trigger.

4.4.4.2 Filter

The filter for the samples passing an object that was created from one of the Filter subclasses. The Filter class is abstract, so a Filter object cannot be created. The following objects can be created:

- AverageFilter
- StdDevFilter
- InterpolationFilter (Vantage S/E or Vantage S6/E6 only)

An AverageFilter object indicates that all of the samples should be average together to return one MeasurePointObject. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| AverageFilter(); | Java |
| AverageFilter(); | C++ |
| AverageFilter(); | C# |

A StdDevFilter object indicates that all of the samples should be averaged together to return one MeasurePoint. In addition, the standard deviation information should be included. The statistics information is obtained by calling the statistics method in the MeasurePointData object that holds the measurement data.

The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| StdDevFilter(); | Java |
| StdDevFilter(); | C++ |
| StdDevFilter(); | C# |

The Interpolation filter is available only for Vantage S/E or Vantage S6/E6 tracker. An InterpolationFilter object indicates to generate the sample by interpolating the data(generated in micro seconds) before and after the trigger to return one Measure Point. If the measurement configuration is using this filter, then the trigger should be set to External Continue Trigger only. In any other case an exception will be thrown when measurements are started.

The constructor has the following signature:

| Constructor Signature | Language |
|------------------------|----------|
| InterpolationFilter(); | Java |
| InterpolationFilter(); | C++ |
| InterpolationFilter(); | C# |

4.4.4.3 Start Trigger

The start trigger specifies when the first observation should be taken. The start trigger is specified by passing an object that was created from one of the StartTrigger subclasses. The StartTrigger class is abstract, so a StartTrigger object cannot be created. The following objects can be created:

- NullStartTrigger
- ExternalStartTrigger

A NullStartTrigger object indicates that the first observation should be taken as soon as possible. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------|----------|
| NullStartTrigger (); | Java |

| Constructor Signature | Language |
|-----------------------|----------|
| NullStartTrigger (); | C++ |
| NullStartTrigger (); | C# |

An ExternalStartTrigger object indicates that the tracker should wait for the trigger on its trigger input connection. The constructor has the following signature:

| Constructor Signature | Language |
|--------------------------|----------|
| ExternalStartTrigger (); | Java |
| ExternalStartTrigger (); | C++ |
| ExternalStartTrigger (); | C# |

4.4.4.4 Continue Trigger

The continue trigger specifies when all observations after the first observation should be take. The continue trigger is specified by passing an object that was created from one of the ContinueTrigger subclasses. The ContinueTrigger class is abstract, so a ContinueTrigger object cannot be created. The following objects can be created:

- NullContinueTrigger
- ExternalContinueTrigger
- IntervalTrigger
- DistanceTrigger

A NullContinueTrigger object indicates that subsequent observations should be taken as fast as possible. The constructor has the following signature:

| Constructor Signature | Language |
|-------------------------|----------|
| NullContinueTrigger (); | Java |
| NullContinueTrigger (); | C++ |
| NullContinueTrigger (); | C# |

An ExternalContinueTrigger object indicates that the tracker should wait for the trigger on its trigger input connection. The constructor has the following signature:

| Constructor Signature | Language |
|-----------------------------|----------|
| ExternalContinueTrigger (); | Java |
| ExternalContinueTrigger (); | C++ |
| ExternalContinueTrigger (); | C# |

For Vantage S/E tracker or Vantage S6/E6, this trigger can be used with interpolation filter only. So, if the meurent configuration is using this trigger, then the filter should be set to Interpolation filter. In any other case, an exception will be thown when measurements are started.

An IntervalTrigger object indicates that subsequent observations should be taken a given rate. The rate is specified in seconds when the IntervalTrigger object is created. The constructor has the following signature:

| Constructor Signature | Language |
|--------------------------------------|----------|
| IntervalTrigger (double interval); | Java |
| IntervalTrigger (double interval); | C++ |

| Constructor Signature | Language |
|--------------------------------------|----------|
| IntervalTrigger (double interval); | C# |

A DistanceTrigger object indicates that subsequent observations should be taken when the target moves a given distance. The distance is specified in meters when the DistanceTrigger object is created.

The constructor has the following signature:

| Constructor Signature | Language |
|--------------------------------------|----------|
| DistanceTrigger (double distance); | Java |
| DistanceTrigger (double distance); | C++ |
| DistanceTrigger (double distance); | C# |

4.5 Tracker Application

There are applications that can be started through the interface. These applications provide tracker specific functionality. To run these applications, a connection must be established with a tracker. The following methods are associated with tracker applications:

- availableApplications
- startedApplications
- startApplicationFrame
- startApplicationFrameAlwaysOnTop
- stopApplicationFrame
- modifyApplication
- applicationExitCode
- releaseLock

Tracker applications will use same Tracker object that the main application uses. If a tracker application is performing an operation with the tracker, and the main application attempts to use the interface, an exception will be thrown. The busy event can be used to avoid this problem by disabling all tracker-related items in the user interface when a busy event is received.

4.5.1 Available Applications

The availableApplications method returns the applications that can be run through the interface. It has the following signature:

| Method Signature | Language |
|--|----------|
| String [] availableApplications(); | Java |
| TrkDrvObjectArray * availableApplications(); | C++ |
| ObjectArray availableApplications(); | C# |

The strings returned are the names of the applications that can be started. The name is used to actually start and stop the application.

Possible applications are as follows:

- Tracker Pad
- Startup Checks
- Firmware Loader

- Closure

4.5.2 Started Applications

The startedApplications method returns the names of the applications that are currently running. It has the following signature:

| Method Signature | Language |
|---|----------|
| String [] startedApplications(); | Java |
| TrkDrvObjectArray *startedApplications(); | C++ |
| ObjectArray startedApplications(); | C# |

Null will be returned if no applications are currently running.

4.5.3 Start Application Frame

An application is started in its own window frame by calling the method startApplicationFrame. This method has the following signature:

| Method Signature | Language |
|--|----------|
| void startApplicationFrame(String appName, String paramStr); | Java |
| void startApplicationFrame(char appName[], char paramStr[]); | C++ |
| void startApplicationFrame(String appName, String paramStr); | C# |

The appName parameter is the name of the application to be started. It should be identical to one of the strings returned by the availableApplications method. Note: availableApplications should be called before attempting to start the application since the application may not be available for all trackers.

The paramStr parameter is a string to be passed to the application. This string represents parameters that are understood by the application itself. If no parameters are necessary, an empty string ("") should be passed.

Restriction

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.
- The application being started must not already be running.
- The tracker must support the requested application.

All of the following events are generated when the control methods are called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.
- Application Event – Indicates that the application started successfully.

4.5.4 Start Application That Is Always Top Window

The startApplicationFrameAlwaysOnTop method is the same as the startApplicationFrame method except for one key difference. Tracker applications started with startApplicationFrameAlwaysOnTop will cause the tracker application to always be the top window on the display.

| Method Signature | Language |
|---|----------|
| void startApplicationFrameAlwaysOnTop(String appName, String paramStr); | Java |
| void startApplicationFrameAlwaysOnTop(char appName[], char paramStr[]); | C++ |
| void startApplicationFrameAlwaysOnTop(String appName, String paramStr); | C# |

4.5.5 Stop Application Frame

A started application is stopped by calling the stopApplicationFrame method. This method has the following signature:

| Method Signature | Language |
|---|----------|
| void stopApplicationFrame(String appName); | Java |
| void stopApplicationFrame(char appName[]); | C++ |
| void stopApplicationFrame(String appName); | C# |

The appName is the name of the application to be stopped. It should be the same name used to start the application.

No error occurs if stopApplicationFrame is called for an application that is not running.

Restriction

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.

All of the following events are generated when the control methods are called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.
- Application Event – Indicates that the application stopped.

4.5.6 Modify Application

A started application can have its parameters modified by calling the method modifyApplication. This method has the following signature:

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| Method Signature | Language |
|---|----------|
| void modifyApplication(String appName, String paramStr); | Java |
| void modifyApplication(char appName[], char paramStr[]); | C++ |
| void modifyApplication(String appName, String paramStr); | C# |

The appName parameter is the name of the application that has already been started. It should be identical to one of the strings returned by the availableApplications method.

The paramStr parameter is a string to be passed to the application. This string represents parameters that are understood by the application itself. If no parameters are necessary, an empty string (“”) should be passed.

Restriction

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- The specified application must already be started.

4.5.7 Get application exit status

Currently only TrackerPad application supports this and again only when the CameraDrive is opened directly. The method to get the exit status has the following signature:

| Method Signature | Language |
|--|----------|
| int applicationExitCode (String appName); | Java |
| int applicationExitCode (char appName[]); | C++ |
| int applicationExitCode (String appName); | C# |

The appName parameter is the name of the application that has already been started. It should be identical to one of the strings returned by the availableApplications method.

The possible return values are 0 for closed, 1 for Canceled, 2 for running, 3 for never run since the driver object is instantiated, 4 for not supported.

Currently “TrackerPad” is the only application that returns meaningful exit status. Other applications simply return the not supported exit code.

From 4.3.0 SDK revision, No UI Quick Compensation and No UI Angular Accuracy Checks also return the application exit status. The names to supply as parameters to get the exit status are “QuickComp” for No UI Quick Compensation and “AngularAccuracy” for No UI Angular Accuracy Checks.

4.5.8 Release the lock on the tracker

There is a chance that same application belonging to different driver objects need to control the tracker. For example there can be multiple CameraDrive applications that talk to the same tracker. But currently only the first application that controls the tracker get the exclusive control on the tracker and the second application would gets an exception(DeviceLockedException). So a new method is added to facilitate this. It has the following signature:

| Method Signature | Language |
|----------------------|----------|
| void releaseLock (); | Java |
| void releaseLock (); | C++ |
| void releaseLock (); | C# |

So when driver object A wants to relinquish the lock, it calls this method thus allowing driver object B to control the tracker. When driver object B again releases the lock, A can get the control on the tracker. Thus this method allows co-operating applications to control the tracker alternately.

4.6 Tracker Asynchronous Messages

The tracker generates asynchronous messages to inform applications about state changes or errors that may not be related to a specific command. The interface generates events when these messages are received.

There are four types of messages:

- Diagnostic – Indicates an error condition that occurred in the tracker that is a one-time event. The error condition may or may not persist depending on the type of failure.
- Status – Indicates that the state of one of the status indicators has changed in tracker. This type of messages has a concept of either being on or off.
- Change – Indicates that a specific item has changed. The application must call another method to determine the state.
- Alarm – Indicates that the state has changed for an alarm configured for one of the temperature sensors.

The blocking flag affects all methods associated with the asynchronous messages. In addition, calling any of these methods will cause the interface to show that it is busy executing a command until the command completes. None of these methods can be called while a measurement is in progress.

The following methods are associated with the asynchronous messages:

- startAsync
- stopAsync
- diagnosticHistory
- readBufferedDiagnosticHistory
- statusHistory
- readBufferedStatusHistory
- alarmHistory
- readBufferedAlarmHistory

4.6.1 Start Asynchronous Messages

The tracker does not send asynchronous events unless instructed. To have the tracker start sending asynchronous messages, the startAsync method is used. It has the following signature:

| Method Signature | Language |
|--------------------|----------|
| void startAsync(); | Java |
| void startAsync(); | C++ |
| void startAsync(); | C# |

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.
- Asynchronous messages must not already be started, otherwise AsyncAlreadyStartedException will be thrown.

All of the following events are generated when the startAsync method is called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

4.6.2 Stop Asynchronous Messages

To stop the tracker from sending asynchronous messages, the stopAsync method is used. It has the following signature:

| Method Signature | Language |
|-------------------|----------|
| void stopAsync(); | Java |
| void stopAsync(); | C++ |
| void stopAsync(); | C# |

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.

All of the following events are generated when the stopAsync method are called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

4.6.3 Diagnostic History

To obtain a list of all diagnostic messages generated by the tracker, the method diagnosticHistory() is used. It can be called regardless of whether asynchronous messages have been started or not. This method has the following signature:

| Method Signature | Language |
|--|----------|
| Diagnostic[] diagnosticHistory(); | Java |
| TrkDrvObjectArray * diagnosticHistory(); | C++ |
| JobObjectArray diagnosticHistory(); | C# |

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.

- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.
- This API will throw NoDataAvailableException for Vantage S/E or Vantage S6/E6 tracker.

All of the following events are generated when the diagnosticHistory method is called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

If blocking is enabled, it returns an array of Diagnostic objects representing all messages in the tracker's diagnostic history list. If blocking is disabled the method returns null. The message can then be obtained after receiving a command complete event by calling the following method:

| Method Signature | Language |
|--|----------|
| Diagnostic[] readBufferedDiagnosticHistory(); | Java |
| TrkDrvObjectArray * readBufferedDiagnosticHistory(); | C++ |
| ObjectArray readBufferedDiagnosticHistory(); | C# |

4.6.4 Status History

To obtain a list of all status messages generated by the tracker, the method statusHistory() is used. It can be called regardless of whether asynchronous messages have been started or not. This method has the following signature:

| Method Signature | Language |
|--------------------------------------|----------|
| Status[] statusHistory(); | Java |
| TrkDrvObjectArray * statusHistory(); | C++ |
| ObjectArray statusHistory(); | C# |

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.
- This API will throw NoDataAvailableException for Vantage S/E or Vantage S6/E6 tracker

All of the following events are generated when the statusHistory method is called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

If blocking is enabled, it returns an array of Status objects representing all messages in the tracker's status history list. If blocking is disabled the method returns null. The message can then be obtained after receiving a command complete event by calling the following method:

| Method Signature | Language |
|---------------------------------------|----------|
| Status[] readBufferedStatusHistory(); | Java |

| Method Signature | Language |
|--|----------|
| TrkDrvObjectArray * readBufferedStatusHistory(); | C++ |
| JobobjectArray readBufferedStatusHistory(); | C# |

4.6.5 Alarm History

To obtain a list of all alarm messages generated by the tracker, the method alarmHistory() is used. It can be called regardless of whether asynchronous messages have been started or not. This method has the following signature:

| Method Signature | Language |
|-------------------------------------|----------|
| Alarm[] alarmHistory(); | Java |
| TrkDrvObjectArray * alarmHistory(); | C++ |
| JobobjectArray alarmHistory(); | C# |

Restrictions:

- A connection must be established with a tracker, otherwise NotConnectedException is thrown.
- A command may not already be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- A foreground measurement cannot be in progress when the method is called, otherwise InterfaceBusyException is thrown.
- Exclusive access must not be held by a thread other than the caller, otherwise InterfaceBusyException is thrown.
- This API will throw NoDataAvailableException for Vantage S/E or Vantage S6/E6 tracker

All of the following events are generated when the alarmHistory method is called:

- Command Complete – Indicates that the command finished and in also indicates whether it succeeded or failed.
- Busy – Is generated twice. The first time indicates that the command state is busy. The second indicates that the command state is not busy. Note: the event is not generated for each method if exclusive access is granted to a thread.

If blocking is enabled, it returns an array of Alarm objects representing all messages in the tracker's status history list. If blocking is disabled the method returns null. The message can then be obtained after receiving a command complete event by calling the following method:

| Method Signature | Language |
|---|----------|
| Alarm readBufferedAlarmHistory(); | Java |
| TrkDrvObjectArray * readBufferedAlarmHistory(); | C++ |
| JobobjectArray readBufferedAlarmHistory(); | C# |

4.6.6 Asynchronous Message Objects

4.6.6.1 Diagnostic Messages

Diagnostic messages are received by an application by either calling diagnosticHistory or by receiving a diagnosticEvent. The Diagnostic class has several methods that provide information about the message. They are as follows:

- getCode()
- getID();
- getTimeStamp();

- `getTrackerSpecifics();`

4.6.6.1.1 Code

Each diagnostic message has a code associated with it. That code is obtained using the `getCode` method, which has the following signature:

| Method Signature | Language |
|-----------------------------|----------|
| <code>int getCode();</code> | Java |
| <code>int getCode();</code> | C++ |
| <code>int getCode();</code> | C# |

4.6.6.1.2 ID

Each diagnostic message has an ID associated with it to differentiate it from other diagnostic messages with the same code. This ID is obtained using the `getID` method, which has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| <code>int getID();</code> | Java |
| <code>int getID();</code> | C++ |
| <code>int getID();</code> | C# |

4.6.6.1.3 Time

The `getTimeStamp` method returns a time stamp that tracker assigned to the message. It has the following signature:

| Method Signature | Language |
|-------------------------------------|----------|
| <code>double getTimeStamp();</code> | Java |
| <code>double getTimeStamp();</code> | C++ |
| <code>double getTimeStamp();</code> | C# |

The unit of measure for the time stamp is in seconds. It represents the number of seconds that have elapsed since January 1, 1970 00:00:00 GMT.

4.6.6.1.4 TrackerSpecifics

Information that is specific to the tracker is kept with the diagnostic message. This information is obtained by calling the `getTrackerSpecifics` method. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>String getTrackerSpecifics();</code> | Java |
| <code>char * getTrackerSpecifics();</code> | C++ |
| <code>String getTrackerSpecifics();</code> | C# |

4.6.6.2 Status Messages

Status messages are received by an application by either calling `statusHistory` or by receiving a `StatusEvent`. The `Status` class has several methods that provide information about the message. They are as follows:

- `getCode`
- `getState`
- `getID`
- `getTimeStamp`
- `getTrackerSpecifics`

4.6.6.2.1 Code

Each status message has a code associated with it. That code is obtained using the `getCode` method, which has the following signature:

| Method Signature | Language |
|-----------------------------|----------|
| <code>int getCode();</code> | Java |
| <code>int getCode();</code> | C++ |
| <code>int getCode();</code> | C# |

4.6.6.2.2 State

A status can either be on or off. This state is obtained by calling the `getState` method, which has the following signature:

| Method Signature | Language |
|----------------------------------|----------|
| <code>boolean getState();</code> | Java |
| <code>bool getState();</code> | C++ |
| <code>bool getState();</code> | C# |

This method returns true if the state is on and false if the state is off.

4.6.6.2.3 ID

Each status message has an ID associated with it to differentiate it from other status messages with the same code and state. This ID is obtained using the `getID` method, which has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| <code>int getID();</code> | Java |
| <code>int getID();</code> | C++ |
| <code>int getID();</code> | C# |

4.6.6.2.4 Time

The `getTimeStamp` method returns a time stamp that tracker assigned to the message. It has the following signature:

| Method Signature | Language |
|-------------------------------------|----------|
| <code>double getTimeStamp();</code> | Java |
| <code>double getTimeStamp();</code> | C++ |
| <code>double getTimeStamp();</code> | C# |

The unit of measure for the time stamp is in seconds. It represents the number of seconds that have elapsed since January 1, 1970 00:00:00 GMT.

4.6.6.2.5 *TrackerSpecifics*

Information that is specific to the tracker is kept with the status message. This information is obtained by calling the `getTrackerSpecifics` method. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>String getTrackerSpecifics();</code> | Java |
| <code>char * getTrackerSpecifics();</code> | C++ |
| <code>String getTrackerSpecifics();</code> | C# |

4.6.6.3 Change Messages

Change messages are received by an application through change events. The `Change` class has several methods that provide information about the message. They are as follows:

- `getCode()`
- `getID();`
- `getTimeStamp();`
- `getTrackerSpecifics();`

4.6.6.3.1 *Code*

Each change message has a code associated with it. That code is obtained using the `getCode` method, which has the following signature:

| Method Signature | Language |
|-----------------------------|----------|
| <code>int getCode();</code> | Java |
| <code>int getCode();</code> | C++ |
| <code>int getCode();</code> | C# |

4.6.6.3.2 *ID*

Each change message has an ID associated with it to differentiate it from other change messages with the same code. This ID is obtained using the `getID` method, which has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| <code>int getID();</code> | Java |
| <code>int getID();</code> | C++ |
| <code>int getID();</code> | C# |

4.6.6.3.3 *Time*

The `getTimeStamp` method returns a time stamp that tracker assigned to the message. It has the following signature:

| Method Signature | Language |
|------------------|----------|
|------------------|----------|

| Method Signature | Language |
|------------------------|----------|
| double getTimeStamp(); | Java |
| double getTimeStamp(); | C++ |
| double getTimeStamp(); | C# |

The unit of measure for the time stamp is in seconds. It represents the number of seconds that have elapsed since January 1, 1970 00:00:00 GMT.

4.6.6.3.4 TrackerSpecifics

Information that is specific to the tracker is kept with the change message. This information is obtained by calling the getTrackerSpecifics method. It has the following signature:

| Method Signature | Language |
|-------------------------------|----------|
| String getTrackerSpecifics(); | Java |
| char * getTrackerSpecifics(); | C++ |
| String getTrackerSpecifics(); | C# |

4.6.6.4 Alarm Messages

Alarm messages are received by an application by either calling alarmHistory or by receiving an AlarmEvent. The Alarm class has several methods that provide information about the message. They are as follows:

- getAlarmNumber
- getState
- getID
- getTimeStamp
- getTrackerSpecifics

4.6.6.4.1 Alarm Number

The alarm number specifies which alarm is being processed. That code is obtained using the getAlarmNumber method, which has the following signature:

| Method Signature | Language |
|-----------------------|----------|
| int getAlarmNumber(); | Java |
| int getAlarmNumber(); | C++ |
| int getAlarmNumber(); | C# |

4.6.6.4.2 State

An alarm can either be on or off. This state is obtained by calling the getState method, which has the following signature:

| Method Signature | Language |
|---------------------|----------|
| boolean getState(); | Java |
| bool getState(); | C++ |
| bool getState(); | C# |

This method returns true if the state is on and false if the state is off.

4.6.6.4.3 ID

Each alarm message has an ID associated with it to differentiate it from other alarm messages with the same code and state. This ID is obtained using the `getID` method, which has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| <code>int getID();</code> | Java |
| <code>int getID();</code> | C++ |
| <code>int getID();</code> | C# |

For Vantage S/E or Vantage S6/E6 tracker the method `getID` will always return -1.

4.6.6.4.4 Time

The `getTimeStamp` method returns a time stamp that tracker assigned to the message. It has the following signature:

| Method Signature | Language |
|-------------------------------------|----------|
| <code>double getTimeStamp();</code> | Java |
| <code>double getTimeStamp();</code> | C++ |
| <code>double getTimeStamp();</code> | C# |

The unit of measure for the time stamp is in seconds. It represents the number of seconds that have elapsed since January 1, 1970 00:00:00 GMT.

4.6.6.4.5 TrackerSpecifics

Information that is specific to the tracker is kept with the alarm message. This information is obtained by calling the `getTrackerSpecifics` method. It has the following signature:

| Method Signature | Language |
|--|----------|
| <code>String getTrackerSpecifics();</code> | Java |
| <code>char * getTrackerSpecifics();</code> | C++ |
| <code>String getTrackerSpecifics();</code> | C# |

For Vantage S/E or Vantage S6/E6 tracker, tracker specifics will contain the value at which the alarm occurred.

4.7 Available Trackers

Use this method to find all the available trackers (FARO, Vantage, VantageS/E, and Vantage S6/E6) on the network. This method is available from SDK 5.1.5 and later.

| Method Signature | Language |
|---|----------|
| <code>AvailableTrackers[] availableTrackers();</code> | Java |
| <code>TrkDrvObjectArray *availableTrackers ();</code> | C++ |
| <code>JobjectArray availableTrackers();</code> | C# |

When this method is called, it returns an array of `AvailableTracker` objects. Please refer to section 4.7.1 for `AvailableTracker` class.

```

//Java
    AvailableTracker[] availableTrackers = Tracker.availableTrackers();
    int num = availableTrackers.length;
    for(int i = 0; i < num; i++)
    {
        availableTrackers[i]. getConnectionType ();
    }

//C++
    TrkDrvObjectArray* availableTrackers = Tracker::availableTrackers();
    int num = availableTrackers->getNumElements();
    for(int i = 0; i < num; i++)
    {
        AvailableTracker* availableTracker = (AvailableTracker *) availableTrackers -
        >getElement(i);
        availableTracker ->getConnectionType();
    }

C#
    JObjectArray availableTrackers = Tracker.availableTrackers();
    int num = availableTrackers.numElements;
    for(int i = 0; i < num; i++)
    {
        AvailableTracker availableTracker = (AvailableTracker)availableTrackers.getElement(i);
        availableTracker.getConnectionType();
    }

```

4.7.1 AvailableTracker

Available Trackers are returned as AvailableTracker objects. The object has several methods for obtaining information about each tracker.

4.7.1.1 Serial Number

This method returns the serial number of the tracker. It has the following signature:

| Method Signature | Language |
|------------------------|----------|
| String getSerialNum(); | Java |
| String getSerialNum(); | C++ |
| String getSerialNum(); | C# |

4.7.1.2 IP Address

This method returns the ip address of the tracker. It has the following signature:

| Method Signature | Language |
|-------------------------|----------|
| String getIpAddress(); | Java |
| char* getIpAddress (); | C++ |
| String getIpAddress (); | C# |

4.7.1.3 Connection Type

This method return the connection type(Wired, Wifi) of the tracker. It has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| int getConnectionType(); | Java |
| int getConnectionType (); | C++ |
| int getConnectionType (); | C# |

This method returns the following values:

- 0 – WIRED
- 1 – WIFI

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|-------------------------|----------|
| 0 | AvailableTracker.WIRED | Java |
| | AvailableTracker::WIRED | C++ |
| | AvailableTracker.WIRED | C# |
| 1 | AvailableTracker.WIFI | Java |
| | AvailableTracker::WIFI | C++ |
| | AvailableTracker.WIFI | C# |

4.7.1.4 Model Type

This method returns the connection type (FARO, Vantage, VantageS) of the tracker. It has the following signature:

| Method Signature | Language |
|----------------------|----------|
| int getModelType(); | Java |
| int getModelType (); | C++ |
| int getModelType (); | C# |

This method returns the following values:

- 0 – MODEL_FARO
- 1 – MODEL_VANTAGE
- 2 – MODEL_VANTAGE_S

These can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|-----------------------------------|----------|
| 0 | AvailableTracker.MODEL_FARO | Java |
| | AvailableTracker::MODEL_FARO | C++ |
| | AvailableTracker.MODEL_FARO | C# |
| 1 | AvailableTracker.MODEL_VANTAGE | Java |
| | AvailableTracker::MODEL_VANTAGE | C++ |
| | AvailableTracker.MODEL_VANTAGE | C# |
| 2 | AvailableTracker.MODEL_VANTAGE_S | Java |
| | AvailableTracker::MODEL_VANTAGE_S | C++ |
| | AvailableTracker.MODEL_VANTAGE_S | C# |

Both VantageS/E and VantageS6/E6 tracker will return model type as MODEL_VANTAGE_S.

4.8 Miscellaneous Methods

A few methods in the tracker class do not fit into any of the other categories already described. These methods are included in this section.

4.8.1 Abort

The abort method is used to stop a control command that is currently in progress. If blocking is not used when the control method is called, the abort method can be called from the same thread or from another thread. However, if blocking is enabled when the control method is called, the abort command, the application will only be able to call abort from another thread. The abort method has the following signature:

| Method Signature | Language |
|------------------|----------|
| void abort(); | Java |
| void abort(); | C++ |
| void abort(); | C# |

If no command is in progress when it is called, it will simply return. If a command is in progress, abort will return when the command abort sequence completes. The method that was aborted will throw AbortedException.

Care must be taken when using the abort method. A control method might complete before the abort method is called. The application must account for race conditions.

4.8.2 Version Information

Version information on the interface is obtained by calling the version method. It has the following signature:

VersionInfo version();

The VersionInfo object that is returned has four methods that provide the version number:

| Method Signature | Language |
|--------------------|----------|
| int majorNumber(); | Java |
| int majorNumber(); | C++ |
| int majorNumber(); | C# |

| Method Signature | Language |
|--------------------|----------|
| int minorNumber(); | Java |
| int minorNumber(); | C++ |
| int minorNumber(); | C# |

| Method Signature | Language |
|-------------------------|----------|
| int minorMinorNumber(); | Java |
| int minorMinorNumber(); | C++ |
| int minorMinorNumber(); | C# |

| Method Signature | Language |
|--------------------|----------|
| int buildNumber(); | Java |

| Method Signature | Language |
|--------------------|----------|
| int buildNumber(); | C++ |
| int buildNumber(); | C# |

For example, if the version of the interface was 1.2.3.4, the major number would be 1, the minor number would be 2, the minor minor number would be 3, and the build number would be 4. All values but the build number is potentially relevant to an application. The build number is intended for use by FARO.

In addition, there is another method that is part of the VersionInfo class that has the following signature:

| Method Signature | Language |
|----------------------|----------|
| String specialStr(); | Java |
| char * specialStr(); | C++ |
| String specialStr(); | C# |

The string returned by this method will usually be empty. It normally is used to indicate the development phase for the interface such as alpha or beta.

4.8.3 Busy

The busy method returns true if the interface is busy and cannot execute another command. It has the following signature:

| Method Signature | Language |
|------------------|----------|
| boolean busy(); | Java |
| bool busy(); | C++ |
| bool busy(); | C# |

4.8.4 Busy For Current Thread

To determine if the thread is able to make calls into the interface, the busyForCurrentThread method is used. It has the following signature:

| Method Signature | Language |
|---------------------------------|----------|
| boolean busyForCurrentThread(); | Java |
| bool busyForCurrentThread(); | C++ |
| bool busyForCurrentThread(); | C# |

This method returns true if the interface is busy. The busyForCurrentThread method will return false when the busy method returns true in the situation where the current thread has gained exclusive access to the interface.

4.8.5 Measurement In Progress

The measureInProgress method is used to determine if the interface is currently running a measurement. It has the following signature:

| Method Signature | Language |
|------------------------------|----------|
| boolean measureInProgress(); | Java |
| bool measureInProgress(); | C++ |

| Method Signature | Language |
|---------------------------|----------|
| bool measureInProgress(); | C# |

The method returns true if a measurement is in progress.

4.8.6 Background Measurement In Progress

The bkndMeasureInProgress method is used to determine if the interface is currently running a background measurement. It has the following signature:

| Method Signature | Language |
|----------------------------------|----------|
| boolean bkndMeasureInProgress(); | Java |
| bool bkndMeasureInProgress(); | C++ |
| bool bkndMeasureInProgress(); | C# |

The method returns true if a background measurement is in progress.

4.8.7 Sample Rate

To determine the sample rate of both foreground and background measurements in the tracker, the sampleRate method can be used. It has the following signature:

| Method Signature | Language |
|-------------------|----------|
| int sampleRate(); | Java |
| int sampleRate(); | C++ |
| int sampleRate(); | C# |

This returns the number samples taken per second.

4.8.8 Asynchronous Messages Started

To determine if asynchronous messages have been started, the asyncStarted method is used. It has the following signature:

| Method Signature | Language |
|-------------------------|----------|
| boolean asyncStarted(); | Java |
| bool asyncStarted(); | C++ |
| bool asyncStarted(); | C# |

This method returns true if asynchronous methods have been successfully started.

4.8.9 Get IPAddress

This method will return the ip address of the tracker that you are currently connected. This method is available from SDK 5.1.5 and later.

| Method Signature | Language |
|-------------------------|----------|
| String getIPAddress(); | Java |
| char* getIPAddress (); | C++ |
| String getIPAddress (); | C# |

5 Events

The interface uses events to provide information to an application in an asynchronous manner. The application receives these events using listeners. Creating a class that extends the listener class associated with the event and implementing the event method that is a member of that class creates a listener. An instance of this class must then be instantiated and added to the interface using the appropriate method call in the Tracker class.

There are a few programming issues related to events:

1. The event methods are called from their own thread. Any data manipulated by the event method must be thread-safe.
2. Most tracker methods **should not** be called from the event method. The exceptions to this are as follows:
 - readBkndMeasurePointData
 - readBufferedLevel
 - readBufferedDiagnosticHistory
 - readBufferedStatusHistory
 - readBufferedLevel
 - readBufferedMaterialTemperature
 - readMeasurePointData

5.1 Connection Event

A connection event is generated whenever the state of the connection between an application and a tracker changes. These events occur when connect and disconnect methods are called and complete successfully.

The listener for this event is added to the Tracker interface by calling the addConnectListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addConnectListener(ConnectListener listener); | Java |
| void addConnectListener(ConnectListener *listener); | C++ |
| void addConnectListener(ConnectListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of ConnectListener. When a connect event occurs, the connection changed method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void connectionChanged(ConnectEvent event); | Java |
| void connectionChanged(ConnectEvent *event); | C++ |
| void connectionChanged(ConnectEvent event); | C# |

The event object that is passed to the connection changed method holds the information about the connect event. The isConnected method that is part of the ConnectEvent class is used to determine if the event represents a successful connect or a successful disconnect. It has the following signature:

| Method Signature | Language |
|------------------------|----------|
| boolean isConnected(); | Java |
| bool isConnected(); | C++ |
| bool isConnected(); | C# |

This method returns true if a connection was established. It returns false if the connection was broken.

5.2 Command Complete Event

A command complete event is generated for all types of methods except for status and configuration methods.

The listener for this event is added to the Tracker interface by calling the addCommandCompleteListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addCommandCompleteListener(CommandCompleteListener listener); | Java |
| void addCommandCompleteListener(CommandCompleteListener *listener); | C++ |
| void addCommandCompleteListener(CommandCompleteListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of CommandCompleteListener. When a command complete event occurs, commandComplete method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void commandComplete(CommandCompleteEvent event); | Java |
| void commandComplete(CommandCompleteEvent *event); | C++ |
| void commandComplete(CommandCompleteEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getRelatedEvent method that is part of the CommandCompleteEvent class is used to determine if the command completed successfully. It has the following signature:

| Method Signature | Language |
|---|----------|
| TrackerException getRelatedException(); | Java |
| TrackerException * getRelatedException(); | C++ |
| TrackerException getRelatedException(); | C# |

This method returns null if the command completed successfully. It returns a TrackerException object if the command failed. Note, it does not throw this exception. The Tracker method that caused this event would also throw this same exception if blocking was enabled.

5.3 Measure Data Event

A measure data event is generated whenever the interface receives the number of observations specified by the measure data event rate property of Tracker.

The listener for this event is added to the Tracker interface by calling the addMeasureDataListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addMeasureDataListener(MeasureDataListener listener); | Java |
| void addMeasureDataListener(MeasureDataListener *listener); | C++ |
| void addMeasureDataListener(MeasureDataListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of MeasureDataListener. When a measure data event occurs, the dataAvailable method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void dataAvailable(MeasureDataEvent event); | Java |
| void dataAvailable(MeasureDataEvent *event); | C++ |
| void dataAvailable(MeasureDataEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getNumMeasurements method that is part of the MeasureDataEvent class is used to determine how many observations were received. It has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| int getNumMeasurements(); | Java |
| int getNumMeasurements(); | C++ |
| int getNumMeasurements(); | C# |

This method returns the number of observations related to this event. This number can then be used to call the readMeasurePointData method in Tracker.

5.4 Background Measure Data Event

A background measure data event is generated whenever the interface receives the number of observations specified by the background measure data event rate property of Tracker.

The listener for this event is added to the Tracker interface by calling the addBkndMeasureDataListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addBkndMeasureDataListener(BkndMeasureDataListener listener); | Java |
| void addBkndMeasureDataListener(BkndMeasureDataListener *listener); | C++ |
| void addBkndMeasureDataListener(BkndMeasureDataListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of BkndMeasureDataListener. When a measure data event occurs, the dataAvailable method is called. It has the following signature:

| Method Signature | Language |
|---|----------|
| void dataAvailable(BkndMeasureDataEvent event); | Java |
| void dataAvailable(BkndMeasureDataEvent * event); | C++ |
| void dataAvailable(BkndMeasureDataEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getNumMeasurements method that is part of the BkndMeasureDataEvent class is used to determine how many observations were received. It has the following signature:

| Method Signature | Language |
|---------------------------|----------|
| int getNumMeasurements(); | Java |
| int getNumMeasurements(); | C++ |
| int getNumMeasurements(); | C# |

This method returns the number of observations related to this event. This number can then be used to call the readBkndMeasurePointData method in Tracker.

5.5 Busy Event

A busy event is generated whenever the interface changes its busy state. The interface can be busy for the following reasons:

- Another command is already in progress.
- Exclusive access has been granted to a thread.
- A foreground measurement is in progress.
- A background measurement is in progress.

Note that the restrictions on the interface vary depending on how the driver is busy. See the various subsections under methods to see how the busy state is effected and how methods are restricted.

The listener for this event is added to the Tracker interface by calling the addBusyListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addBusyListener(BusyListener listener); | Java |
| void addBusyListener(BusyListener *listener); | C++ |
| void addBusyListener(BusyListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of BusyListener. When a busy event occurs, the stateChanged method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void stateChanged(BusyEvent event); | Java |
| void stateChanged(BusyEvent *event); | C++ |
| void stateChanged(BusyEvent event); | C# |

The event object that is passed to this method holds the information about the event. The following methods can be called to determine which states have changed:

| Method Signature | Language |
|----------------------------|----------|
| boolean getCommandState(); | Java |
| bool getCommandState(); | C++ |
| bool getCommandState(); | C# |

| Method Signature | Language |
|----------------------------|----------|
| boolean getMeasureState(); | Java |
| bool getMeasureState(); | C++ |
| bool getMeasureState(); | C# |

| Method Signature | Language |
|--------------------------------|----------|
| boolean getBkndMeasureState(); | Java |
| bool getBkndMeasureState(); | C++ |
| bool getBkndMeasureState(); | C# |

Note that if two states change at the same time, only one event will be generated. For example, if the interface if the stopMeasurePoint method is called in the Tracker object, the busy event generated when this command completes will show that the command state from true to false and the measure state changed from true to false.

5.6 Application Event

An application event is generated whenever a tracker application starts or stops successfully. This event is most useful for tracker applications that can close on their own without the parent application calling stopApplicationFrame.

The listener for this event is added to the Tracker interface by calling the addApplicationListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addApplicationListener(ApplicationListener listener); | Java |
| void addApplicationListener(ApplicationListener *listener); | C++ |
| void addApplicationListener(ApplicationListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of ApplicationListener. When an application event occurs, the stateChanged method is called. It has the following signature:

| Method Signature | Language |
|---|----------|
| void stateChanged(ApplicationEvent event); | Java |
| void stateChanged(ApplicationEvent *event); | C++ |
| void stateChanged(ApplicationEvent event); | C# |

The event object that is passed to this method holds the information about the event. The following methods can be called to determine what changed:

| Method Signature | Language |
|----------------------|----------|
| String getAppName(); | Java |
| char * getAppName(); | C++ |
| String getAppName(); | C# |

| Method Signature | Language |
|-------------------------|----------|
| boolean isAppRunning(); | Java |
| bool isAppRunning(); | C++ |
| bool isAppRunning(); | C# |

The getAppName method returns the name of the application that changed state. The isAppRunning methods returns true if the event represents an application that has started running and returns false if the event is for an application that has stopped.

5.7 Diagnostic Event

A diagnostic event is generated whenever the interface receives an asynchronous diagnostic message from the tracker.

The listener for this event is added to the Tracker interface by calling the addDiagnosticListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addDiagnosticListener(DiagnosticListener listener); | Java |
| void addDiagnosticListener(DiagnosticListener *listener); | C++ |
| void addDiagnosticListener(DiagnosticListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of DiagnosticListener. When a diagnostic event occurs, the diagnosticReported method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void diagnosticReported(DiagnosticEvent event); | Java |
| void diagnosticReported(DiagnosticEvent *event); | C++ |
| void diagnosticReported(DiagnosticEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getDiagnostic method that is part of the DiagnosticEvent class is used to get the related diagnostic record. It has the following signature:

| Method Signature | Language |
|-------------------------------|----------|
| Diagnostic getDiagnostic(); | Java |
| Diagnostic * getDiagnostic(); | C++ |
| Diagnostic getDiagnostic(); | C# |

See section 4.6.6.1 for an explanation of the Diagnostic class.

Diagnostic Events are not generated for Vantage S/E or Vantage S6/E6 tracker.

5.8 Status Event

A status event is generated whenever the interface receives an asynchronous status message from the tracker.

The listener for this event is added to the Tracker interface by calling the addStatusListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addStatusListener(StatusListener listener); | Java |
| void addStatusListener(StatusListener *listener); | C++ |
| void addStatusListener(StatusListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of StatusListener. When a status event occurs, the statusChanged method is called. It has the following signature:

| Method Signature | Language |
|---|----------|
| void statusChanged(StatusEvent event); | Java |
| void statusChanged(StatusEvent *event); | C++ |
| void statusChanged(StatusEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getStatus method that is part of the StatusEvent class is used to get the related status record. It has the following signature:

| Method Signature | Language |
|---------------------|----------|
| Status getStatus(); | Java |

| Method Signature | Language |
|-----------------------|----------|
| Status * getStatus(); | C++ |
| Status getStatus(); | C# |

See section 4.6.6.2 for an explanation of the Status class.

Status Events are not generated for Vantage S/E or Vantage S6/E6 trackers.

5.9 Change Event

A change event is generated whenever the interface receives an asynchronous change message from the tracker.

The listener for this event is added to the Tracker interface by calling the addChangeListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addChangeListener(ChangeListener listener); | Java |
| void addChangeListener(ChangeListener *listener); | C++ |
| void addChangeListener(ChangeListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of ChangeListener. When a change event occurs, the changeOccurred method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void changeOccurred(ChangeEvent event); | Java |
| void changeOccurred(ChangeEvent *event); | C++ |
| void changeOccurred(ChangeEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getChange method that is part of the ChangeEvent class is used to get the related change record. It has the following signature:

| Method Signature | Language |
|-----------------------|----------|
| Change getChange(); | Java |
| Change * getChange(); | C++ |
| Change getChange(); | C# |

See section 4.6.6.3 for an explanation of the Change class.

For Vantage S/E trackers, below are the change event(s) generated.

TRACKER_BATTERY_1_STATE_CHANGE (SDK >= 5.1.8 and firmware >= 2.0)
 TRACKER_BATTERY_2_STATE_CHANGE (SDK >= 5.1.8 and firmware >= 2.0)
 RUN_AUTO_COMP (SDK >= 5.1.8 and firmware >= 2.10.0)

For Vantage S6/E6 sixdof capable trackers, change events generated are

TARGET_TYPE (SDK >= 5.1.0 and firmware >= 2.0)
 PROBE_CONNECTED (SDK >= 5.1.0 and firmware >= 2.0)
 PROBE_CHANGE (SDK >= 5.1.5 and firmware >= 2.6.1)
 PROBE_BATTERY_STATE_CHANGE (SDK >= 5.1.5 and firmware >= 2.6.1)

Below is an example for these events in C++.


```

class MyChangeListener : public ChangeListener
{
    void changeOccured(ChangeEvent* changeEvent)
    {
        Change* change = changeEvent->getChange();
        int code = change->getCode();
        if (code == Change::TARGET_TYPE)
        {
            TargetType* target= ((TargetType*)change->getRelatedData());
            SixDofTargetType* sixdofTarget =
dynamic_cast<SixDofTargetType*>(target);
            if(sixdofTarget)
            {
                //Catch the event. Do not call any tracker methods here.
                //You need to initiate any tracker methods outside this
                callback event
            }
        }
        else if(code == Change::PROBE_CONNECTED)
        {
            bool state = ((Boolean*)change->getRelatedData())->booleanValue();
            //Catch the event. Do not call any tracker methods here.
            //You need to initiate any tracker methods outside this callback event
        }
        else if(code == Change::PROBE_CHANGE)
        {
            Probe* probe = ((Probe*)change->getRelatedData());
            double d = probe->getDiameter();
            //Catch the event. Do not call any tracker methods here.
            //You need to initiate any tracker methods outside this callback event
        }
        else if (code == Change::PROBE_BATTERY_STATE_CHANGE)
        {
            ProbeBatteryState* state= ((ProbeBatteryState*)change-
>getRelatedData());
            int state = state->getBatteryState();
            //Please refer to probeBatteryState API for the different states returned
            //Catch the event. Do not call any tracker methods here.
            //You need to initiate any tracker methods outside this callback
        }
        else if(code == Change::TRACKER_BATTERY_1_STATE_CHANGE)
        {
            BatteryState* state= ((BatteryState*)change->getRelatedData());
            bool state = state->getBatteryState();
            //Please refer to Battery State API for the different states returned
            //Catch the event. Do not call any tracker methods here.
            //You need to initiate any tracker methods outside this callback event
        }
    }
}

```

```

else if(code == Change:: TRACKER_BATTERY_2_STATE_CHANGE)
{
    BatteryState* state= ((BatteryState*)change->getRelatedData());
    bool state = state->getBatteryState();
    //Please refer to Battery State API for the different states returned
    //Catch the event. Do not call any tracker methods here.
    //You need to initiate any tracker methods outside this callback event
}
else if(code == Change:: RUN_AUTO_COMP)
{
    bool state = ((Boolean*)change->getRelatedData())->booleanValue();
    If true, it is recommended to run the compensation. If false, compensation has completed the run.

    //Catch the event. Do not call any tracker methods here.
    //When this event occurs, call the runAutomatedComp API. Please see
    section 4.2.18 for the API.
    //Please note that you need to initiate this method outside this callback
    event
}
}

}

//C++
//Create listener
MyChangeListener * pListener = new MyChangeListener ();
//Start Asynchronous messages
trk->startAsync();
//Add Listener
trk-> addChangeListener(pListener);
//To remove listener
trk->removeChangeListener(pListener);
trk->stopAsync();

```

5.9.1 RUN_AUTO_COMP

Please note that you need to add the change listener and start the asynchronous messages to get the event. If you have a change listener already implemented, you can just capture the event in the existing listener as shown above. A working example on how to capture the event and launch the compensation is also provided for your reference in the both the C++ and C# demo examples that come with the SDK kit.

```

C++
class MyChangeListener : public ChangeListener
{
    void changeOccured(ChangeEvent* changeEvent)
    {
        Change* change = changeEvent->getChange();
        int code = change->getCode();
        if(code == Change:: RUN_AUTO_COMP)
        {
            bool state = ((Boolean*)change->getRelatedData())->booleanValue();
            If true, it is recommended to run the compensation. If false, compensation has completed the run.

            //Catch the event. Do not call any tracker methods here.
            //When this event occurs, call the runAutomatedComp API.
            //Please note that you need to initiate this method outside this callback

            event
        }
    }
}

void launchRunAutomatedComp()
{
    //This method should be called within blocking mode to get the return value.
    //If run in non blocking mode, the cmd will return -1.
    //store the blocking state
    boolean block = trk->getBlocking();
    //turn the blocking mode on
    trk->setBlocking(true);
    int result = trk->runAutomatedComp();
    trk->setBlocking(block); //restore blocking state

    if(result == AutomatedCompResult::FAILED_TO_FIND_TARGET_1 ||
       result == AutomatedCompResult::FAILED_TO_FIND_TARGET_2 ||
       result == AutomatedCompResult::LEARN_TARGETS_PROCEDURE_NOT_RUN)
    {
        startApplicationFrame("CompIT", "-a AUTOMATED_COMP");
    }
}

```

5.10 Alarm Event

An alarm event is generated whenever the interface receives an asynchronous alarm message from the tracker. See section 4.3.13 for an explanation of setting up alarms.

The listener for this event is added to the Tracker interface by calling the addAlarmListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addAlarmListener(AlarmListener listener); | Java |
| void addAlarmListener(AlarmListener *listener); | C++ |
| void addAlarmListener(AlarmListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of AlarmListener. When an alarm event occurs, the alarmChanged method is called. It has the following signature:

| Method Signature | Language |
|---|----------|
| void alarmChanged(AlarmEvent event); | Java |
| void alarmChanged(AlarmEvent *event); | C++ |
| void alarmChanged(AlarmEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getAlarm method that is part of the AlarmEvent class is used to get the related status record. It has the following signature:

| Method Signature | Language |
|---------------------|----------|
| Alarm getAlarm(); | Java |
| Alarm * getAlarm(); | C++ |
| Alarm getAlarm(); | C# |

See section 4.6.6.4 for an explanation of the Alarm class.

5.11 Probe Button Event

SDK version 5.1.0 and later only supports the probe button event. The tracker also has to be Vantage S6 or E6 and be sixdof capable.

- Sixdof Probe consists of four buttons. Probe button events are generated for all the four buttons when they are pressed and released.
- These events are generated whenever the interface receives an asynchronous change message from the tracker.
- For the events to be generated, asynchronous messages has to be started using startAsync method and probe listener needs to be added using addProbeButtonListener method.
- These events can be configured to execute any tracker functionality like start measurements for Button 1 press and stop measurements for Button 2 press as in below example in section 5.11.1

It has the following signature:

| Method Signature | Language |
|---|----------|
| void addProbeButtonListener (ProbeButtonListener listener); | Java |
| void addProbeButtonListener (ProbeButtonListener *listener); | C++ |
| void addProbeButtonListener (ProbeButtonListener listener); | C# |

This method will throw an UnsupportedOperationException if the tracker is not sixdof capable.

The listener parameter is an instance of an object created from a subclass of ProbeButtonListener. When a probe button event occurs, the buttonStatusChanged method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void buttonStatusChanged(ProbeButtonEvent event); | Java |
| void buttonStatusChanged (ProbeButtonEvent *event); | C++ |
| void buttonStatusChanged (ProbeButtonEvent event); | C# |

The event object that is passed to this method holds the information about the event. The getID method that is part of the ProbeButtonEvent class is used to get the ID of the button on which the event occurred. It has the following signature:

| Method Signature | Language |
|------------------|----------|
| int getID(); | Java |
| int getID(); | C++ |
| int getID(); | C# |

The ID of the button's can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|-----------------------------|----------|
| 1 | ProbeButtonIndex.BUTTON_1 | Java |
| | ProbeButtonIndex:: BUTTON_1 | C++ |
| | ProbeButtonIndex. BUTTON_1 | C# |
| 2 | ProbeButtonIndex. BUTTON_2 | Java |
| | ProbeButtonIndex:: BUTTON_2 | C++ |
| | ProbeButtonIndex. BUTTON_2 | C# |
| 3 | ProbeButtonIndex. BUTTON_3 | Java |
| | ProbeButtonIndex:: BUTTON_3 | C++ |
| | ProbeButtonIndex. BUTTON_3 | C# |
| 4 | ProbeButtonIndex. BUTTON_4 | Java |
| | ProbeButtonIndex:: BUTTON_4 | C++ |
| | ProbeButtonIndex. BUTTON_4 | C# |

The getStatus method that is part of the ProbeButtonEvent class is used to get the status of the button (on or off) on which the event occurred. It has the following signature:

| Method Signature | Language |
|------------------|----------|
| int getStatus(); | Java |
| int getStatus(); | C++ |
| int getStatus(); | C# |

The status of the button's can be referenced as follows for each of the languages:

| Value | Reference | Language |
|-------|------------------------|----------|
| 0 | ProbeButtonStatus.OFF | Java |
| | ProbeButtonStatus::OFF | C++ |
| | ProbeButtonStatus.OFF | C# |
| 1 | ProbeButtonStatus.ON | Java |
| | ProbeButtonStatus::ON | C++ |
| | ProbeButtonStatus. ON | C# |

5.11.1 Example:

```

class MyProbeButtonListener : public ProbeButtonListener
{
    void buttonStatusChanged(ProbeButtonEvent* probeButtonEvent)
    {
        int buttonStatus = probeButtonEvent->getStatus();
        int buttonId = probeButtonEvent->getId();
        if (buttonStatus == ProbeButtonStatus::ON)
        {
            if (buttonId == ProbeButtonIndex::BUTTON_1)
            {
                //Catch the event
                //Do not call any tracker methods here.
                //You need to initiate any tracker methods outside
                //this callback event
            }
        }
    }
}

//Create add listener
MyProbeButtonListener* pListener = new MyProbeButtonListener();
//Start Asynchronous messages
trk->startAsync();
//Add Listener
trk-> addProbeButtonListener(pListener);

//To remove listener
trk->removeProbeButtonListener(pListener);

```

5.12 Available Event

An available event is generated whenever communications are dropped and reestablished.

The listener for this event is added to the Tracker interface by calling the addAvailableListener method. It has the following signature:

| Method Signature | Language |
|---|----------|
| void addAvailableListener(AvailableListener listener); | Java |
| void addAvailableListener (AvailableListener *listener); | C++ |
| void addAvailableListener (AvailableListener listener); | C# |

The listener parameter is an instance of an object created from a subclass of AvailableListener. When an available event occurs, the availabilityChanged method is called. It has the following signature:

| Method Signature | Language |
|--|----------|
| void availabilityChanged (AvailableEvent event); | Java |
| void availabilityChanged (AvailableEvent *event); | C++ |
| void availabilityChanged (AvailableEvent event); | C# |

The event object that is passed to this method holds the information about the event. The following method can be called to determine if available or not:

| Method Signature | Language |
|-------------------------|----------|
| boolean isAvailable (); | Java |

| Method Signature | Language |
|----------------------|----------|
| bool isAvailable (); | C++ |
| bool isAvailable (); | C# |

The isAvailable methods returns false if communications dropped and returns true if communications are established.

6 .NET specific issues

As mentioned in the introduction, the tracker interface is implemented in Java. The .NET interface is simply a “wrapper” that is placed around the Java implementation. Because of some differences in the two languages, there are a few issues that .NET developers must understand.

6.1 First Instantiation Performs Loads DLL's

The first time any object in the interface is created, the interface attempts to load the library for the Java Runtime Environment. Therefore, the application must ensure that the first time an object from the tracker interface is created, it is safe for it to load the library.

6.2 Namespace

The namespace for the .NET interface to the tracker is “FARO.DeviceInterface”.

6.3 Change the Installation Directory

As noted in section 1.4.2.2, the directory where the files from the kit are installed can be modified using the following methods:

```
FARO.DeviceInterface.Cfg.setInstallDir( pathname )
FARO.DeviceInterface.Cfg.setAndCheckInstallDir( pathname )
```

These methods will return ‘true’ if the path is successfully changed. Note that these methods will fail if the Java Runtime environment is already loaded (see section 6.1).

6.4 JObjectArray

Methods that return arrays will return a single object called JObjectArray rather than an array of objects. The JObjectArray has two methods. The first method provides the number of objects in the array and has the following signature:

| Method Signature | Language |
|-----------------------|----------|
| int getNumElements(); | Java |
| int getNumElements(); | C++ |
| int getNumElements(); | C# |

The second retrieves an object from the array. The parameter for this method specified the index into the array. The index is zero relative. The method has the following signature:

| Method Signature | Language |
|------------------------------|----------|
| object element(int index); | Java |
| object element(int index); | C++ |

| Method Signature | Language |
|------------------------------|----------|
| object element(int index); | C# |

The returned object may be cast to the appropriate class. The class is determined by the method that returned the JSONArray.

6.5 Exceptions

In addition to throwing TrackerException, the interface may throw an ApplicationException. These exceptions are failures of the wrapper and not failures of the instrument. .NET applications should make sure that both exceptions are caught.

7 C++ specific issues

As mentioned in the introduction, the tracker interface is implemented in Java. The C++ interface is simply a “wrapper” that is placed around the Java implementation. Because of some differences in the two languages, there are a few issues that C++ developers must understand.

7.1 First Instantiation Performs Loads DLL's

The first time any object in the interface is created, the interface attempts to load the library for the Java Runtime Environment. Therefore, the application must ensure that the first time an object from the tracker interface is created, it is safe for it to load the library.

7.2 Change the Installation Directory

As noted in section 1.4.5, the directory where the files from the kit are installed can be modified using the following methods:

```
trackerSetInstallDir( pathname )
trackerSetAndCheckInstallDir( pathname )
```

These methods will return ‘true’ if the path is successfully changed. These methods will fail if the Java Runtime environment is already loaded (see section 7.1).

7.3 Destroy Objects Passes as Parameters

Several methods in the interface require objects to be passed as parameters. The interface makes a copy of the object. The caller is responsible for destroying this object after the call. Using the changeTargetType method as an example, the code would be similar to that shown in Figure 3.


```

void appSetMirrorTarget()
{
    MirrorTargetType* mirror;

    try
    {
        /* Create a mirror object */
        mirror = new MirrorTargetType();

        /* Call the interface to set the target type.
        The trk variable is a Tracker object that was
        instantiated and connected elsewhere.*/

        trk->changeTargetType(mirror);

        /* Destroy the mirror object */
        delete mirror;
    }
    catch( TrackerException  *e )
    {
        /* Handle exception here */
        delete e;
    }
}

```

Figure 3

A simpler solution is shown in Figure 4, which simply lets the object go out of scope.

```

void appSetMirrorTarget()
{
    try
    {
        MirrorTargetType mirror;

        /* Call the interface to set the target type.
        The trk variable is a Tracker object that was
        instantiated and connected elsewhere.*/

        trk->changeTargetType(&mirror);
    }
    catch( TrackerException  e )
    {
        /* Handle exception here */
        delete e;
    }
}

```

Figure 4

7.4 Destroy Objects from Method Returns and Exceptions

When a method in the interface returns an object, it actually is returning a copy of the data (TrkDrvObjectArray is an exception. See section 7.6). Because of this, the caller must free that memory allocated for that data. The delete operator must be used for objects, and a special function called trackerDeleteString must be used to delete strings that are returned. An example is shown in Figure 5.

```
void appSetMirrorTarget()
{
    try
    {
        MirrorTargetType mirror;

        /* Call the interface to set the target type. The trk variable is a Tracker object that was */
        /* instantiated and connected elsewhere. */

        trk->changeTargetType(&mirror);
    }
    catch( TrackerException e )
    {
        /* Handle exception here */

        delete e;
    }
}
```

Figure 5

7.5 Do Not Store or Destroy Event Pointer

When a listener method is called, a pointer to the event is passed as a parameter. The method should not attempt to store the pointer or attempt to destroy the object. Instead the method should extract the information from the event and store the data. If the data extracted from the event is an object, the pointer may be stored, but the application is responsible for destroying that object later.

```

class MyDiagListener :: public DiagnosticListener
{
    private:
        Diagnostic *diag;

    public:
        void diagnosticReported(DiagnosticEvent *event)
        {
            /* diag will be destroyed when destroyDiag is called later */
            diag = event->getDiagnostic();

            /* Do not destroy event object before exiting */
        }

        void destroyDiag()
        {
            delete diag;
        }
}

```

Figure 6

7.6 TrkDrvObjectArray

When a method must return an array of data, it returns it as a TrkDrvObjectArray. This class provides a mechanism to extract data and to determine the size of the array.

To obtain the number of elements in the array, the following function can be called:

```
int getNumElements();
```

To retrieve a pointer to an element in the array, the following function can be called:

```
void * getElement( int    index );
```

Index is the element number that is being requested from the array. The pointer that is returned must be cast to the correct object. Note that the pointer returned by the getElement method should not be deleted because it is not a copy of the data. This is an exception to how objects are handled in the rest of the interface. In this case, it is a pointer to a member of the array and will be freed when the TrkDrvObjectArray is destroyed.

```

void getStatusHistory()
{
    Status* status;
    TrkDrvObjectArray* statusArray;
    int i, numElements;
    char* specifics;

    try
    {
        statusArray = trk->diagnosticHistory();
        numElements = statusArray->numElements();
        for (i = 0; i < numElements; ++i)
        {
            status = (Status*) statusArray->getElement(i);

            code = status->getCode();
            specifics = status->getSpecifics();

            /* Do processing here */

            /* The string obtained from the status object must be
            /* destroyed. */
            trackerDeleteString(specifics);

            /* Do not delete status. It was returned from getElement.*/
        }

        /* This deletes the status array object and all of
        the element in the array.*/
        delete statusArray;
    }
    catch( TrackerException e )
    {
        /* Handle exception here */
        delete e;
    }
}

```

7.7 TrkDrvException

Certain exceptions that are thrown by the interface are not a TrackerException or an object derived from TrackerException. This exception type is TrkDrvException. These exceptions are either exceptions that happen in the C++ wrapper, or they are due to low level failures in the Java portion of the interface. These failures are usually related to configuration items such as the jar files or DLL's cannot be located.

If a TrkDrvException is caught, there is one method that can be called to get information about the exception. It is as follows:

```

void getMsg( char    exceptionMsg[],
            int      maxLen      );

```

When called, the message string is copied into the exceptionMsg buffer provided by the caller. The maxLen parameter is passed by the caller to indicate the maximum size message that can be placed in the buffer.

8 Tracker Applications

Tracker applications are handled through the tracker interface using the methods described in section 4.5. Each application has a name and a set of parameters associated with it. The following subsections describe the various tracker applications that may be available depending on the type of tracker you connect to.

Detailed explanations are available in the Tracker User's Guide.

8.1 General

Many of the tracker applications share common traits. Each of those explained in the subsections below.

8.1.1 Parameters

Most tracker applications accept parameters as specified in the parameter string in the startApplicationFrame method (see section 4.5.3). The parameters common to most applications are explained below.

Note, that if a tracker application is given a parameter that it does not understand, it will simply ignore that parameter rather than throw an error. The reason for this is to avoid compatibility problems.

8.1.1.1 -u

The -u parameter can be used to modify units of measure that are displayed and entered by the user.

8.1.1.1.1 Length

The length unit of measure is modified using -uLength as the parameter. It has the following format:

`-uLength = name, scale [,decimal places]`

- *name* – a text string that should be displayed when showing a length value.
- *scale* – a value multiplied with the base unit of measure to convert it to the specified unit of measure. The base unit of measure is meters.
- *decimal places* - The number of decimal places to be displayed when the value is displayed. This parameter is optional. If not specified, the number of decimal places is computed base on the scale.

For example: if the tracker pad should show millimeters for length values, the following would be placed in the parameter string:

`-uLength = millimeters, 1000.0`

To specify that 4 decimal places are to be used with the above example, the following would be placed in the parameter string:

`-uLength = millimeters, 1000.0, 4`

8.1.1.1.2 Angular

The angular unit of measure is modified using `-uAngular` as the parameter. It has the following format:

`-uAngular = name, scale [,decimal places]`

- *name* – a text string that should be displayed when showing an angular value.
- *scale* – a value multiplied with the base unit of measure to convert it to the specified unit of measure. The base unit of measure is radians.
- *decimal places* - The number of decimal places to be displayed when the value is displayed. This parameter is optional. If not specified, the number of decimal places is computed base on the scale.

For example: if the tracker pad should show degrees for angular values, the following would be placed in the parameter string:

`-uAngular = degrees, 57.29577951`

To specify that 6 decimal places are to be used with the above example, the following would be placed in the parameter string:

`-uAngular = degrees, 57.29577951, 6`

8.1.1.1.3 Pressure

The pressure unit of measure is modified using `-uPressure` as the parameter. It has the following format:

`-uPressure = name, scale [,decimal places]`

- *name* – a text string that should be displayed when showing a pressure value.
- *scale* – a value multiplied with the base unit of measure to convert it to the specified unit of measure. The base unit of measure is millimeters of mercury (mm Hg).
- *decimal places* - The number of decimal places to be displayed when the value is displayed. This parameter is optional. If not specified, the number of decimal places is computed base on the scale.

For example: if the tracker pad should show cm Hg pressure values, the following would be placed in the parameter string:

`-uPressure = cm Hg, 0.1`

To specify that 4 decimal places are to be used with the above example, the following would be placed in the parameter string:

`-uPressure = cm Hg, 0.1, 4`

8.1.1.1.4 Temperature

The angular unit of measure is modified using `-uTemperature` as the parameter. It has the following format:

`-uTemperature = name, scale, offset [,decimal places]`

- *name* – a text string that should be displayed when showing a temperature value.
- *scale* – a value multiplied with the base unit of measure to convert it to the specified unit of measure. The base unit of measure is degrees Celsius.
- *offset* – a value to be added after the scale is applied.

- *decimal places* - The number of decimal places to be displayed when the value is displayed. This parameter is optional. If not specified, the number of decimal places is computed base on the scale.

For example: if the tracker pad should show Fahrenheit for temperature values, the following would be placed in the parameter string:

-uTemperature = Fahrenheit, 1.8, 32.0

To specify that 4 decimal places are to be used with the above example, the following would be placed in the parameter string:

-uTemperature = Fahrenheit, 1.8, 32.0, 4

8.2 Tracker Pad

The tracker pad is a dialog that allows a user to control the tracker.

The application is started using the name TrackerPad. An example would be as follows:

```
trk.startApplicationFrame("TrackerPad", "");
```

This application will attempt to acquire exclusive access to the tracker interface whenever one of its buttons is pressed. Therefore, any application that starts the tracker pad should use the busy event and disable access to the tracker whenever the event indicates that that interface is processing a command.

The tracker pad attaches a busy listener when it is started. If a busy event is generated indicating that the interface is busy processing a command, the buttons of the pad are disabled.

8.2.1 Parameters

The following parameters can be passed in the parameter string to change the behavior of the tracker pad.

- -u (See section 8.1.1.1)
- -a
- -b

8.2.1.1 -a

The -a parameter is used to start one particular item on the tracker pad rather than starting the entire tracker pad. It has the following format:

-a *name*

name is the item that should be run directly. The following are valid names:

- SEARCH_CFG – Brings up the search dialog. This is the same dialog that would appear if the user pressed “Search” on the tracker pad.
- SET_DISTANCE – Brings up the set distance dialog. This is the same dialog that would appear if the user pressed “Set Distance...” on the tracker pad.
- VISUAL_DRIVE – Brings up the visual drive display directly.
- ANGULAR_DRIVE – Brings up the angular drive display directly.
- MANUAL_DRIVE - Brings up the manual drive display directly.
- CAMERA_DRIVE – Brings up the drive display that uses the TrackCAM (available only on trackers with this option installed. **Also the tracker must be in front sight mode and need to have the distance measuring mode set to either ADM only mode or IFM set by ADM mode.**

Other options could be appended to the CAMERA_DRIVE option as explained in the section Camera Drive options.

- FOLLOW_ME – Brings up the Follow Me settings dialog. This is for Vantage S/E trackers and Vantage S6/E6 trackers. Vantage S/E trackers need firmware versions 1.2.0 or later.
- PROBE_MANAGEMENT – Brings up the probe management dialog. This is only for Vantage S6/E6 trackers.
- PROBE_COMPENSATION – Brings up the probe compensation dialog. This is only for Vantage S6/E6 trackers
- PROBE_CALIBRATION – Brings up the probe calibration dialog. This is only for Vantage S6/E6 trackers

When the -a parameter is used, the -b parameter is ignored.

8.2.1.2 -b

The -b parameter is used to add or remove button from the tracker pad. It has the following format:

-b *buttonname* = *state*

- *buttonname* – The name of the button to be added or removed.
- The table shows the buttons supported by trackers. * is supported. Blank not supported
- The name in the first column (Button Name) should be used to add or remove a button.
- The name in the second column (Button that will be added or removed) is the name displayed on the Tracker pad.

| Button Name | Button that will be added or removed | 4xxx | FARO | Vantage | Vantage S/E | Vantage S6/E6 |
|-----------------|--|------|------|---------|-------------|---------------|
| Initialize | Initialize | * | * | * | * | * |
| Home | Home | * | * | * | * | * |
| MotorState | Motors On/Off | * | * | * | * | * |
| TrackingMode | Tracking On/Off | * | * | * | * | * |
| Backsight | Backsight | * | * | * | * | * |
| SearchT | Search | * | * | * | * | * |
| HighlightBeam | Find Beam | * | * | * | | |
| DriveOption | Visual Drive | * | * | * | * | * |
| DistMode | Distance Mode | * | * | * | * | * |
| SetTarget | Set Target | * | * | * | * | * |
| SetWeather | Weather Settings | * | * | * | * | * |
| TempConfig | Temperature Sensor Configuration | * | * | * | | |
| SetTime | Set Time | * | * | * | * | * |
| ShutterState | Shutter Open/Close | * | | | | |
| SetDistance | Set Distance | * | * | * | * | * |
| ChangeIPAddress | Set IP Address(4000 and ION) Change Network Settings(Vantage, Vantage S/E, Vantage S6/E6) | * | * | | | |
| ChangeAdminPwd | Change Administrators Password | * | * | * | * | * |
| BubbleLevel | Display Bubble | | | * | * | * |

| | | | | | | |
|------------------|---------------------|--|--|---|---|---|
| | Level | | | | | |
| ReleaseLock | Release Lock | | | * | * | * |
| *Wireless | Turn Wireless On | | | * | * | * |
| Gesture | Turn Gesture On | | | | * | * |
| PowerState | Enable Power Button | | | | * | * |
| Reboot | Scheduled Power On | | | | * | * |
| *FollowMeSetting | Follow Me Settings | | | | * | * |
| ProbeManagement | Probe Management | | | | | * |

*FollowMeSetting is available for Vantage S/E trackers and Vantage S6/E6 trackers. Vantage S/E trackers need firmware version 1.2.0 or later.

*Turn Wireless On need firmware version 2.6.4 and later for Vantage S/E trackers and Vantage S6/E6 trackers.

state – set to on to turn a button on or set to off to turn the button off.

For example, to remove the “Set Target Type” and “Distance Measurement Mode...” buttons, the following would be added to the parameter string:

-b SetTarget = off -b DistMode=off

The state parameter values that can either be “off” or “on” are case insensitive.

8.2.1.3 Camera Drive options

8.2.1.3.1 Vantage Tracker

To launch Camera Drive on top of other windows the parameter to specify is `-c alwaysOnTop = on/off`.

To launch with a Cancel button the parameter is `-c Cancel = on/off`.

By default, the options are off. Please see the example below on how to turn them on. Note, these options can be combined as shown in the second example.

For Ex:

To launch Camera Drive on top:

```
trk.startApplicationFrame("TrackerPad", "-a Camera_Drive -c alwaysontop = on);
```

To launch Camera Drive on top and with a cancel button:

```
trk.startApplicationFrame("TrackerPad", "-a Camera_Drive -c alwaysontop = on -c Cancel = on);
```

Clicking the Cancel or the Close button closes the Camera Drive dialog. The difference between choosing either of the buttons can be seen in the application exit code returned when Camera Drive is launched directly using the `startApplicationXXX()` API calls.

8.2.1.3.2 FARO Laser Tracker

8.2.1.3.2.1 To set a custom title to Camera Drive window

-T Camera_Drive_Title = Custom String will set append the Custom String to the title of the Camera Drive window.

8.2.1.3.2.2 Customize fields to be shown to user

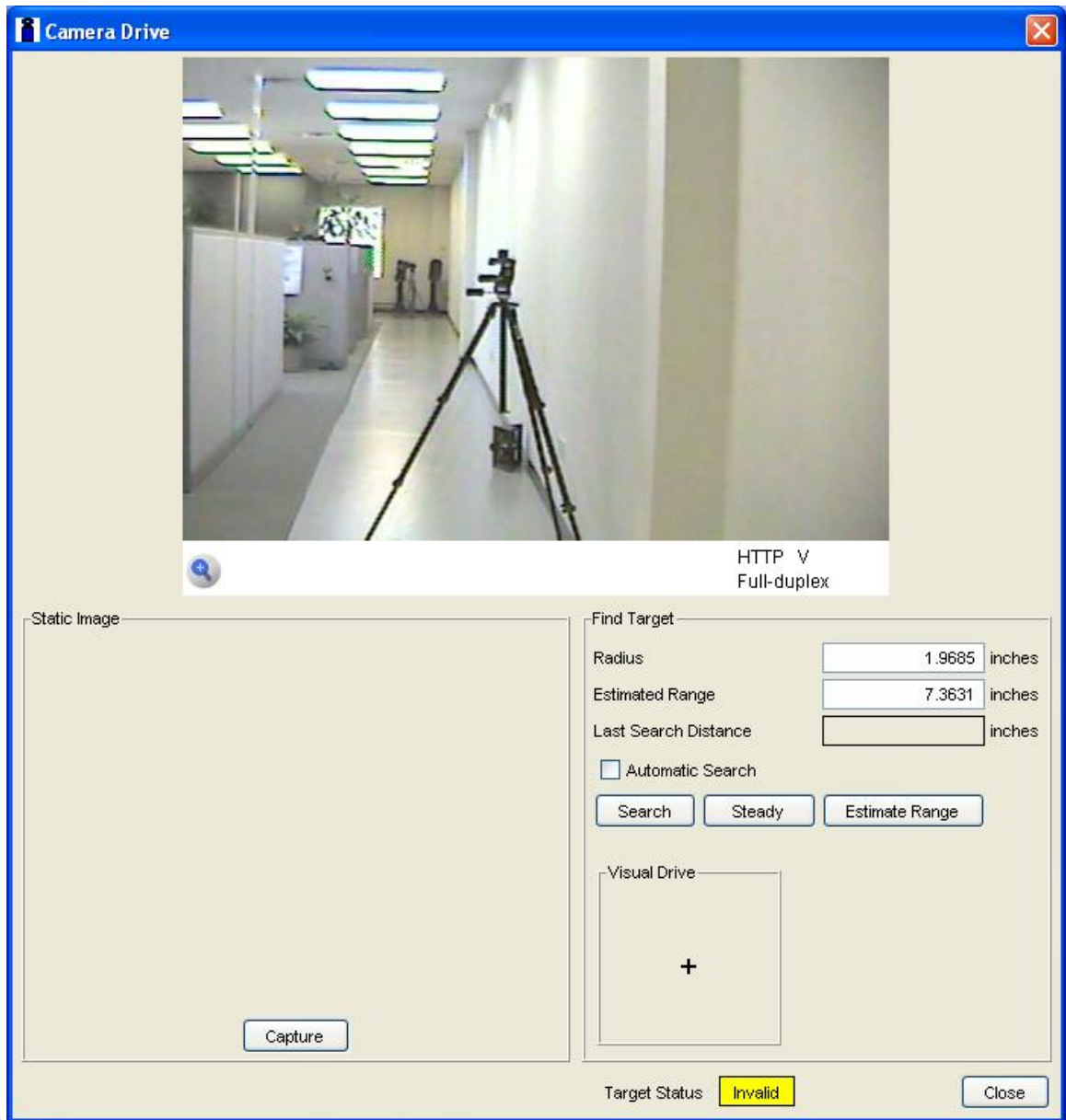
The UI is divided into 3 main components and the calling application can choose to show all the components or any combination of them. The three main components would be

- a) The dynamic display
- b) The static portion of the display with the capture button
The static portion can be customized to save a bitmap (a .BMP file, always a 352 x 240 pixels area) to a specified local hard drive every time the CAPTURE button is clicked.
- c) The controls portion of the main UI which has all the user entry fields and buttons along with the visual drive and close buttons.

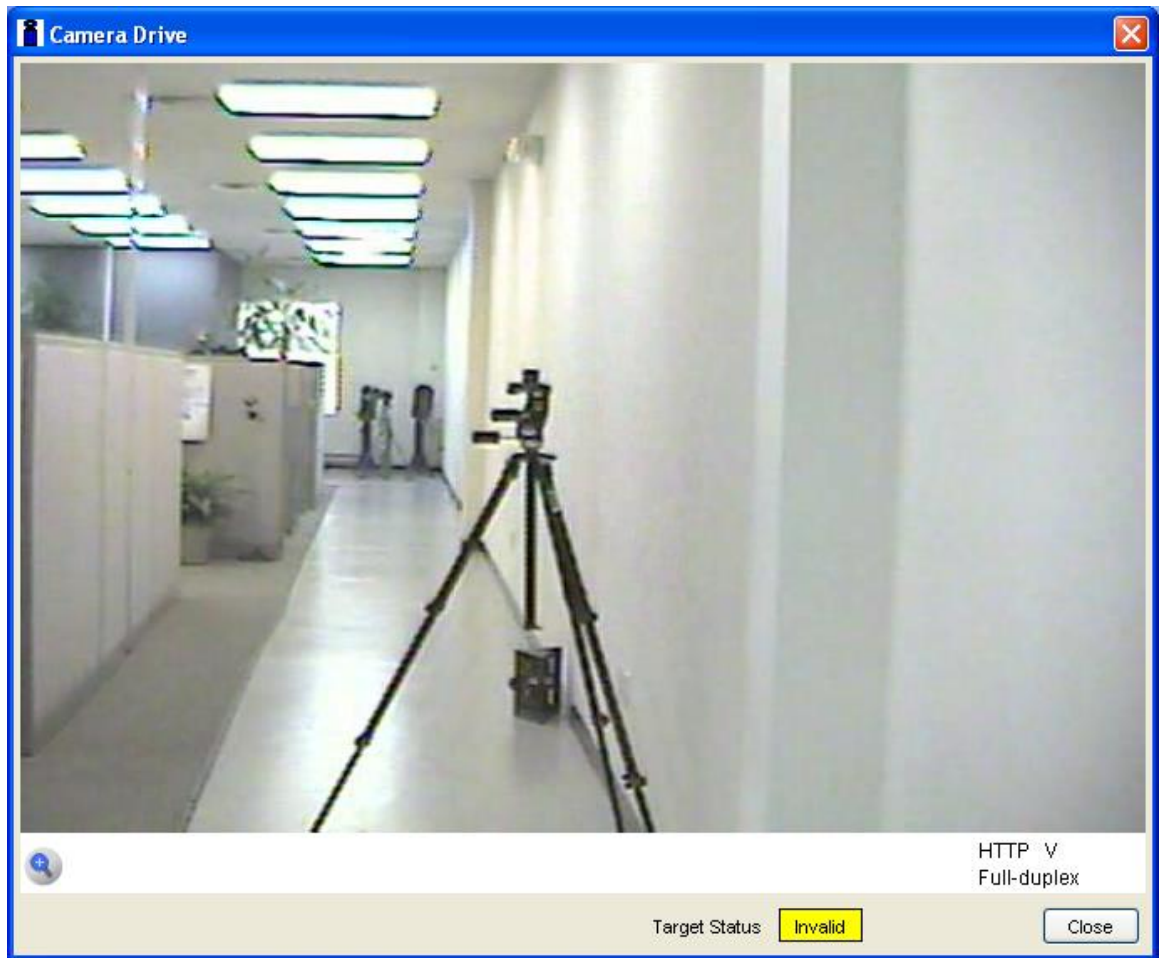
The calling application can further customize the controls display component by choosing to hide any or all the fields. If all the controls are hidden then it would be treated as if the whole controls display is chosen to be hidden. The controls component resizes automatically depending on the number of controls displayed or hidden.

Also once the TargetCAM application is started, it can programmatically be customized. For example, say that initially the TargetCAM application had only the dynamic display shown. If later it is desired to add the static component or say a few controls of the control display (like adding only the search radius field to it) then it will be possible to achieve this without closing and opening the TargetCAM application. This helps with keeping the real estate space on the screen for the TargetCAM application to a minimum. The dynamic change in the UI is programmatically achieved by calling a method **modifyApplication()** with appropriate parameters.

When all the three components of the display are shown, then the display would as shown below. This is the default view for Camera Drive when no options are specified.



When the dynamic component alone is shown, then the UI looks as shown below. This is achieved by specifying “-dStatic=off-dcontrol=off”. (Note that the dynamic display is on by default and hence there is no need to turn it on explicitly)



When the dynamic and the static components are displayed, then the UI looks as shown below. This is achieved by specifying “-dStatic=on”. Note that there is no need to turn on the dynamic or turn off the controls display. The options have a cumulative affect.



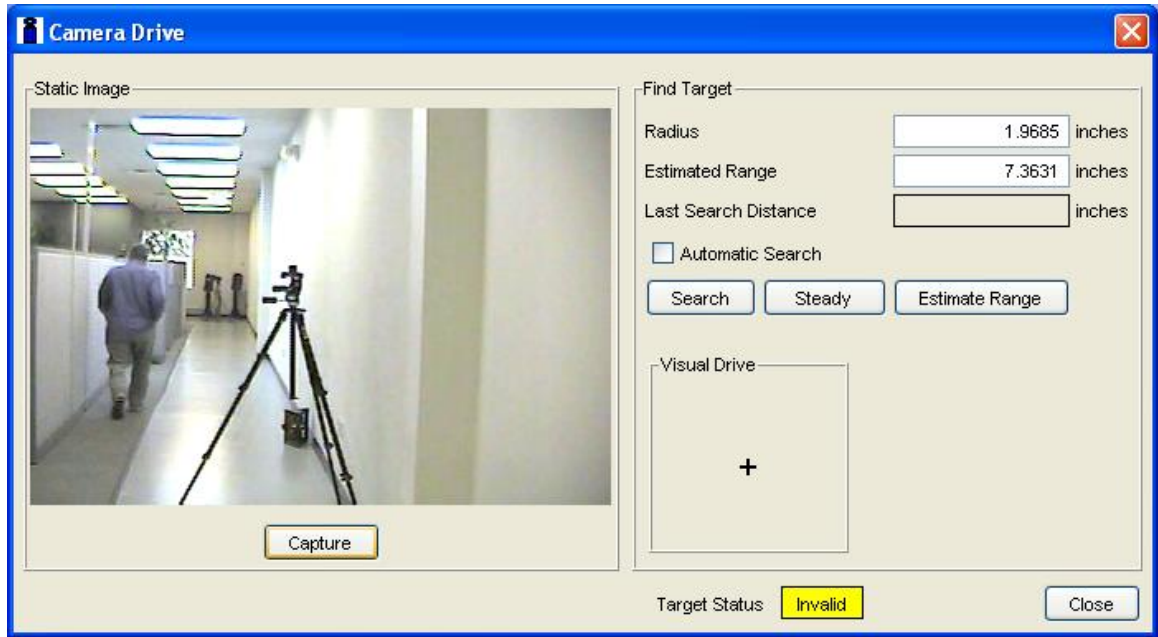
When dynamic and control components are the only ones shown, the display looks as shown below. This is achieved by specifying “-dStatic=off -dcontrol=on”.



When only the radius and the Automatic Search controls are shown on the control component along with the dynamic display component, the UI looks as shown below. This is achieved by specifying "-cRange=off -cSearch=off -cSteady=off -cRange=off -cVisual=off".



When the static and controls display are the only ones, then the UI looks as shown below. This is achieved by specifying "-cRange=on -cSearch=on -cSteady=on -cRange=on -cVisual=on -dDynamic = off -dStatic = on".



The option to turn off the component displays or the option to hide buttons on the control component display has to be passed at the time of calling startApplicationXXX method.

So here is a summary of all the various options to turn off component displays.

- D Dynamic = off
- D Static = off
- D Controls = off

The options to hide the buttons on the control component display would be

- c Radius = off
- cRange = off (hides the Estimated Range text field)
- cLast = Off (hides the Last Search Distance text field)
- cAutoSearch = off
- c Search = off
- c Steady = off
- cEstimate = Off (hides the Estimate Range push button)
- cVisual = Off
- cRadius = Off

- cClose = Off
- cTarget_Status = Off

8.2.1.3.2.3 Populate all fields programmatically (for the controls component UI portion)

This adds the capability to programmatically populate any fields that are currently exposed to the user. This includes the programmatically setting the values for the text fields (radius and estimated distance) and setting the checked/un-checked state for the Auto Search toggle button. Note that these settings will be used even when the controls themselves are hidden.

The option to programmatically set the values for the user input fields for the control component display has to be passed at the time of calling startApplicationXXX method.

The options would be

-V Search = 0.05
-V Range = 2.25
-V AutoSearch = Off
-V AutoSearch = On

Note that all the values passed would be in **meters** and in **locale specific format** (precision is 3).
So in German, it would be for example –V Search = 0,05

8.2.1.4 Probe Management

- SDK 5.1.0 is required for SixProbe 1.0 feature. SDK 5.1.9 and firmware 2.11 is required for SixProbe 2.0 feature.
- The below documentenation has been documented using SDK 5.1.9 and firmware 2.11.
- The dialog is available only if tracker is sixdof capable. The dialog can be launched from Tracker Pad by selecting the button “Probe Management”. The dialog can be also be launched directly from SDK as below:

```
trk.startApplicationFrame(“TrackerPad”, “-a probe_management”);
```
- Using this interface, users can can turn the sixdof feature on/off, add, modify, compensate and calibrate the probe.
- When the dialog is launched for the very first time, Enable SixDOF is not selected as in the below figure 1. By default, the sixdof feature is disabled. Once enabled the setting is persistent through power cycles.

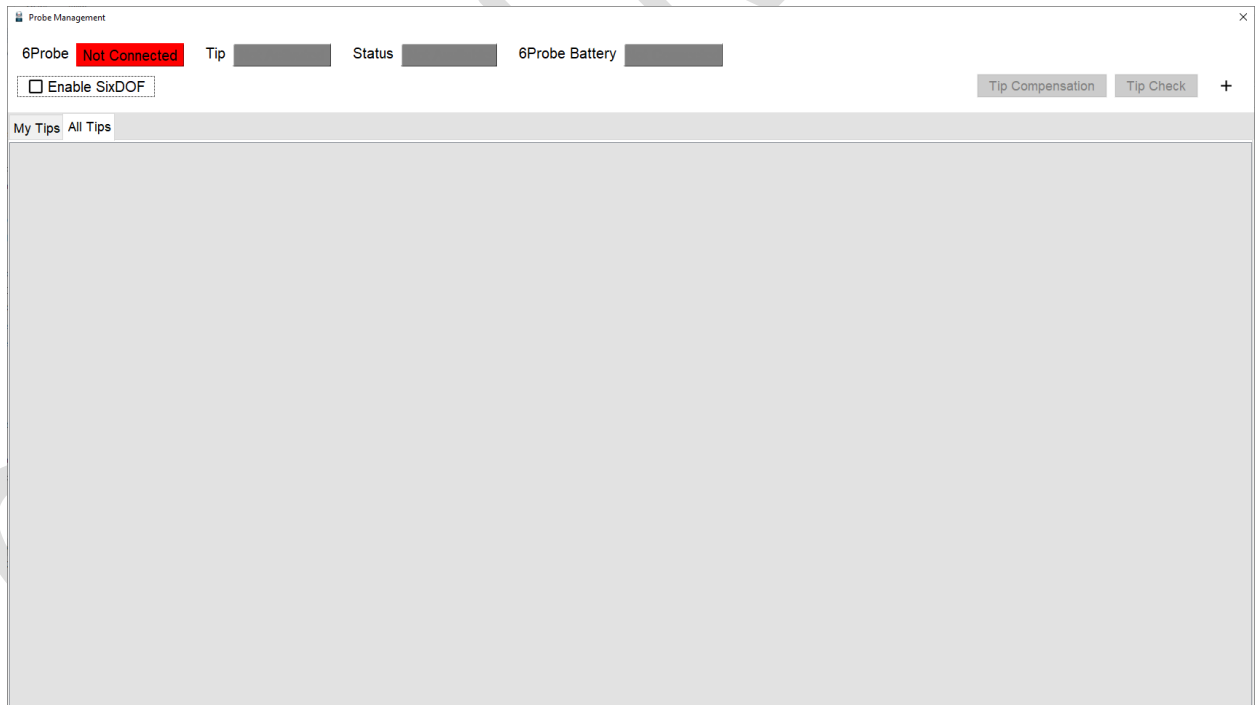


Figure 1

- Select the “Enable SixDof” checkbox and connect to a sixprobe. See section [1.4.7](#) on how to connect to the sixprobe as in Figure 2.

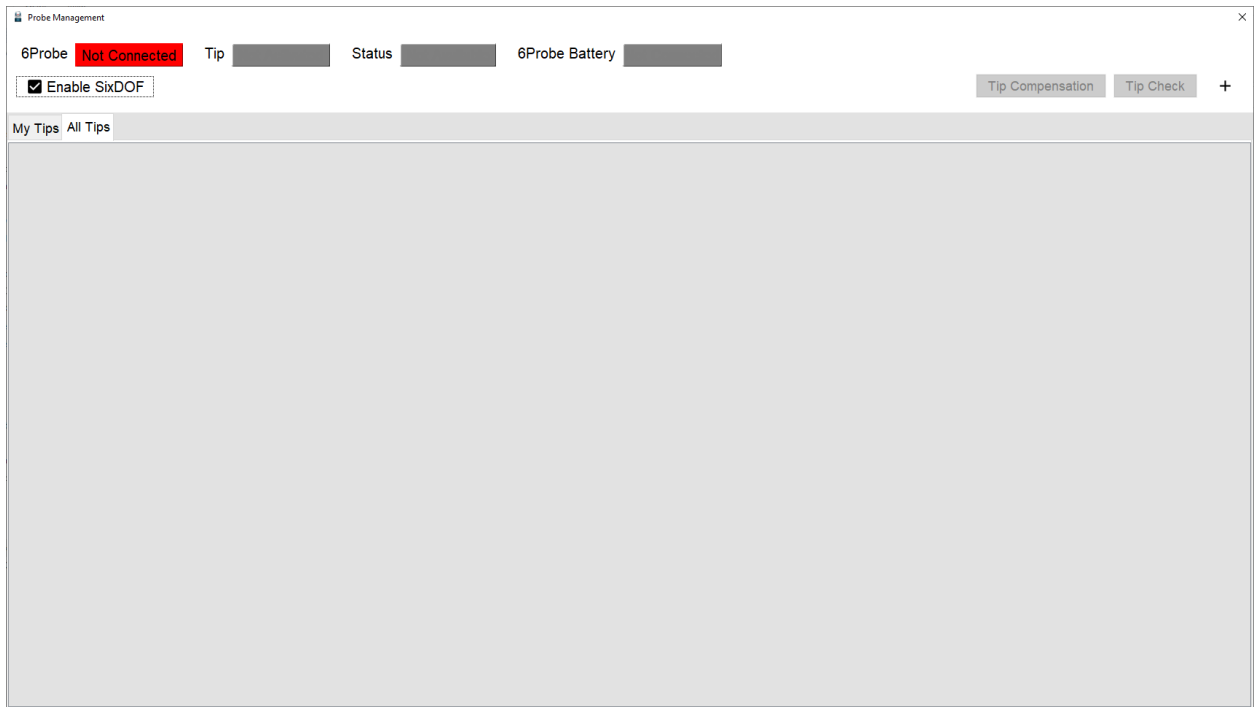


Figure 2

- When connected to the sixprobe, the dialog is updated with the the serial number of the sixprobe, if any tip is activated or not, status and the battery charge remaining.

8.2.1.4.1 Six Probe 1.0

If connected to SixProbe 1.0

- The dialog is populated with all the standard tips as in the below figure 1. All the standard tips shown in the dialog act as templates to create new tips.

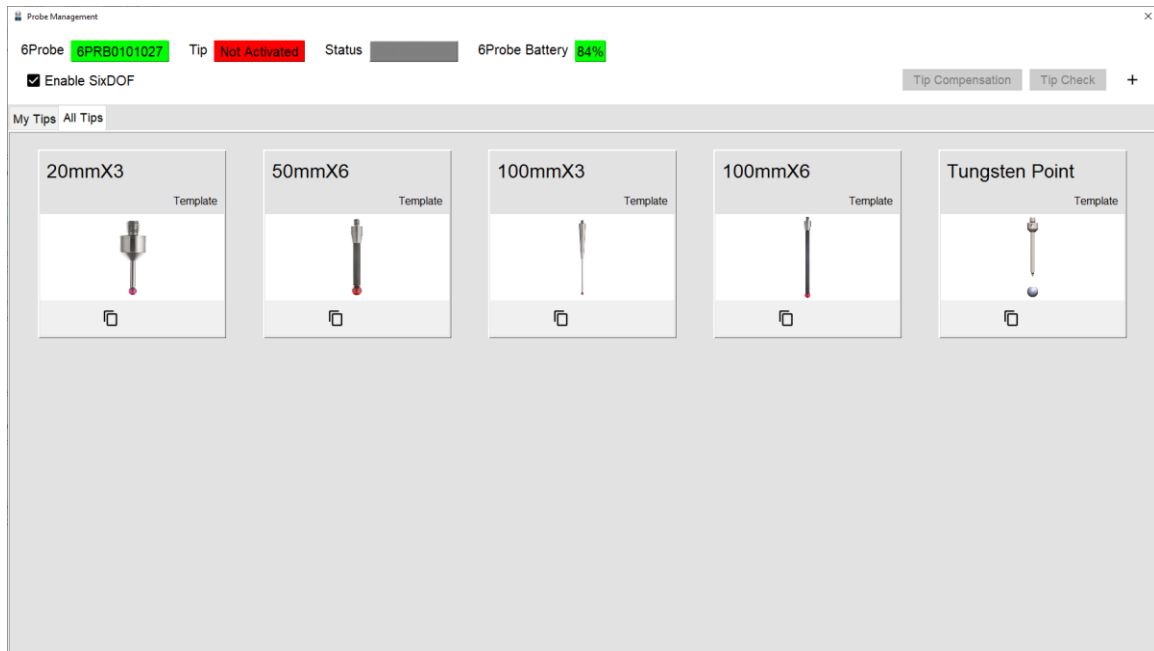




Figure 3

- On launching the dialog, if the tracker was already connected to a Six Probe 1.0, then the dialog will be populated with all the tips created.
- Please note that the tips are added and removed when connected and disconnected to the sixprobe with the dialog open.
- To create a tip, select plus  icon on the upper right side of the dialog or press the clone  button on one of the existing templates as shown in Figure 3.

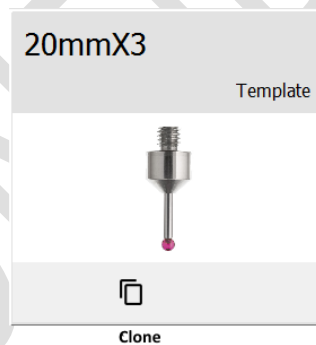


Figure 4

- When the plus icon is selected, a dialog as shown in Figure 4 pops up. Specify the name and diameter of the tip and click save on the dialog. If the clone button is selected, then the dialog is populated with the diameter of the template tip.

Hard Tip

Information

Name
Demo

Diameter (m)
0.006000

6Probe
6PRB0101027

☐ Set to Zero Length Tip

Dimensions

X (m)
0.000000

Y (m)
0.000000

Z (m)
0.000000

Compensation

Temperature (°C)
0.00

Date
Never Compensated

Save

Cancel

Figure 5

- Clicking save on the dialog will create “Demo” tip which we call as custom tip. The dialog will be updated with the newly create tip as in Figure 5.

6Probe

6PRB0101027

Tip

Not Activated

Status

6Probe Battery

84%

☒ Enable SixDOF

Tip Compensation

Tip Check

+

My Tips

All Tips

20mmX3

Template

50mmX6

Template

100mmX3

Template

100mmX6

Template

Tungsten Point

Template

Demo

6PRB0101027

Available

Figure 6

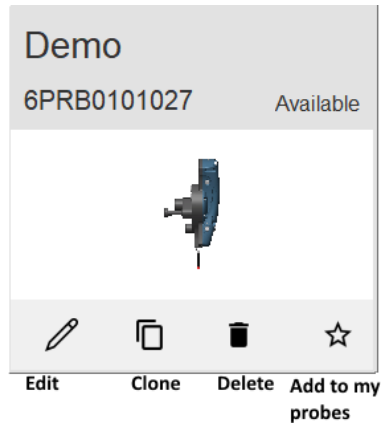


Figure 7

- A custom tip can be edited to modify any of its parameters (Name and Diameter). It can be removed by selecting the delete button. A new tip can be created from it by using the clone button.
- Please note:
 - Tip list is unique to each tracker.
 - A tip created for a sixprobe can only be used with it. For ex: if a tip is created for sixprobe with serial number 6PRB0010003, the tip can be used only with that sixprobe.
 - When the tip says “Available”, it means that the tip has been created using this sixprobe and can be used with it.
 - If connected to a sixprobe on launching the dialog, it is populated with the tips created using this sixprobe.
- After creating a new tip, the tip must be compensated to be used. If a tip is not compensated, it will affect the accuracy of the measurements. To compensate, the tip must be activated. To activate, hover the mouse on the tip. You will see that the text on the tip will change from “Available” to “Click to Activate” as can be seen below in Figure 7.

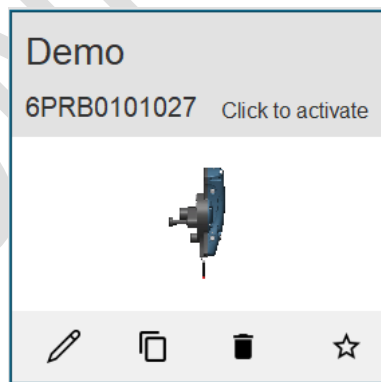


Figure 8

- Click on “Click to Activate” and the tip will be activated. The tip will now indicate that it has been activated as below Figure 9.

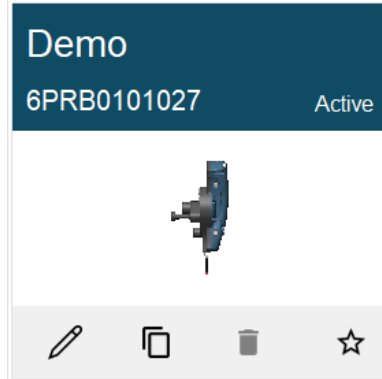


Figure 9

- Click to select the tip and click the “Tip Compensation” button on the upper right corner of the dialog. Follow the instructions on the dialog to compensate the tip. When compensation is done, the compensated tip vector needs to be updated. When updated, the status on the dialog will be updated as valid indicating that the tip is ready to be used as in Figure 10.

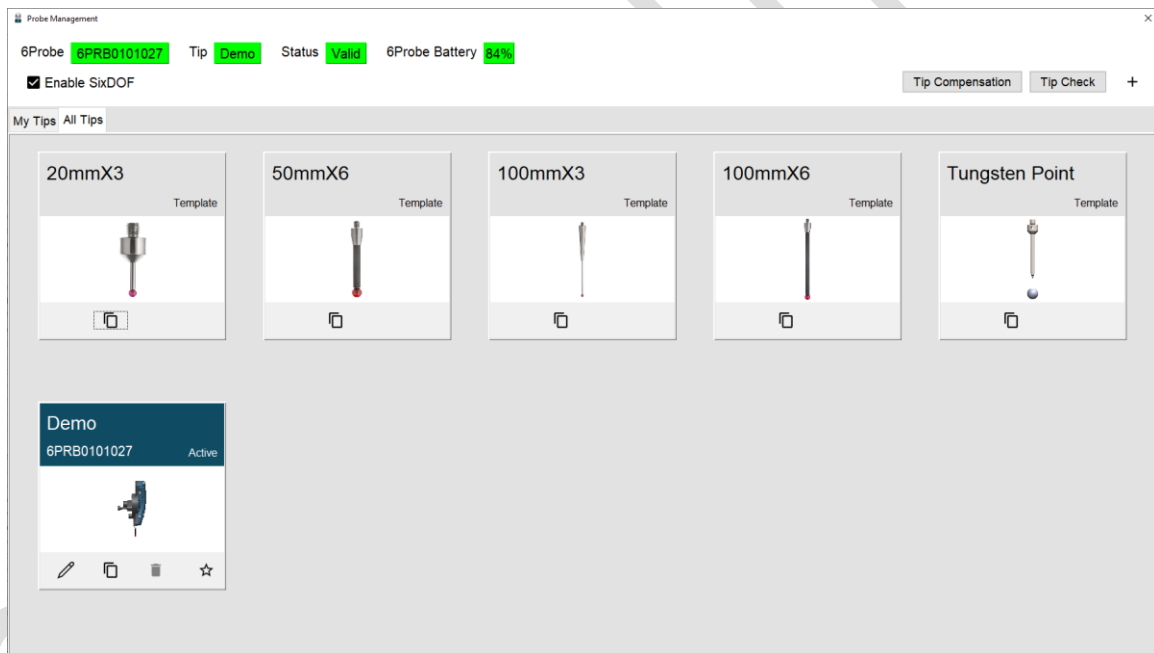


Figure 10

8.2.1.4.2 Six Probe 2.0

Six Probe 2.0 supports the kinematic tips. If a compensated kinematic tip is connected, then the sixprobe is ready to use.

If connected to Six Probe 2.0

- When a kinematic tip is plugged in, it is automatically detected. The dialog shows the tip with default parameters as in Figure 11.

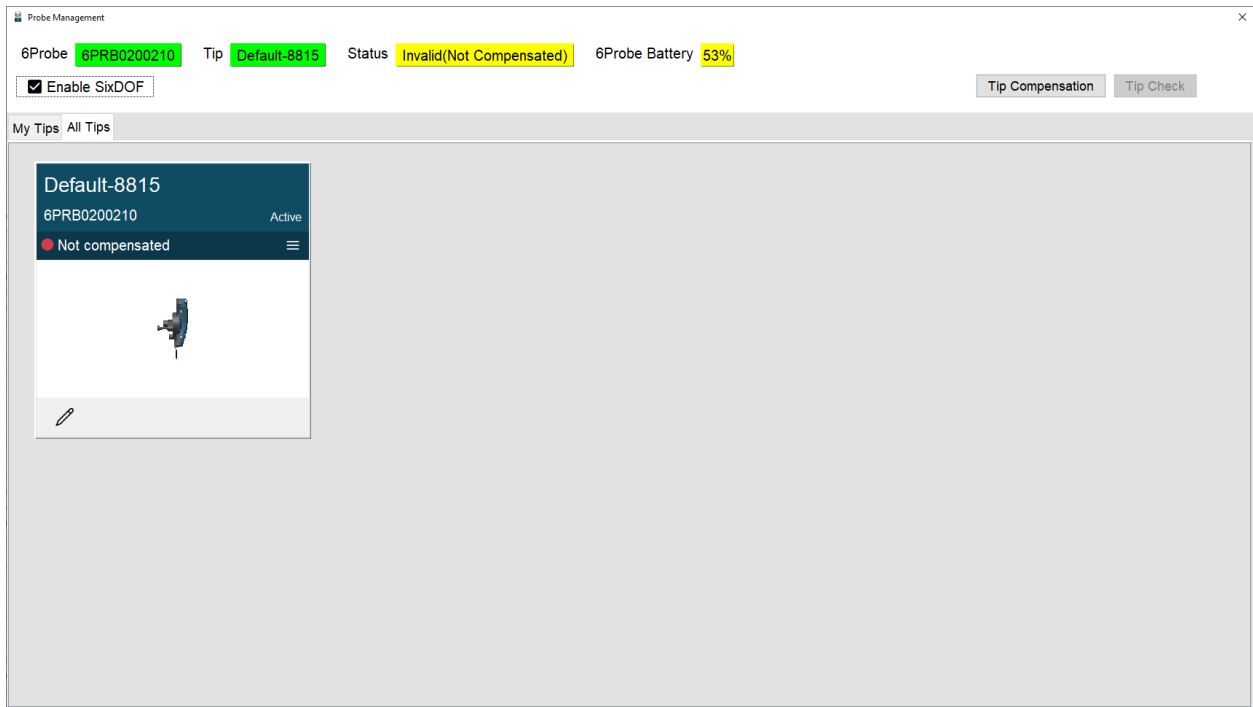


Figure 11

- To be able to use this tip, users need to compensate it. Select Tip compensation button on the dialog to run the compensation. When the button is selected, a dialog is provided to specify the name and diameter of the tip as in the below Figure 12.

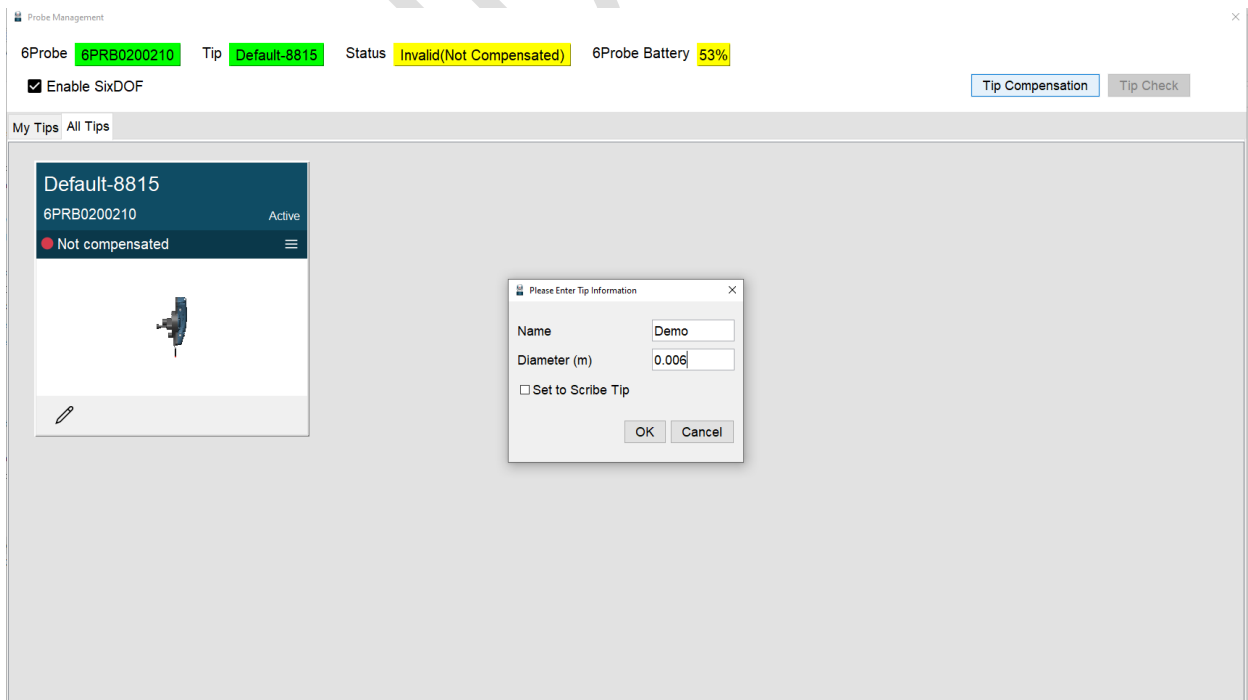


Figure 12

- Selecting ok on the dialog after providing the name and diameter as shown in the Figure 13, tip compensation is launched.

Please Enter Tip Information

Name: Demo

Diameter (m): 0.006

☐ Set to Scribe Tip

OK Cancel

Figure 13

- Please follow the instructions to run the compensation. When compensation is successfully completed, the tip gets added and the status of the tip becomes active as can be seen in Figure 14.

Probe Management

6Probe: 6PRB0200210 Tip: Demo Status: Valid 6Probe Battery: 51%

☒ Enable SixDOF

Tip Compensation Tip Check

My Tips All Tips

Demo
6PRB0200210 Active
2020-10-13 16:33:47 0.032 mm

Figure 14

- Select the button on the tip to see the details of the compensation as in Figure 15.

Demo

6PRB0200210 Active

Latest Comp Log X

2020-10-13 16:33:47 0.032 mm

View All 1 of 20 tip spaces used

Figure 15

- If the tip consists of more than one compensation, it will be displayed as in Figure 16

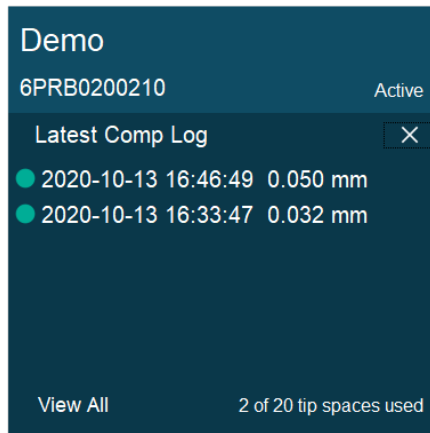


Figure 16

- To view all the compensation history, Select “View All”. A dialog with all the compensation history is launched as can be seen as in the Figure 17.

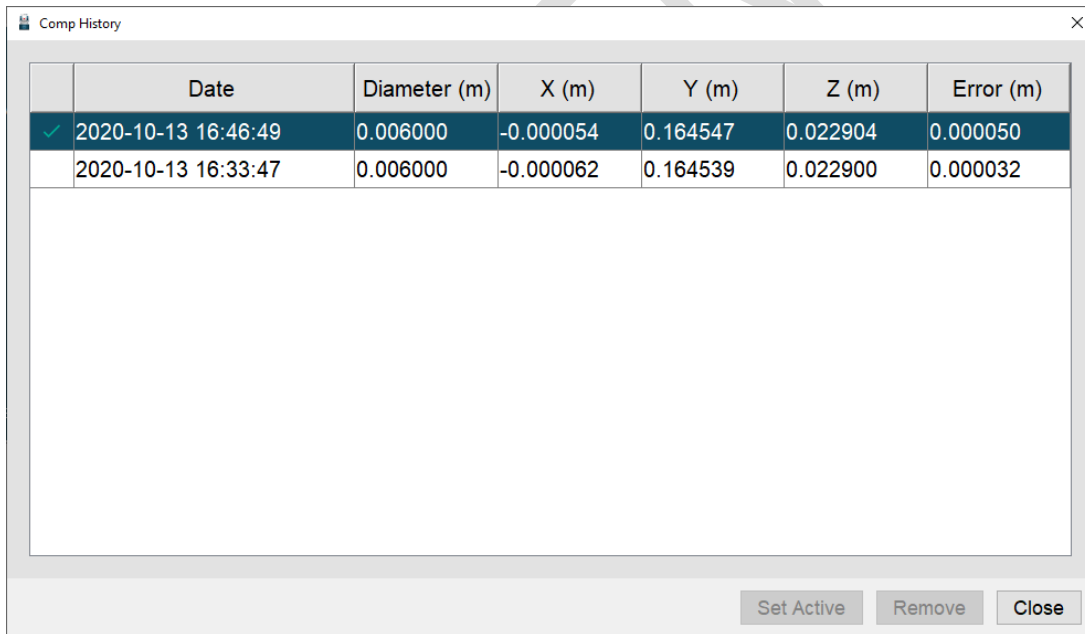




Figure 17

- As can be seen in the dialog, the compensation that is currently in use is marked with .
- Users also have the option to select which compensation to use by selecting the “Set Active” button.
- Users also have the option to remove a compensation that is no longer required using “Remove” button.
- The users have the option to modify the tip name or diameter. Select  on the tip. This will launch a dialog to modify the parameters as shown in the below figure 18.

Hard Tip

Information

Serial Number: 316238815

Name: Demo

Diameter (m): 0.006000

6Probe: 6PRB0200210

☐ Set to Zero Length Tip

☐ Set to Scribe Tip

Dimensions

X (m): -0.000062

Y (m): 0.164539

Z (m): 0.022900

Compensation

Temperature (°C): 22.12

Date: 2020-10-13 16:33:47

Save Cancel

Figure 18

- The Probe Compensation dialog can be launched directly from SDK. If connected to SixProbe 1.0, to launch the dialog the tip must be activated first. If connected to SixProbe 2.0, kinematic adapter tip must be connected. To launch the dialog, use the below method.

```
trk.startApplicationFrame("CompIT", "-a PROBE_COMP");
```
- If the tip has already been compensated, click on the tip to use it.
- A tip can also be calibrated to verify if the compensation of the tip is still good and would give right results. To calibrate the tip select the "Tip Calibration" button which also on the upper right corner of the dialog. Follow the instructions on the dialog to calibrate the tip.
- The dialog can be launched directly from SDK. To launch the dialog the probe must be activated first. To launch the dialog, use the below method.

```
trk.startApplicationFrame("TrackerPad", "-a probe_calibration");
```

8.3 Closure

The closure is an application used to check closure at TMR location or against a specific point location. Below is its basic appearance. dR is the difference between the distance of TMR read from tracker and the distance measurement of the target if checking closure against TMR. If checking closure against a point, dR will show the difference between the distance measurement of a point and the distance value passed in by user.

The application is started using the name Closure. An example would be as follows:

```
trk.startApplicationFrame("Closure", "");
```

This application will start a factory background measurement. Therefore, any other application that starts a factory background measurement will not be able to run at the same time.

8.3.1 Parameters

The following parameters can be passed in the parameter string to change the behavior of the tracker pad.

- -d
- -b

Each is described in the following subsections.

8.3.1.1 -d

The -d parameter can be used to change the location used to check closure against. It has the following format:

8.3.1.1.1 -d TMR

- used to indicate that the closure should be checked against TMR location.

8.3.1.1.2 -d distance pointname

- distance – a value in meters used to calculate distance difference.
- pointname – a text string indicating the point that will be checked closure against. It will be displayed on the right status bar on the closure dialog.

For example: if check closure against point2 3 meters away from tracker, the following would be placed in the parameter string:

-d 3 point2

8.3.1.2 -b

Closure dialog hides Close button from UI display by default. The -b parameter is used to add or remove the Close button from the closure dialog. It has the following format:

-b Close = state

- state – set to “on” to show Close button or set to “off” to hide Close button.

The state parameter values that can either be “off” or “on” are case insensitive.

8.4 CompIT

The CompIT is an application that allows a user to perform different compensation and interim test procedures.

This application will acquire exclusive access to the tracker interface after it is up running. Therefore, the interface will not be able to run any other application at the same time.

8.4.1 FARO Tracker Version

The application is started using the name CompIT. An example would be as follows:

```
trk.startApplicationFrame("CompIT", "");
```

8.4.1.1 Parameters

The following parameters can be passed in the parameter string to run quick backsight alone.

- -a QBC

8.4.2 4xxx version

8.4.2.1 CompIT

The application is started using the name CompIT, which will run CompIT.exe under the bin directory. An example would be as follows:

```
trk.startApplicationFrame("CompIT", "");
```

8.4.2.2 ADM CompIT

The application is started using the name ADMComp, which will run ADMComp.exe under the bin directory. An example would be as follows:

```
trk.startApplicationFrame("ADMComp", "");
```

This ADM CompIT is for the 4000 series trackers only and the ADM Comp is part of CompIT in Keystone trackers.

8.5 Startup Checks

The startup checks is an application used to check readiness of certain devices in the tracker before the tracker is ready for use. If a device is not ready, the application waits.

The application is started using the name StartupChecks. An example would be as follows:

```
trk.startApplicationFrame("StartupChekcs", "");
```

This application will start a factory background measurement and also acquire exclusive access to the tracker interface. Therefore, the interface will not be able to run any other application at the same time.

8.6 MeasurePad

The Measure Pad application is a simple data collection tool that can be used to gather target data from the tracker. In addition to being able to write data to a file, the application can interface to other applications through a socket used a defined protocol.

A protocol has been established to receive data from "Holos NT" software (transformation matrices), using an entry COUNTER 9999/tcp (any *non reserved number* can be used instead of 9999, for example don't use 21 since it is reserved for FTP) in the C:\winnt\System32\drivers\etc\services file.

8.6.1 Starting the application

8.6.2 Starting parameters

The "Measurepad" application takes starting parameters in form of a concatenated string.

8.6.2.1 Units for length

The units by default are meters for length (equivalent to specifying "-uLength=m, 1, 6") unless specified by string say for example "-uLength=mm, 1000, 3" where 'mm' or 'm' from the example stand for the description of the "length" unit. The number (1 or 1000 from the above example) is the conversion factor followed by the number (6 or 3 for the example) which is the precision level for the numbers i.e. the number of digits after the decimal.

8.6.2.2 Removing the "Close" or "Setup..." buttons

By concatenating the parameter string with "-bClose=off" and or "-bSetup...=off" the "Close", "Setup..." buttons can be removed from the application. The string "off" can be in any case like Off or OFF etc.

8.6.2.3 Specifying the port number for the socket

By concatenating the parameter string with "-hXXXX", the application creates a socket to listen on port specified by the number XXXX. For example to setup the socket connection on port 1234 on the local machine, we append "-h1234" to the parameter string.

8.6.3 Startup errors

Immediately after the application is initialized or in other words started, there is a chance that the application reports a fatal error and automatically closes down. The reason could be that there are other tracker applications running from the same session. For example, trying to launch MeasurePad after OperationChecks is already started from the same program would result in a fatal error for MeasurePad, since MeasurePad cannot start a factory background measurements which is already started by OperationalChecks. Note that it is not a problem if the applications are started from different programs.

8.6.3.1.1 Sending data

In order to save the measurement data to a file the user has to check the "To file" field in the Send data group. This enables the edit field (into which the user can type the name of the text file) and the "Browse..." button. Choosing the "Browse..." button brings up the common "File Save As" dialog with an option to append data. The user can choose to send the data to application software by checking the "To Socket" checkbox.

The data saved to a text file and or sent to the socket has the following format.

X,Y,Z,timeSince1970

The data sent to the socket has the following format.

A: X@Y@Z [I@J@K](#) ProbeDiameter (for measured data)

S: [X@Y@Z](#) (for target position data)

The above data is in user specified units for length.

If during a measurement the beam is broken, then the measurement is aborted automatically and the (X, Y, Z, I, J, K) display shows "*****" in red color.

8.6.3.1.2 Number of samples per point

This is the number of points averaged into one point for reporting the measurements.

8.7 HealthChecks

- “HealthChecks” simply tell the user if the tracker is ready to be used or not, like a Go-NoGo gauge. The health checks start running automatically without user intervention (for a 4xxx tracker, UI interaction could be required at the startup of the application, if ADM is present but disabled).

8.7.1 Starting parameters

The “HealthChecks” application takes starting parameters in form of a concatenated string.

8.7.1.1 Units for length

The units by default are meters for length (equivalent to specifying “-uLength=m, 1, 6”) unless specified by string say for example “-uLength=mm, 1000, 3” where ‘mm’ or ‘m’ from the example stand for the description of the “length” unit. The number (1 or 1000 from the above example) is the conversion factor followed by the number (6 or 3 for the example) which is the precision level for the numbers i.e. the number of digits after the decimal.

8.7.1.2 Removing the buttons

This option is not supported. If the -b option is provided in the startup parameter string, it is ignored.

8.7.1.3 Specifying the filename to store the detailed results

By concatenating the parameter string with “-fABCD”, the application writes the details of running the HealthChecks to the file “ABCD.txt”. Note that if no extension is specified the default .txt is added to the filename. Also the user has to specify the full path of the filename if the file is to be saved to a different folder other than the current folder from which the “HealthChecks” application is launched from, like for example “c:\\temp\\results.txt”.

8.7.2 Startup errors

Immediately after the application is initialized or in other words started, there is a chance that the application reports a fatal error and automatically closes down. The reason could be that there are other tracker applications running from the same session. For example, trying to launch HealthChecks while TrackerPad, which is started from the same program is doing a tracker initialization, would result in a fatal error for HealthChecks, since HealthChecks cannot grab exclusive access of the tracker. If another application, either the same application that is launching the HealthChecks or another application (same PC or a different PC) is issuing a tracker command, then starting HealthChecks at the same time would result in a fatal error.

8.8 Operational Checks

“OperationalChecks” are the checks that the operator performs to check the environment in his/her work volume.

8.8.1 Starting parameters

The “Operational Checks” application takes starting parameters in the form of a concatenated string.

8.8.1.1 Units for length

The units by default are meters for length (equivalent to specifying “-uLength=m, 1, 6”) unless specified by string say for example “-uLength=mm, 1000, 3” where ‘mm’ or ‘m’ from the example stand for the description of the “length” unit. The number (1 or 1000 from the above example) is the conversion factor followed by the number (6 or 3 for the example) which is the precision level for the numbers i.e. the number of digits after the decimal.

8.8.1.2 Removing the buttons

This option is not supported at all. So even if the `-b` option is provided in the startup parameter string, it is ignored.

8.8.1.3 Specifying the quick backsight option

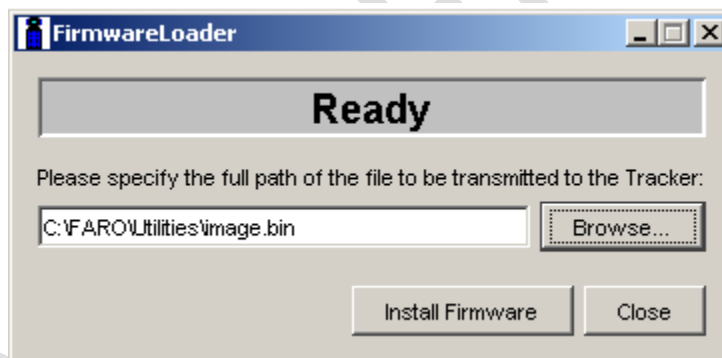
By concatenating the parameter string with “`-a qbc`”, the application is launched such that the Backsights page is the active page. By default the General page is the first page and the active page when the application is launched.

8.8.2 Startup errors

Immediately after the application is initialized or in other words started, there is a chance that the application reports a fatal error and automatically closes down. The reason could be that there are other tracker applications running from the same session. For example, trying to launch OperationalChecks after MeasurePad is already started from the same program, would result in a fatal error for OperationalChecks, since OperationalChecks cannot start a factory background measurement which is already locked by MeasurePad. Note that it is not a problem if the applications are started from different programs.

8.9 Firmware Loader

Firmware Loader allows the user to load new MCU firmware. The main display is shown below.

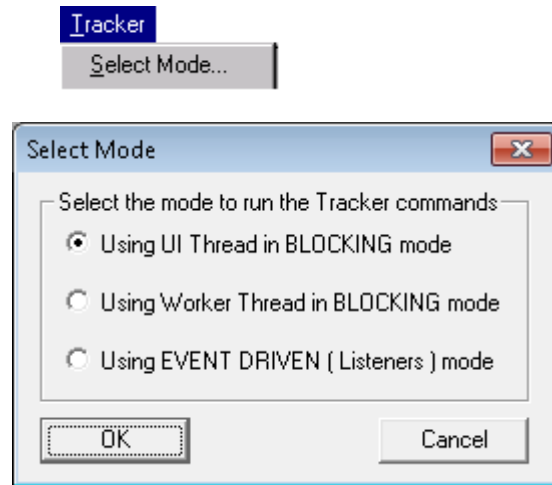


9 Tracker Driver Demo

- All the demos in the tracker development kit in 32 bit folder are 32 bit.
- TrackerDriverDemo is the old demo and does not support Vantage tracker. It is deprecated. TrackerDriverDemo_VC++2008 and TrackerDriverDemo_VC#2008 both are compiled in Visual Studio 2008 supports Vantage tracker. Both are deprecated.
- The latest demos are TrackerDriverDemo_VC++2017 and TrackerDriverDemo_VC#2017 in Visual Studio 2017 are always compiled with latest 32bit SDK. The executables are located in Release directory. Double click the executable to launch the demo application.
- The 32 bit libraries for C++ and C# are already copied into their respective project working directory. Launch the solution for your development needs.
- To use the project in Debug mode, launch the solution. Set the project configuration to Debug and recompile it.

9.1 TrackerDriverDemo_VC++

- To launch the demo application, double click on the executable in the release directory. When this opens up go to the Tracker menu and select the menu option “Select mode...”



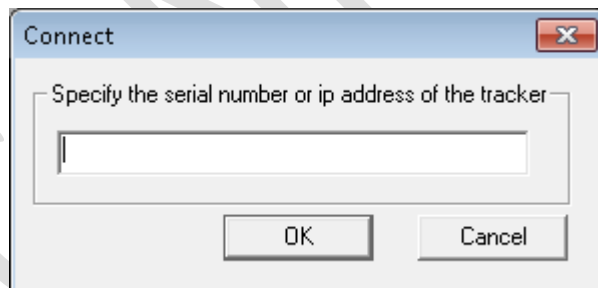
- This menu option will open up a dialog which is as shown below. Please select a mode to run the Tracker commands. If no mode is selected then the default will be “Run UI thread in Blocking mode.”
- Once the mode is selected, go to the Commands menu and select the menu option “Connect...”.

9.2 Connect

Menu Option: Commands => Connect...



Modes: This demonstration is shown in all three modes.



- This dialog provides the option to specify the tracker serial number or it's IP Address.
- Once the serial number or ipadress is specified and OK button is clicked it establishes connection to the Tracker.
- In Blocking mode, the command returns only after the connection is established to the Tracker.
- In Listeners mode, connect listener fires when the connection is completed

Tracker Commands used:

- Connect and Disconnect

Functions called in Demo:

- OnConnect ()

Event:

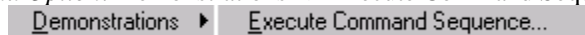
- Connection Event

Comments:

- Initially when the Tracker is not connected all the menu options are disabled except the connect menu option.
- Once the connection is established to the Tracker all the menu options are enabled and the connect menu option is disabled.
- If a connection is being tried to establish with a Tracker to which already a connection is established then it would throw Already connected exception.

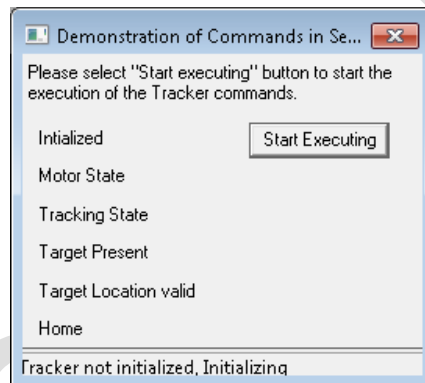
9.3 Execute Command Sequence

Menu Option: Demonstrations => Execute Command Sequence...



Modes: This demonstration is shown in all three modes.

When the dialog pops up it can be seen there stating the commands it is executing. When the button “Start Executing” is selected all the commands are executed one after another.



- In UI thread Blocking mode the command doesn't return until the Tracker completes executing the command
- In worker thread Blocking mode it will behave in the same way as Blocking in UI thread behaves except that Tracker commands are executed in a separate Worker Thread.
- In Listeners mode the command is submitted to the Tracker and returns immediately. Here the listener used is a command complete listener. Once the listener fires and says that no exception is thrown it means that the driver has successfully executed the Tracker command and can move on to the next command.
- When the commands are executing, the status bar on the dialog is updated informing which command the Tracker is currently executing and what is the result of the execution of each command i.e. whether the command was a success or not.

Tracker Commands used:

- Initialized
- Initialize
- Get Current Motor State
- Change Motor State

- Get Current Tracking State
- Target Location Valid
- Target Present
- Home

Functions called in Demo:

- ExecuteCommandSequence () in all the modes.

Event:

- Command Complete Event

Comments:

The advantage of running the commands in a separate worker thread is it keeps the UI free.

If the commands are executed in UI thread blocking mode the UI is frozen. If that is not a concern then the commands can be executed in that mode. However, if this is a concern, the best options would be to run the commands in a separate worker thread or to use listeners.

9.4 Asynchronous messages

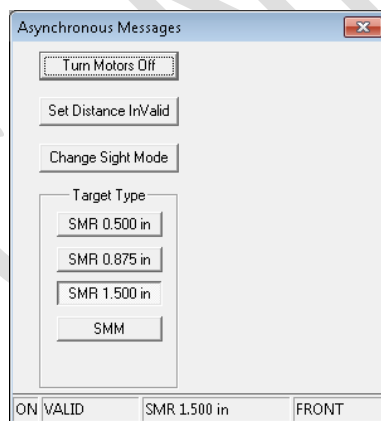
Menu Option: Demonstrations => Asynchronous Messages...

Demonstrations ► Asynchronous Messages...

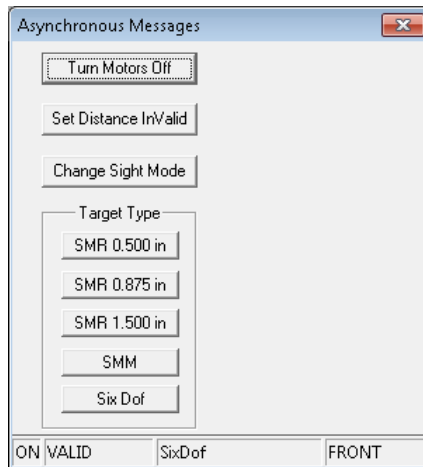
Modes: For the Asynchronous messages to fire it does not matter in which mode the Tracker commands are executed. All that matters is execution of Tracker command. For now the commands executed for this are in Listeners.cpp file.

For Asynchronous messages to work two things must be done.

1. Asynchronous messages must be started.
 2. Status listener and Change listener must be added.
- When the dialog pops up it can be seen that there are four buttons. The first button is to turn Motors ON or OFF, Set Distance Valid or Invalid, Change Sight mode, and buttons to change target types.



- If connected to sixdof capable tracker, the dialog popped up contains the ability to set the sixdof target as below.



- When any of the first two buttons are selected, the Status listener fires and any of the last two buttons are selected then Change listener fires.
- Initially when the dialog pops up, the status about the motors, distance, sight mode and target type is updated on the status bar and correspondingly the text on the buttons.
- For example: If the motors are turned on when the dialog pops up then the status on the status bar would say Motors ON and the text on the button would say “Turn Motors off” since motors are turned on. This would be the case with the Set Distance Valid/Invalid button.
- It can be noticed that whenever there is a change in any of the above mentioned commands the status on the status bar is updated whereas the text on the buttons is updated only for motors and distance.
- Once the dialog pops up with all the status and text on the buttons updated press Turn the Motors ON/OFF button and see that the status getting updated about the motors.
- The Set Distance Valid/Invalid button can be used to set the distance valid and invalid. To set the distance valid the Tracker is commanded to Home in front sight to the TMR location expecting the target to be there. If the target is not there then the status of the distance would be invalid else valid. To set the distance invalid, the Tracker is pointed to a location with no target.
- If the motors are turned OFF and if the change sight mode or Set distance valid/Invalid button is selected it might throw a motor state exception.
- To get a better view of how all these things are working run the demo.

Tracker Commands used:

- Start Asynchronous Messages
- Get Current Motor State
- Change Motor State
- Target Location Valid
- Home
- Target Type
- Change Target Type
- Stop Asynchronous Messages

Functions called in Demo App:

- GetAsyncCommandStatus () to get the status of the motors, sight, and distance.
- OnGetTargetAsyncStatus () is to get the status of the Target
- ExecuteAsyncCommand () executes the commands of motors, sight and distance.
- ChangeTarget () changes the target type.

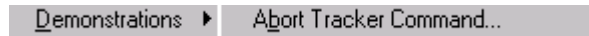
Event: Status Event, Change Event

Also Refer to Status Messages, Change Messages

Comments: If asynchronous messages are started they have to be stopped also using the Tracker command stopAsync().

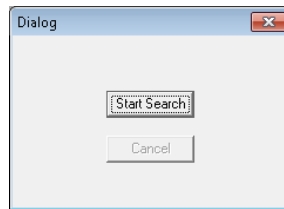
9.5 Abort Tracker Command

Menu Option: Demonstrations => Abort Tracker Command...



Modes: This works in any mode.

- Initially when the dialog pops up the Start Search button is enabled and Cancel button is disabled.



- Once the Search is started then the Start Search command is disabled and Cancel button is enabled.
- While the Search is in progress, if the Cancel button is selected, then the search is aborted and the driver throws an aborted exception. Whenever a Tracker command is successfully aborted it throws an aborted exception.

Tracker Commands used:

- Search
- Abort

Functions called in Demo App:

- StartSearch() is used in blocking mode
- AbortCommand () is used in blocking mode
- In listeners mode a callback function SEARCHDlgProc is implemented just to provide an example of how a callback function can be used with listeners.

Event:

- Command Complete Event

Comments: To provide ample time to cancel the search, the beam is pointed to a place where there is no target using a move command. This is just done as to provide an example for search.

9.6 Handling Exceptions

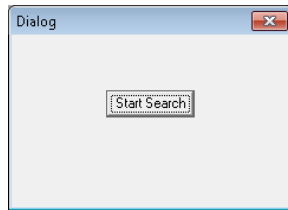
Menu Option: Demonstrations => Handle Exceptions...



Modes: This works in any mode.

In the demo application

- Initially when the dialog pops up the Start Search button is enabled.
- Once the Search is started then the Start Search command is disabled.
- Search command throws an exception after completing search that it cannot find the target. How the exception is caught can be seen both in blocking mode and listener mode.



Tracker Commands used:

- Search

Functions called in Demo App:

- StartSearch() is used in blocking mode.
- In listeners mode a callback function SEARCHDlgProc is implemented just to provide an example of how a callback function can be used with listeners.

Event:

- Command Complete Event

Comments:

- The beam is pointed to a place where there is no target using a move command.
- This is just done as to provide an example for search to throw an exception when it cannot find the target.
- Actually, Exceptions are used all over the place in demo app. But still, an example is provided forcing it to generate an exception so that it would be easier to see how driver generates an exception.

9.7 Run Applications

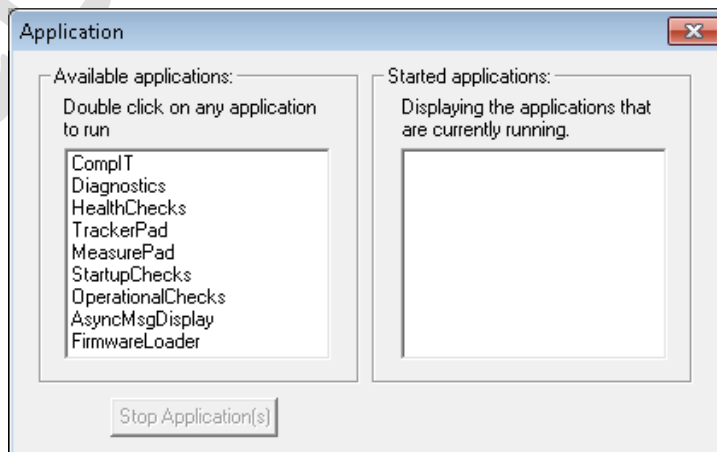
Menu Option: Demonstrations => Run Applications...



Modes: This works in any mode.

In the demo application

- When the dialog pops up there are two list boxes. One which displays the available applications and the other which displays the started applications.



- By the time the dialog opens up the available application list box is filled with all available applications for the Tracker connected. Double click on any of the application in the list box and it is started.
- When the application is started then the started application list box is filled with the started application(s). To fill the started application list box application listener is being used.
- When ever an application is started or stopped application listener fires. The event of the listener returns the name of the application and whether it is started or stopped. Depending upon that, application name is added or removed from the started application list box.
- Also, one more thing to notice is that whenever applications like Health Checks or Operational checks etc. i.e. the applications that keep the Tracker busy are started then the available application list box is disabled. This is being done as the Tracker is busy and if any other application is started it would throw Interface busy exception.
- Also, there is a button called “Stop Application(s).” Whenever this button is selected all the applications that are running are stopped. However when the Tracker is busy this button is disabled so that Tracker does not throw a busy exception if any application is stopped.

Tracker Commands used:

- Available Applications
- Started Applications
- Start Application
- Stop Application

Functions called in Demo App:

- OnAvailableApplications()
- OnStopApplication()
- OnRunApplication() in all the three modes.
- AddListenersforApps() in Listeners mode
- RemoveListenersofApps() in Listeners mode

Event:

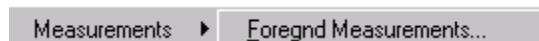
- Application Event
- Busy Event

Comments:

Busy Listener is used to enable and disable the available application list box and the stop application(s) button.

9.8 Foreground Measurements

Menu Option: Measurements => Foreground Measurements...

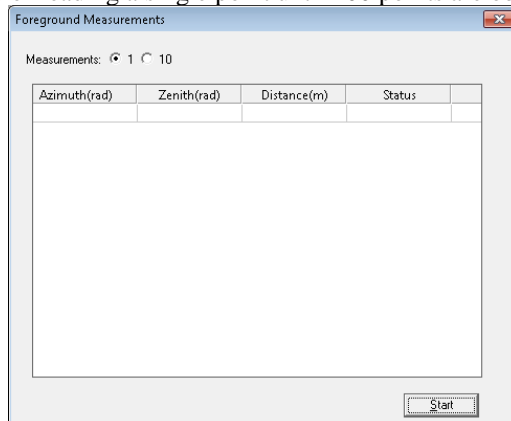


Modes: This works in any mode.

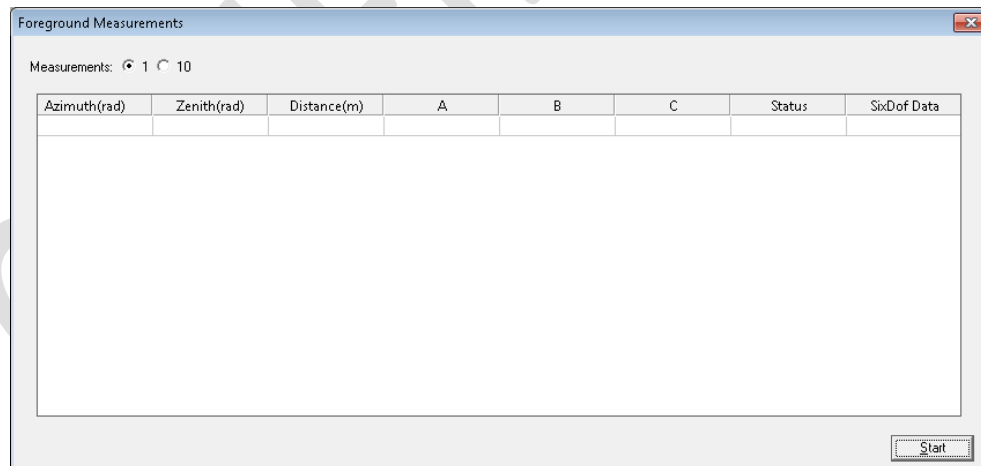
In the demo application

- This demonstration shows how a single point can be read and how an array of data can be read.
- On this dialog there are two options which are single measurement and 10, which is how to get an array of point data instead of single data. Setting the measure event rate of the Tracker can do this. The primary purpose of providing this option is to provide an example of how to read a single measure point data and an array of data. To read a single point it would be enough to set the measure event rate to 1 and to read one array of data at a time, set measure event to 10 to read 10 observations. It just depends on how many multiples of array's data are to be read. Like for example: To read 100 points

then the points can be read in multiples of array of 10, ten times or in multiples of array of 20 for 5 times instead of reading a single point until 100 points are obtained.



- To retrieve an array of data TrkDrvObjectArray should be used as demonstrated in Blocking.cpp, Listeners.cpp and WorkerThread.cpp files wherever readmeasurepoint data is done.
- Whenever data is retrieved using TrkDrvObjectArray data should be stored in a buffer locally since once the object array is deleted all the data is also deleted.
- After selecting the option of single or 10, select Start button. On clicking this button it will start taking the measurement(s), reading the data and stopping the measurement(s).
- The list control on the dialog updates the data obtained. There is a column of Status, which displays the status of the data i.e. whether the data obtained is accurate, inaccurate, or in error along with updating columns of Azimuth, Zenith and Distance. Time and statistics can also be obtained.
- If tracker is sixdof capable then Rotation A, Rotation B, Rotation C and SixDof Data column are also updated. If the tracker is locked on a sixdof probe, the sixdof data column would display yes else no. To get valid sixdof measurements, sixdof probe must be connected to the tracker and a probe must be selected. Probe can be selected using probe management dialog in Tracker pad. Tracker Pad can be launched via Demonstrations => Run Applications.



- Before starting the measurements a configuration must also be specified to tell the tracker how measurements must be taken. This configuration includes the number of samples for each observation and the type of triggers.
- How configurations are specified and for the different types of triggers refer to the code in the demo app and also to the documentation.
- For both single point and array of point measurements, it shows how a measurement is

started, how data is read, how the measurement is stopped, and how azimuth, zenith, distance and status information is obtained. Please refer to the code in the demo applications.

- How time and statistics can be obtained and for more details refer to the documentation

Tracker Commands used:

- Measurement Event Rate
- Start
- Read Data
- Stop
- Measurement Point Data
- Measurement Configuration

Functions called in Demo App:

- OnForegndMeasurements()
- OnBackgndMeasurements

Event:

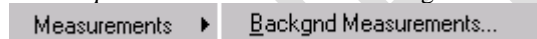
- Measure Data Event
- Busy Event

Comments:

- In the demo app, in blocking mode, the data is retrieved and stored in a list.
- In listeners mode the data is posted and in the worker thread mode also the data is posted.
- After the data is retrieved into a list or posted then the azimuth, zenith, distance and status is obtained. From this data if time and statistics are required they also can be obtained. Refer to the code in the demo app.

9.9 Background Measurements

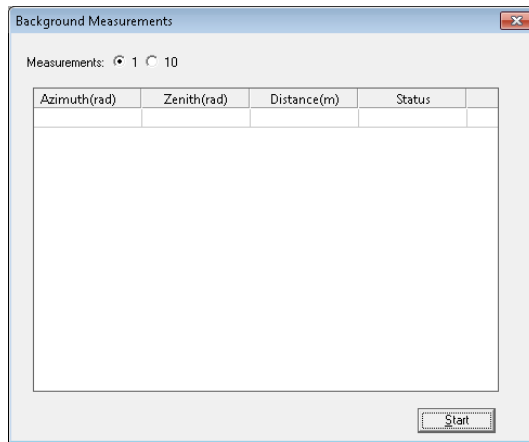
Menu Option: Measurements => Background Measurements...



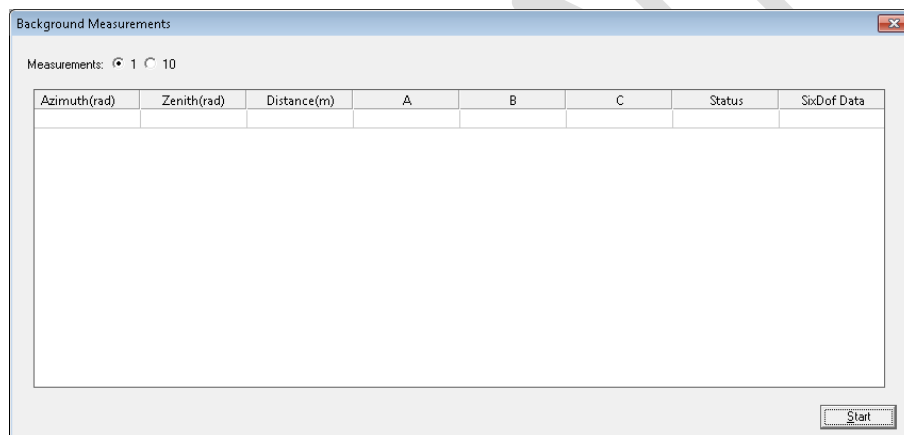
Modes: This works in any mode.

In the demo application

- Everything works similarly to foreground measurements except that it is background measurements that are started, stopped and data read. On this dialog also single and 10(array of data) options are provided.



- If connected to sixdof capable tracker then Rotation A, Rotation B, Rotation C and SixDof Data column are shown. If the tracker is locked on a sixdof probe, the sixdof data column would display yes else no. To get valid sixdof measurements, sixdof probe must be connected to the tracker and a probe must be selected. Probe can be selected using probe management dialog in Tracker pad. . Tracker Pad can be launched via Demonstrations => Run Applications.



Tracker Commands used:

- Background Measurement Event Rate
- Start
- Read Data
- Stop
- Measurement Point Data
- Measurement Configuration

Functions called in Demo App:

- OnForegndMeasurements()
- OnBackgndMeasurements
- OnStartBackgndMeasurements()
- OnStopBackgndMeasurements ()

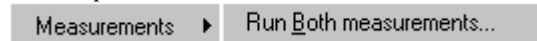
Event:

- Background Measure Data Event
- Busy Event

Comments: Similar to foreground measurements.

9.10 Run Both Measurements

Menu Option: Measurements => Run both Measurements...



Modes: This works in any mode.

- This demonstration is to show how both Foreground and Background measurements can be taken simultaneously.
- Everything works similarly to foreground and background measurements. Here also the measurement can be a single point or array of points of data. However, in this demo the measurements run until the button stop is clicked after the measurements are started.
- A very important thing is Background measurements cannot be started or stopped while the Tracker is busy executing any Tracker command or if the foreground measurements are running whereas Foreground measurements can be started with background measurements running.

In the demo application

- On the dialog initially both Start buttons of Foreground and Background are enabled whereas Stop button of both the measurements are disabled.
- If Foreground measurements are started before background measurements are started then the Start button of Backgnd is disabled and the Stop button of Foregnd is enabled whereas Stop button of Backgnd remains disabled.
- Once Foregnd measurements are stopped then the Start button of Backgnd is enabled.
- If the Backgnd measurements are started when the Tracker is not busy and Foregnd measurements are not running then start button of Foregnd is enabled and Foregnd measurements can be started.
- If the Foregnd measurements are started while the Backgnd measurements are running then the stop button of Backgnd is disabled since Backgnd measurements cannot be started or stopped with tracker busy. Stop button of Backgnd is enabled once Foregnd measurements are stopped.
- All this enabling and disabling of buttons is achieved using busy listener. Event of busy listener returns whenever the Tracker command, Foregnd measurements, Backgnd measurements are busy and not busy.
- Busy listener can play a very important rule in updating the UI whenever tracker is busy. Status and Change listener can play a very important role whenever status of the Tracker has to be updated etc.
- If connected to six dof capable tracker then azimuth, zenith, distance, rotation A, rotation B, rotation C, status and whether the data is sixdof or not is displayed. If the tracker is locked on a sixdof probe, the sixdof data column would display yes else no. To get valid sixdof measurements, sixdof probe must be connected to the tracker and a probe must be selected. Probe can be selected using probe management dialog in Tracker pad. . Tracker Pad can be launched via Demonstrations => Run Applications.

- Background Measurement Event Rate
- Foreground Measurement
- Background Measurement
- Measurement Point Data
- Measurement Configuration

Functions called in Demo App:

- OnStartBackgndMeasurements()
- OnStopBackgndMeasurements()

Event:

- Measure Data Event
- Background Measure Data Event
- Busy Event

Comments: Make sure Background measurements are started before Foreground stopped after Foregnd measurements are stopped.

9.11 Listeners

A special note about the listeners.

- How the listeners can be implemented can be seen in the demo app in Listeners.h and Listeners.cpp file. The listeners that are implemented in the Demo app are Command complete Listener, Measure Data Listener, BkgndMeasureData Listener, Status Listener, Change Listener, Application Listener and Busy Listener.
- Busy Listener is used to update the UI of most of the dialogs so that the user does not execute any Tracker command while the Tracker is busy. If a Tracker command is executed while the Tracker is already busy then it would throw Interface busy exception.
- Also, Blocking can be used with Listeners. This is shown as an example in case of background measurements. To stop the Backgnd measurements, blocking is used whereas to stop Foregnd measurements commandcomplete listener is used. This is demonstrated in Listeners.cpp file in method OnStopBackgndMeasurements(). Also, blocking is used in case of stopping the applications. This also can be seen in Listeners.cpp file in method OnStopApplications().
- Change Listener, Status listener and Busy Listener are used irrespective of any mode. These listeners are added after connecting to the Tracker and removed while disconnecting from the Tracker irrespective of any mode.
- Application listener is also used irrespective of any mode to fill the started application list box. The listener is added and removed when the async messages are started and stopped.
- For more information about Listeners please refer to Events.

9.12 Disconnect

- Click on the menu option Disconnect and this should disconnect from the tracker.
- Before disconnecting from the tracker all the listeners must be removed, asynchronous messages have to be stopped if they are started.
- After disconnecting from the tracker blocking should be set to false.
- The tracker command is disconnect. Refer to Connect and Disconnect and the method in demo app is OnDisconnect();

9.13 TrackerDriverDemo_VCSharp

- To launch the demo application, double click the executable in release directory. To run the application follow the same directions as for the C++ application.