Leica Metrology Foundation
User Programming Guide
Tracker SDK

July 19, 2023
Version 1.10.0

Contents

Chapter 1

Introduction

Dear developer,

The following two parts of this manual will introduce you into the Leica Metrology Foundation (LMF) for Leica Absolute Trackers. Currently ATS600, AT960, AT930, AT500, AT401, AT402 and AT403 models are supported.

Part One will give you a general overview over the design principles of LMF. Beginning with commonly understood object orientated design principles which build up the tracker object model, over basic data types like value objects, collections, over error handling down to threading aspects and finally, supported programming languages.

If you are comfortable with these concepts then you can directly go to the second part of this manual.

Part Two will go into the tracker specific objects of the LMF SDK. You will learn the concepts behind each tracker object model, from the top level object *Tracker* over *Measurements* to *Compensations*. Each chapter comes with short code samples which visualize the concepts and how they are used in code directly.

For a complete description of each property, command and event, please look into the **LMF SDK API Reference** manual.

Part I

Getting Started

Chapter 2

Installing the SDK

In order to use the LMF SDK you have to run the Setup.exe which will install all the tools you need to start.

2.1 System Requirements

In order to use the LMF SDK you need to have the following software installed:

- Windows 10 or higher
- .Net Framework 4.8 including Common Language Runtime (CLR) 4
- Visual C++ Redistributable for VS 2019

2.2 SDK Contents

The Leica Metrology Foundation SDK is composed out of the following parts:

2.2.1 LMF.Tracker.Connection.dll

This is the main part of the SDK. The LMF.Tracker.Connection.dll contains the whole LMF interface. The LMF Interface uses .NET 4.8 and can be directly referenced in a C#/.NET or Visual C++ project. In addition it is also COM-compatible allowing the use of LMF from VBA (For example Excel or PowerPoint).

2.2.2 LMFDocumentation.chm

The LMFDocumentation.chm is a compiled HTML help file containing the XML API documentation of the LMF interface.

2.2.3 Logalyzer

The Logalyzer is a tool to open and analyze the logfiles generated by LMF. It can also be used to connect to a process which is using LMF and display a live view of all log entries, which can be used for debugging purposes.

2.2.4 TrackerScope

The TrackerScope tool is a tool used to visualize the LMF programming interface in a property-grid manner. It allows you to connect to a real tracker or alternatively to a simulator. As soon as you are connected it displays all properties, methods and events of the LMF interface, allowing the user to call a method and see which properties change or which events get raised.

Chapter 3

Setting Up Your IDE

We provide multiple sample project to demonstrates common tasks in combination with MFC and Visual Studio. You can find this example project in the installation directory of the Leica Metrology Interface SDK.

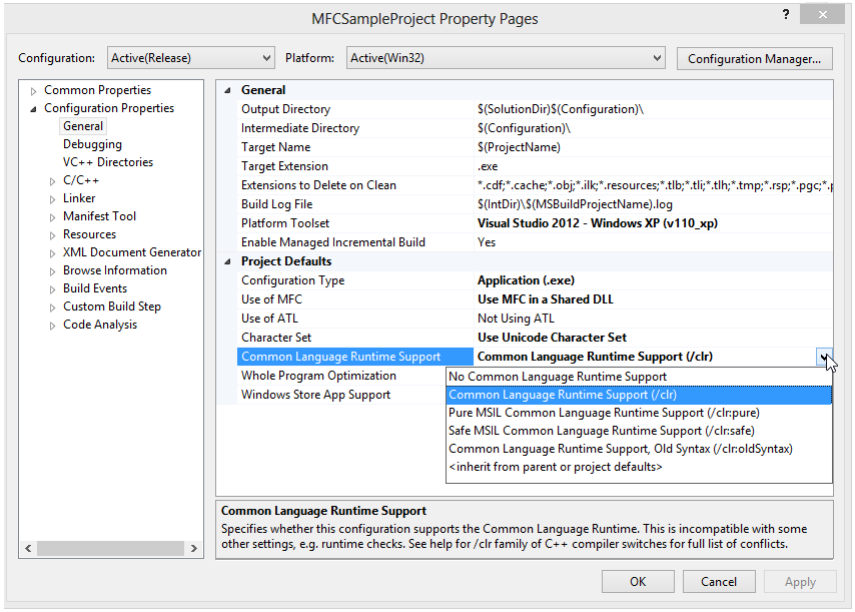
In case you are using the precompiled .exe files from the sample projects and do not have Visual Studio installed, please install the Visual Studio Redistributables which can be found under **%ProgramFiles(x86)%/Leica Metrology Foundation - Tracker SDK/VC++ Redistributables** and choose the one which fits to your processor architecture.

3.1 Visual C++ with CLR Support

In order to use LMF from a Visual C++ project with CLR support you need to do the following steps:

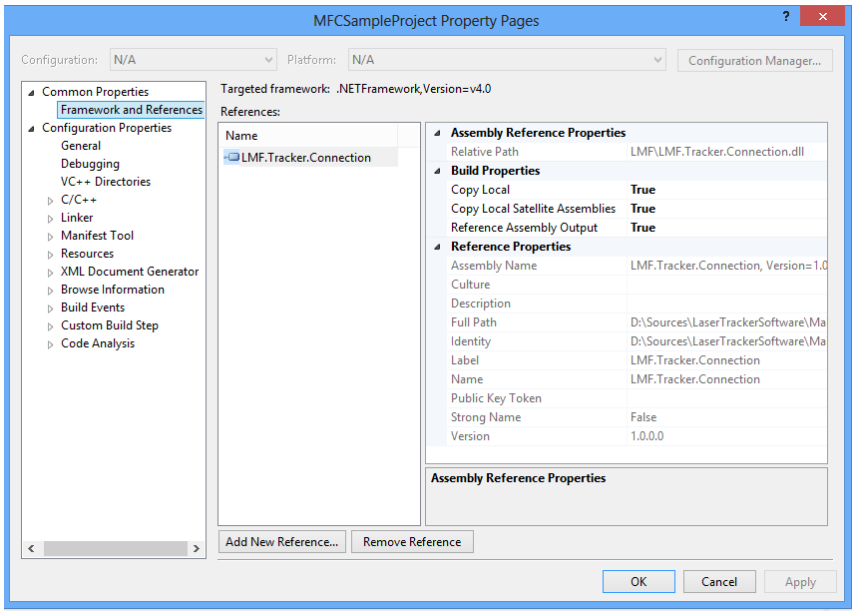
3.1.1 Activate Common Language Runtime Support

Any project that references LMF needs Common Language Runtime Support (/clr). You can activate CLR in your project properties inside Visual Studio or use the /clr compiler option.



3.1.2 Add a Reference to the LMF.Tracker.Connection.dll

As soon as you've activated Common Langugage Runtime Support you can simply add a new reference to the LMF.Tracker.Connection.dll.



3.2 Unmanaged C++ using the COM interface

In order to use LMF from a Visual C++ project you need to do the following steps:

3.2.1 Import the LMF.Tracker.Connection.tlb file

To use the LMF COM interface you have to import the LMF.Tracker.Connection.tlb and the mscorlib.tlb file as seen in the following snippet:

Listing 3.1: Import LMF COM Interface

```
#import <mscorlib.tlb>
#import "LMF.Tracker.Connection.tlb" no_namespace named_guids
```

To enable COM you have to call CoInitialize(NULL) at the start of the application and CoUninitialize() when closing the application.

3.2.2 Create a connection to a tracker

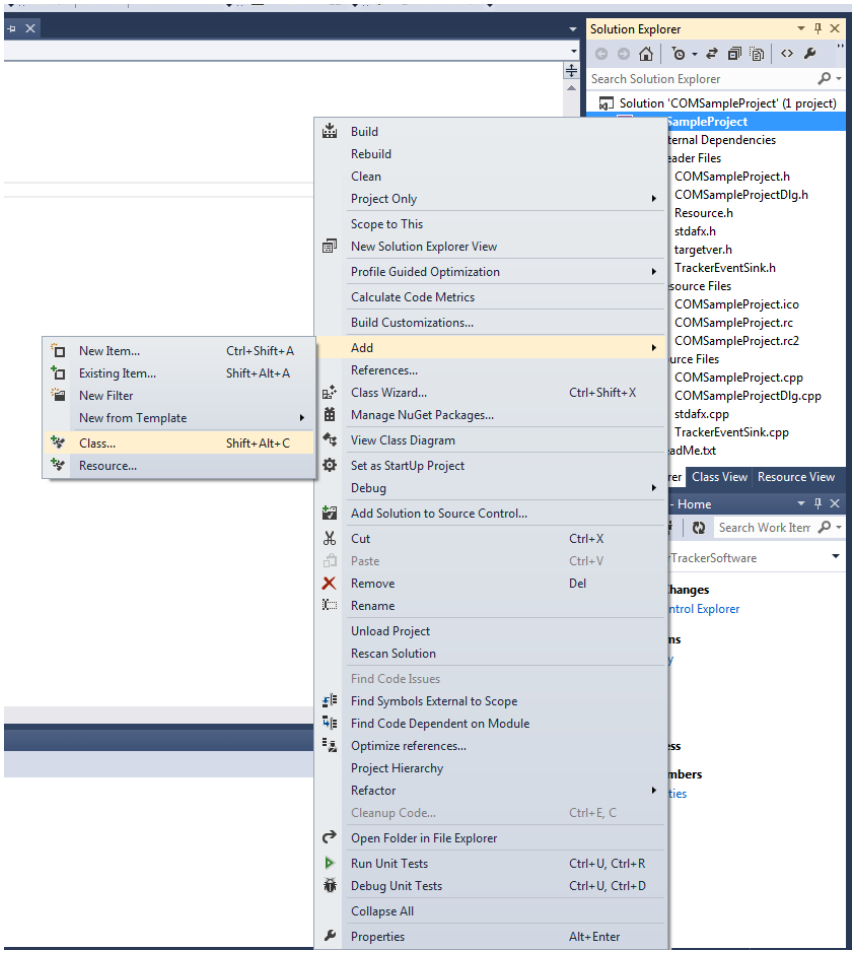
To create a connection to a tracker you need to create an instance of the Connection Class and call the Connect method with the appropriate ip address.

Listing 3.2: COM Connection

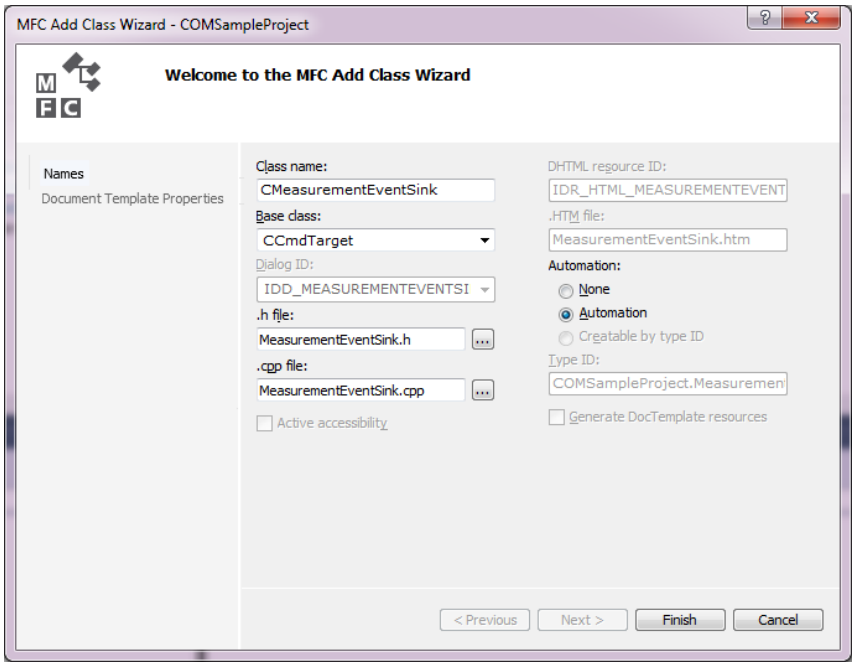
```
IConnectionCOMPtr connection;
connection.CreateInstance(CLSID_Connection);
tracker = connection->Connect(_T("AT960Simulator"));
```

3.2.3 Creating an EventSink

If you want to handle LMF Events like MeasurementArrived you need to implement a custom EventSink. This can be done using the MFC Class wizard inside Visual Studio as seen in the following image.



Name your EventSink class and choose CCmdTarget as the base class. Additionally you need to enable automation for the events to work.



You can now use the generated class to implement custom event handlers for all events of a specific event interface. You will need to map the interface with the INTERFACE_PART macro using the GUID of the event-class you want to handle, as seen in the code snippet below.

Listing 3.3: Interface Mapping

```
BEGIN_INTERFACE_MAP(CMeasurementEventSink, CCmdTarget)
  INTERFACE_PART(CMeasurementEventSink, DIID_IMeasurementSettingsCOMEvents, Dispatch)
END_INTERFACE_MAP()
```

For each event you want to handle you will need to call the DISP_FUNCTION macro using the name of the event, your handler method, the return value and all event parameters. VTS_DISPATCH is used for reference parameters like IMeasurementCollectionCOMPtr. For value parameters like double or bool you need to choose the corresponding VTS macro.

Listing 3.4: Dispatch Mapping

```
BEGIN_DISPATCH_MAP(CMeasurementEventSink, CCmdTarget)
  DISP_FUNCTION(CMeasurementEventSink,"MeasurementArrived",OnMeasurementArrived,VT_EMPTY,↩
      VTS_DISPATCH VTS_DISPATCH VTS_DISPATCH)
END_DISPATCH_MAP()
```

3.2.4 Connect the tracker object with the event sink

To finally connect an instance of the tracker object with the new EventSink you can do this the following way:

Listing 3.5: Connect EventSink

```
CMeasurementEventSink* measurementEventSink = new CMeasurementEventSink(this);
LPUNKNOWN pUnkMeasSink = measurementEventSink->GetIDispatch(FALSE);
AfxConnectionAdvise(tracker->Measurement, DIID_IMeasurementSettingsCOMEvents, pUnk-MeasSink, ↩
    FALSE, &m_dwCookie);
```

3.2.5 Disconnect

In order to disconnect from LMF you need to call Disconnect() on the tracker object and afterwards unadvise and release the used eventsinks.

Listing 3.6: Disconnect EventSink

```
   Tracker->Disconnect();

   AfxConnectionUnadvise(Tracker, DIID_ITrackerCOMEvents, pUnkSink, FALSE, m_dwCookie);
```

```
    pUnkSink->Release();
```

3.3 Unmanaged C++ using a wrapper project

In order to use LMF from a Visual C++ project you need to do the following steps:

If you want to handle LMF Events like MeasurementArrived you need to implement a custom EventSink. This can be done using the MFC Class wizard inside Visual Studio.

3.4 VBA/COM

To develop against the LMF Interface with VBA you first need to register the COM DLL. For this purpose you can find a deploy32Bit.bat and a deploy64.bat script at the install location of the Leica Metrology Foundation SDK. Before you run the batch file, please make sure you the logged in user has administrator rights. After you successfully ran the script, the LMF.Tracker.Connection can be accessed from VBA.

3.4.1 Microsoft Excel

As an example we will demonstrate how to set up a simple VBA project with LMF in Microsoft Excel 2010. For this reason open up **Excel 2010** and navigate to **Developer -> Visual Basic** to open up the Visual Basic IDE.

In order to add a Reference to the Leica Metrology Foundation, click on **Tools -> References..** to open the References window. Here you will see a list of all available Libraries. Search this list for the entry **LMF_Tracker_Connection** and check the box left to it. Now you are ready to access the LMF Library from within Excel. Please have a look at our provided Excel sample project if you are interested.

Correct Handling of References to LMF Objects

The memory handling is different in the LMF domain and in the domain of the VBA script in Excel:

- LMF is written in .NET and uses Garbage Collection
- The VBA script internally uses Reference Counting

These memory handling strategies are different and do not cooperate very well. Microsoft describes that this problem cannot be solved by the environment. So the user is responsible to handle the memory the correct way so that both memory systems may correctly cooperate. Otherwise Excel crashes when the whole application is closed. The reason is that the memory cannot be cleaned up correctly due to the different working principles of the two mentioned strategies.
Please follow the guidelines described here to avoid the crashes in Excel. Following these guidelines means that the script correctly releases all used memory at the end and lets the two memory handling strategies properly clean up when Excel is closed.

Guidelines
The following memory handling guidelines in the script avoid the Excel crashes when closing the application:

- All objects that are received from LMF into a VBA function are delivered according to the rules of .NET: Most of them (except the base types) are provided using call by reference (This even holds for strings). So if one of these objects is assigned to a script global reference this reference has to be set to Nothing before the application is closed.

- In case an object received by LMF is assigned to a GUI element (label text, caption etc.) the reference in the GUI has to be set to Nothing before the application is closed. This even holds for a name of an object that is assigned to a label as it also holds for strings.

- VBA functions allow to receive function parameters using call by value (keyword ByVal). This only copies the object itself. In case this object contains any references to other objects they are still provided as references. So the keyword ByVal does not do a deep copy of all references but only a copy of the topmost level. As every object received from LMF contains some internal references to other objects (except the base types) even the objects received using call by value have to be handled according to the rules mentioned before.

- In case an LMF object is only assigned to a function internal variable no special care has to be taken as the references of a function are cleaned up when the function is left.

- Sometimes it is useful to assign a property of an LMF object to a script global variable or to a GUI element (e.g. list of available targets showing their names). These references to subobjects also have to be set to Nothing before the application is closed. It is not sufficient to clean up the topmost reference to the LMF object (e.g. target list) as every reference pointing to any LMF object prevents the memory system from correctly cleaning up if it is not set to Nothing (except the base types).

- In case a string is directly received from LMF into a VBA function using the keyword ByVal the system correctly creates a copy of it. And the string itself has no references to any further LMF objects. So this string does not need to be cleared.

In the sample the references are cleaned up properly according to the guidelines at the following two places:

- In the Disconnect function all references to LMF objects related to the previously connected tracker are cleaned up.

- In the UserForm_Terminate function all other references to LMF objects are cleaned up.

3.5 LabView

LabView supports .Net assemblies, therefore you can simply use the LMF.Tracker.Connection.dll and start working. However due to a limitation, LabView supports only the events that have the standard .Net event signature with the sender and the derived class of EventArgs. Therefore some LMF events, like for example the MeasurementArrived event are not working out of the box.

In order to support all events you need to implement a wrapper dll that maps the events to provide a LabView compatible signature. You can find an example for this in the provided LabView sample.

3.6 C#/.NET

LMF uses the .NET Framework version described in 2.1 , therefore its compatible with any .NET Project of that version or higher. Please have a look at our provided C# Sample Project.

3.7 Python

LMF may be included into a Python (standard CPython) environment using the Pythonnet package (Python for .NET) that is used to integrate a .NET component into the Python application. There are several available Python environments that are based on different versions of Python. To be able to integrate LMF into a specific Python environment a Pythonnet package matching the given Python version is required.
The Pythonnet packages can be downloaded from:
https://pypi.org/project/pythonnet/
A guide for the integration is available in:
https://pythonnet.github.io/

Required Components and their Installation

The sample is based and tested using the following software:

- Python 3.9 64 bit or Python 3.10 64 bit on Windows 10
- The package pythonnet 3.0.0.post1
- Visual Studio 2022 to create and debug the sample (not the only way to start the sample (see below), just used for developing / debugging)
- Latest release of LMF.Tracker.Connection.dll (included in the sample)
- Package tkinter included in the Python environment of Visual Studio (no explicit installation is required)

- Previously the sample was tested with Python 3.6.3, pythonnet 2.3.0 and Visual Studio 2019.

The system needs the following preparation steps:

- Install the following packages using the Visual Studio Installer (or use installed Python interpreter):
  - Python development
  - Python 3 64-bit
- Install pythonnet package into the Python 3.9 or Python 3.10 environment as "pip install pythonnet" (e.g. within "Manage Python Packages" in Visual Studio)

Starting the Python Sample

There are two ways to start this specific sample:

- Using Visual Studio 2022 the provided LMF.Tracker.SDK.PythonSampleProject.sln can be directly opened (required Python installation would be triggered). The contained sample may be started directly in case all required software packages are installed (see above).
  Using this solution the LMF.Tracker.Connection.dll may be moved to a different place than the default one by inserting a matching search path into the project. Using this search path the dll is found by the Python environment.
- Starting the sample using the Python interpreter only requires to have the Python file and the LMF dll in the same folder to be started (another solution with Python paths is possible).
  Installing the Python environment (e.g. Python 3.9) through the Visual Studio Installer puts it to: C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python39_64.
  The following command is used to start the sample in case all required software is installed: C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python39_64>python.exe C:\Program Files (x86)\Leica Metrology Foundation - Tracker SDK\PythonSampleProject\PythonSample.pyc
  In case Python (e.g. Python 3.10) was installed directly the following command can be used: py C:\Program Files (x86)\Leica Metrology Foundation - Tracker SDK\PythonSampleProject\PythonSample.pyc

Technical Details for Developers

General remarks for mixing Python and C#:

- Python is not restricted to registering handlers for standard .NET events using sender and EventArgs: Events with less / more and different event parameters may also be received. So LMF may be used in Python directly without any manually written event wrapper in between.

Pitfalls in the mixed environment:

- The Visual Studio debugger breaks on a break point that is hit by the main thread. Receiving the .NET events is done on a worker thread in Python. In that case the debugger does not halt.
- The convenient implicit type conversions that are given in C# do not work in Python: So if an event handler runs into a type mismatch error in Python it does not continue its work, but the program itself continues. In case the same event arrives again the event handler is called again and "returns" at the point the exception occurs. The exception is logged in the console window.
- Type mismatch exceptions in the Python event handlers that handle an event fired by the .NET assembly are not caught by the Visual Studio debugger debugging the Python project. But attaching the LMF debugger (C# environment) shows the exceptions in the event handlers as type mismatch exceptions. So the Python handler is somehow debugged in the C# context.
- Importing of the C# components in Python is based on the namespaces of the C# / .NET packages. So not all types residing in the LMF.Tracker.Connection.dll may be imported with the same statement.
- There have been some changes in pythonnet 2.x to pythonnet 3.x regarding enum objects, requiring slight code changes (documented in the sample).

Chapter 4

Deployment

4.1 Common Requirements

LMF is a .Net Library based on the .Net Framework 4.8 and the Common Language Runtime (CLR) 4. This results into the following prerequisites when deploying an application that includes LMF.

- .Net Framework 4.8 or higher.
- Visual C++ Redistributable for Visual Studio 2019 (only for the provided sample projects which are Visual Studio 2019 solutions)

4.2 Using LMF as .Net Assembly

When using LMF in a .Net project, you simply need to distribute the LMF.Tracker.Connection.dll together with your application. Make sure the dll is placed alongside the executable (exe).

4.3 Using LMF as COM Component

When using LMF over COM in a unmanaged C++ project or with other COM Clients like VBA (Excel) you need to do the following steps when deploying your application.

1. Make sure the LMF.Tracker.Connection.dll is distributed alongside your executable.
2. Register the LMF.Tracker.Connection.dll on the client machine using RegAsm.exe (The LMF SDK contains a **deploy32Bit.bat** which can be used to register the LMF dll.)
3. If you are developing a 64Bit application, you will need to use the **deploy64Bit.bat** to register the LMF.Tracker.Connection.dll
4. To uninstall LMF you have to use the **undeploy32Bit.bat** or **undeploy64Bit.bat** respectively.

Part II

Leica Metrology Foundation - Basic Concepts

Chapter 5

Basic Concepts

5.1 Introduction

This part of the documentation will give you a general overview over some of the basic concepts of LMF. It will cover aspect which apply to a wide range of object in the SDK and will highlight common structures and behaviors throughout the SDK.

5.2 Tracker Object Tree

The Tracker LMF is organized as an object tree structure. That means that once you have a connection to a tracker you will get a living **Tracker** object which contains subobjects which contain all properties, events and methods ordered into a topological structure. From than on all changes from and to the system are handled over this object tree as long as the connection is alive.

5.2.1 Tracker Object Tree and Event Handlers

All objects attached at the tracker tree send out an event each time they have been changed. An event handler consuming these events must never block but should return as fast as possible. Letting the thread sleep within the event handler is not sufficient. It really needs to return as fast as possible, because all events are delivered using the same thread. Otherwise the further events cannot be delivered to the application and the tracker tree is not updated anymore. The state of the tracker tree is preserved as far as possible until the event handler returns not to change the tree during the execution of the event handler. This provides a consistent tree during the whole execution time of the event handler but prevents from getting any updates during this time. So in case an application relies on an up-to-date tracker tree all the time it must never block any event handler.

Event Handling Example (Positioning Loop)

The following sample shows how to position the tracker to a given target (e.g. a reflector) and to wait until the tracker is ready to measure after positioning it to the target. The sample uses the asynchronous execution scheme where a lot of events are involved to notify the application that an action has been completed.
It is not required to wait until the tracker is ready to measure after positioning but the sample shows how to wait for two combined events correctly without blocking any thread during the execution.

Listing 5.1: PositioningLoop Sample

```
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using LMF.Tracker;
using LMF.Tracker.Enums;
using LMF.Tracker.ErrorHandling;
using LMF.Tracker.Targets;
using LMF.Units;
```

```
namespace PositioningLoopSample
{
    /// <summary>
    /// Sample code for positioning and waiting for the ready to measure state using the asynchronous execution scheme
    /// It is taken out of a WPF application but the synchronisation scheme holds for all types of applications.
    /// It shows how to let a positioning loop correctly wait for some events without blocking any thread.
    /// </summary>
    public partial class MainWindow : Window
    {
        private readonly AutoResetEvent _are = new AutoResetEvent(false); // Synchronisation point
        bool _positionToFinishedReceived; // Represents the state if the PositionToFinishedEvent arrived
        bool _statusReadyReceived; // Represents the state if the ready event received

        public Tracker Tracker { get; private set; }

        public MainWindow()
        {
            InitializeComponent();
        }

        private void connect_Click(object sender, RoutedEventArgs e)
        {
            Tracker = new Connection().Connect("10.62.34.175");
            Tracker.Settings.CoordinateType = ECoordinateType.Spherical;
            Tracker.Measurement.Status.Changed += Status_Changed;
            Tracker.PositionToFinished += TrackerOnPositionToFinished;
        }

        /// <summary>
        /// Event handler to receive the PositionToFinishedEvent and to update the internal state
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="foundTarget"></param>
        /// <param name="lmfException"></param>
        private void TrackerOnPositionToFinished(Tracker sender, Target foundTarget, LmfException lmfException)
        {
            // Here we must not wait for any other change in the tracker tree but return immediately so that
            // the thread LMF delivers all events on is ready to deliver the next event.
            _positionToFinishedReceived = true;
            CheckCondition();
        }

        /// <summary>
        /// Helper method to release the synchronisation point in case both conditions are fulfilled
        /// </summary>
        private void CheckCondition()
        {
            if (_positionToFinishedReceived && _statusReadyReceived)
            {
                _are.Set();
            }
        }

        /// <summary>
        /// Event handler to receive the measurement status and to update the internal state
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="newValue"></param>
        private void Status_Changed(LMF.Tracker.MeasurementStatus.MeasurementStatusValue sender, LMF.Tracker.Enums.EMeasurementStatus newValue)
        {
            if (newValue == EMeasurementStatus.ReadyToMeasure)
            {
```

```
                _statusReadyReceived = true;
                CheckCondition();
            }
        }

        /// <summary>
        /// Central positioning loop to execute several positionings without blocking any thread.
        /// The goal is to wait until the previous positioning command has been completed and the tracker
        /// is ready to measure before continuing with the subsequent positioning command.
        /// But no event handler must be blocked: They just have to set some flags to return immediately
        /// not to block the thread LMF delivers the events on.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void startPositioning_Click(object sender, RoutedEventArgs e)
        {
            int i = 0;

            while (true)
            {
                // It is important to start the positioning loop in a separate thread, because it waits until
                // the positioning command is completed and until the tracker is ready to measure in this ↩
                    sample.
                // All events are delivered on the same thread (the one that created the connection class to ↩
                    connect
                // to the tracker). So the loop must not be executed on this thread. Otherwise LMF cannot ↩
                    deliver
                // its events anymore and the application runs into a deadlock.
                await Task.Run(() =>
                {
                    _positionToFinishedReceived = false;
                    _statusReadyReceived = false;
                    // Positions to point 0 or 1 depending on the value of i.
                    Tracker.PositionToAsync(true, false, i == 0 ? -127.0 : 123.0, i == 0 ? 94.0 : 92.0, 4549);
                    i++;
                    i = i % 2;
                    _are.WaitOne();
                    // After this synchronisation point the system is ready to measure or to do some other ↩
                        actions.
                });
            }
        }

        private void initialize_Click(object sender, RoutedEventArgs e)
        {
            Tracker.Initialize();
        }
    }
}
```

DRO and Measurement Queue

An application uses the DRO stream to show the approximate position of the currently active target (e.g. a reflector) in case it is locked on. And in case the application does some continuous measurements (continuous time measurement profile) the application receives a lot of measurements from the continuous measurement stream. Using the maximum speed LMF delivers 1 kHz measurements packed into 50 packets per second.
Receiving the DRO measurements and the continuous ones requires the application to consume the arrived measurements very fast not to block the thread LMF delivers all the events on. Otherwise LMF queues up the measurements internally.

In case more than 30 DRO measurements are not delivered LMF writes a warning into the LMF log file:
Client EventHandler too slow. DRO Queue at 31
For the continuous measurements the warning boundary of the queue is at 150 measurement collections. The warning message is the following:
Client EventHandler too slow, Cont. Measurement Queue at 151
In case several warning messages of either type can be found in the LMF log file the application does not handle all events fast enough.
Queuing up the measurements in LMF is harmless for a short time period. No measurement is lost. But the events informing about some changes in the LMF tracker tree are delayed during this period. This may reduce the responsiveness of the application. And in case the period is very long the memory gets filled up by the queued measurements.

5.3 Object Oriented Design

The LMF SDK follows an object oriented software design meaning that it groups real world scenarios into classes with properties, events and methods. To use the SDK in your preferred programming language you should be familiar with the following concepts of object oriented programming:

5.3.1 Inheritance

Since many objects in the SDK are grouped into collections of some kind these object derive from more abstract classes which encapsulate the more common properties of these objects, i.e. different reflector types all derive from the same Reflector class which itself derives from the more abstract class *Target*. LMF uses the abstract way of data to give access to the common functionality of all these different objects, i.e. to iterate through all targets the system knows off it provides a collection of **Target** objects which all have a **Name** and a **GUID**. To be able to come back from the abstract representation of an object to the underlying object again you will need the next two concepts of OOP.

5.3.2 Type Checking & Type Casting

If one has an abstract representation of an object but one want to use the derived representation of that object one would do the following. One would first **Type Check** the abstract representation of that object and then **Type Cast** this object into the derived representation so that it can be used in the code. The following list contains examples how this can be done for a few languages:

- C#

```
if(objTarget is Reflector)
{
  Reflector r = objTarget as Reflector;
  // Do something with 'r'
}
```

- C++

```
if (typeid(*objTarget) == typeid(Reflector)
{
  Reflector* r = dynamic_cast<Reflector*>(objTarget);
  // Do something with 'r'
}
```

- VBA

```
If TypeOf objTarget Is Reflector Then
   DIM r as Reflector
   SET r = objTarget
   ' Do something with 'r'
End If
```

Chapter 6

Data Types

All properties in LMF use different basic data types to provide additional functionality like unit conversion, change notifications or formatted strings. These are the three basic data types used in LMF:

- BoolValue
- IntValue
- DoubleValue
- EnumValue (for Example PowerSourceValue)

6.1 Value Property

The most important property of each data type is the value property. It contains the actual value converted into the currently active unit setting.

6.2 ValueInBaseUnits Property

Besides the Value property, numeric data types have a ValueInBaseUnits property which is always uses the Leica default units. This eliminates the need to check which units are set in the settings.

6.3 Change Notification

Each data type has its own **changed** notification. This event gets raised each time the data type gets updated. If you want to get informed about something like the MeasurementStatus you can use its change notification. Each change notification has information about the sender and the new value of the property.

Listing 6.1: React on a Change Notification

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.BasicTypes.BoolValue;

namespace SDK.Samples.CodeSamples
{
    class ChangeNotificationExample
    {
        public void Example1()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            // Attach to the Changed event which is offered by a sample of properties or classes
            tracker.Compensations.Changed += Compensations_Changed;
        }

        void Compensations_Changed(LMF.Tracker.Compensations.CompensationCollection sender)
        {
            //do something with the compensation collection
        }
    }
}
```

6.4 Unit Conversion

The value of each property gets automatically converted into the units you set on the settings object.

6.5 Additional Properties

There are different variations the basic data types which provide additional functionality.

- **ReadOnly**, These properties can only be set by LMF internally. They are used for informational properties like for example **ReadOnlyBoolValue IsSensorConnected** to inform the clients about the connection state of the sensor.
- **WithRange**, These properties have additionally a minimum and / or maximum property to specify the range of the value. An exception will be thrown if you try to set a value outside of these boundaries

Chapter 7

Command Pattern

7.1 Synchronuous/Asynchronous Commands

Time consuming commands like e.g. positioning or initializing offer besides the synchronous way also asynchronous alternatives.

7.2 CommandFinished Event

The functions of the synchronous interface do not return before the task is completed, while the asynchronous functions do so. In the asynchronous case, an event is sent upon completion of the command informing of a result or the reason failure.

Chapter 8

Collections

At the time when there are more than one item of an object of the same base class, the SDK provides a collection class which contains all these items. For example, the **TargetCollection** class contains all **Target** objects a tracker can understand. All the different collection classes provide more or less the same common interface to the user. This chapter will describe the most common members of the collection classes. Please note that not all classes necessarily have to offer all of the members described next.

8.1 Count Property

The **Count** property of a collection class simply gives back the number of items in the collection.

8.2 Selected Property

This property gives you the actually selected item of the collection. For example, if the tracker points onto a reflector the **Targets.Selected** property gives back the actually selected reflector. If nothing is selected this property is null.

8.3 Select Method

In the LMF the selection of items in a collection follows the rule that the **item itself** offers a **Select** method if it can be selected from the user. Not all items of a collection necessarily have to offer this method, i.e. only *Reflectors* can be selected but no *TProbe*. Typically these *unselectable* items are managed by LMF itself and are selected automatically when it is most appropriate.

8.4 Changed Event

If the collection changes somehow, i.e. add/removing items to/from the collection, then a **Changed** event will be fired.

8.5 SelectedChanged Event

On the other side, if the selected item changes, the **SelectedChanged** event will fire. This can be due to user interaction (selecting a new item) or due to automatic selection by the LMF framework.

Chapter 9

Threading

9.1 Multithreading

9.2 GUI Thread Synchronization in .Net

There are a lot of event handlers in the Tracker object tree to which you can register to be informed about various changes of underlying tracker data. By its intrinsic nature these events happen asynchronously and would normally get fired from a separate thread than the main thread of the running program. To make life easier in using these events to i.e. trigger an update on the GUI part of your application without the need to synchronize the event with your main thread (GUI controls don't like to be modified from other threads), LMF has a built in mechanism to always throw these events on the same thread in which your first LMF object has been instantiated. To be more specific: all events will be thrown on that thread on which the instance of type **Connection** or **TrackerFinder** has been instantiated.

Best practice in this case would be to create a new instance of i.e. the **Connection** class on your GUI startup sequence to let LMF know that all events should be thrown on that same thread afterwards. Establishing the connection itself (i.e. by **Connection.Connect()**) can be done in a separate thread if needed and will not interfere this mechanism. This mechanic only works when using LMF in a .Net Project.

9.3 Threads to Work on the Tracker Tree in .Net

The tracker tree is designed to fire an event every time a property is changed or an asynchronous command terminates (e.g. PositionToTarget): These events are delivered using the thread that instantiated the **Connection** class. So in case the same thread is used to execute a series of commands in LMF in a loop it might get stuck in a deadlock as most of the commands change the internal state of LMF and may trigger an event: Delivering all the events is done in a synchronous manner by LMF so that they do not overtake each other. And as long as an event handler is not returning no other event is fired and the tracker tree is unchanged as much as possible so that it is consistent during the execution of the event handler. So the command triggering an event is stalled until

the event is consumed. But the event cannot be consumed if the command is executed on the thread that is used to deliver the events as it is still executing the command. So the application runs into a deadlock. It is very important to use a different thread to execute the commands in LMF so that the events may be correctly delivered, therefore.

9.4 COM Threading Model

When using the LMF COM Interface, the default COM Threading model applies. In the provided sample project, we use a Single Threaded Apartment on the main thread to communicate with LMF. To receive the events on a different thread, you will need to call **CoInitialize/CoInitializeEx** on this thread instead of the main thread. More Information about this topic can be found in MSDN (https://msdn.microsoft.com/en-us/library/ms809971.aspx)

Chapter 10

Error Handling

10.1 Exceptions

Each property or method call in LMF does throw an exception if something went wrong. For example if you try to do a stationary measurement while the Tracker is not locked on a target, you will receive an exception instead of a measurement result. Unlike synchronous commands, asynchronous commands will return an exception in their finished-event.

10.1.1 LmfException

All exceptions that are thrown in LMF are of the type LmfException. An LmfException offers detailed information about the error in form of the following properties:

- Number
- Title
- Description
- Solution

10.1.2 Example

Every client using LMF needs to implement some degree of exceptionhandling. The following Example shows how to handle exceptions when doing synchronous and asynchronous measurements.

Listing 10.1: Exceptionhandling

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.ErrorHandling;
using LMF.Tracker.MeasurementResults;
```

```
namespace SDK.Samples.CodeSamples
{
    public class ExceptionHandling
    {
        private readonly Tracker _tracker;

        public ExceptionHandling()
        {
            _tracker = new Connection().Connect("192.168.1.1");
        }

        public void MeasureSynchronousExample()
        {
            try
            {
                Measurement measurement = _tracker.Measurement.MeasureStationary();
            }
            catch (LmfException exception)
            {
                Console.WriteLine(exception.Number);
                Console.WriteLine(exception.Title);
                Console.WriteLine(exception.Description);
                Console.WriteLine(exception.Solution);
            }
        }

        public void MeasureAsynchronousExample()
        {
            _tracker.Measurement.MeasurementArrived += Measurement_MeasurementArrived;
            _tracker.Measurement.StartMeasurement();
        }

        private void Measurement_MeasurementArrived(LMF.Tracker.Measurements.MeasurementSettings ↩
             sender, MeasurementCollection measurements, LmfException exception)
        {
            if (exception != null)
            {
                Console.WriteLine("An error happened during measurement");
                Console.WriteLine(exception.Number);
                Console.WriteLine(exception.Title);
                Console.WriteLine(exception.Description);
                Console.WriteLine(exception.Solution);
            }
        }
    }

}
```

10.1.3 Exceptions over COM

When using the LMF COM Interface you will receive Exceptions of the type *_com_error* instead of LmfExceptions. Each *_com_error* has a description which is a combination of the number, title and description of the original LMFException. To access these properties individually, you can use the Errornumber to query the LMF error database by calling *Tracker.GetErrorDescription(number)* or *TrackerErrors.GetErrorDescription(number)*.

The following Example shows how to handle thrown exceptions using LMF from an unmanaged c++ MFC client.

Listing 10.2: COM Exceptionhandling

```
try
{
  IMeasurementCOMPtr meas = Tracker->Measurement->MeasureStationary();
}
catch(_com_error &err)
{
  BSTR bstrDescription;
  bstrDescription = err.Description();
  AfxMessageBox(bstrDescription);
  return ;
}
```

10.2 Errors, Warnings, Information

In contrast to exceptions that can only occur as a direct answer to a property or method call, errors warnings and information can be received at any time. This is mostly used to send information about the current state of the tracker without relation to any command. In the following example you can see how to handle errors warnings and information events.

Listing 10.3: Error, Warning, Information Events

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;

namespace SDK.Samples.CodeSamples
{
    public class ErrorHandling
    {
        public ErrorHandling()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            //Register all Error events
            tracker.ErrorArrived += tracker_ErrorArrived;
            tracker.WarningArrived += tracker_WarningArrived;
            tracker.InformationArrived += tracker_InformationArrived;
        }

        private void tracker_InformationArrived(Tracker sender, LMF.Tracker.ErrorHandling.↩
            LmfInformation info)
        {
            //Print all the information to the Console
            Console.WriteLine(info.Number);
            Console.WriteLine(info.Title);
            Console.WriteLine(info.Description);
            Console.WriteLine(info.Solution);
        }

        private void tracker_WarningArrived(Tracker sender, LMF.Tracker.ErrorHandling.LmfWarning ↩
            warning)
        {
            //Print all the information to the Console
            Console.WriteLine(warning.Number);
            Console.WriteLine(warning.Title);
            Console.WriteLine(warning.Description);
```

```
            Console.WriteLine(warning.Solution);
        }

        private void tracker_ErrorArrived(Tracker sender, LMF.Tracker.ErrorHandling.LmfError error)
        {
            //Print all the information to the Console
            Console.WriteLine(error.Number);
            Console.WriteLine(error.Title);
            Console.WriteLine(error.Description);
            Console.WriteLine(error.Solution);
        }
    }
}
```

Chapter 11

Persistency

The LMF interface provides an easy and simple way to communicate with your tracker. To guarantee the same behavior of the tracker in each session (from connection with a tracker until its disconnection), it generally sets all values and settings which you can change on the LMF object tree back to default values. These default values can differ slightly from one tracker type or family to another but stay the same for each connection as long as you connect to the same tracker type or family.

This means you always find the same start situation after you have connected to a tracker and can now change the settings to your choices or leave it as it is. This simplifies the startup procedure in your application as you only have to setup the differences to the default values as your application demands.

Of course there is no rule without exceptions and so it is in LMF. The following table lists the exceptions in terms of which properties are **NOT** following this rule of the default values.

Property in object tree	Persistent after LMF reconnect	Persistent after controller reboot
Tracker.InclinationSensor.CurrentInclinationToGravity	yes	no
Tracker.InclinationSensor.InclinedToGravity	yes	no
Tracker.Targets.Tips[].Select() if the tip is a VirtualTip	yes	no
Tracker.MeteoStation.Source.Value	yes	no

Chapter 12

Simulator

If no hardware is available to test the software, it is possible to connect to different simulators. All simulators have a graphical interface to manipulate the representing hardware.

12.1 Connection

To connect to a simulator a specific string is needed instead of an ip address.
The string is constructed as followed:
<Tracker Type><Range>Simulator
For example:

- AT960LRSimulator - AT960 long range Tracker
- AT960MRSimulator - AT960 mid-range Tracker
- AT401Simulator - AT401 Tracker, they do not have different ranges

If the application can handle different Trackers at the same time, it is possible to test that too. The serial number of the simulator can be alternated to distinguish them. Else the error 10011 - Controller Already In Use - will be thrown if multiple connections, in the same application, are made to a simulator with the same type and serial number. To successfully connect to a different simulator of the same type, the connection string needs to be changed as followed:

<Tracker Type><Range>Simulator#<Serial Number>
For example:

- AT960LRSimulator#506432 - AT960 long range Tracker with serial 506432
- AT960LRSimulator#1 - AT960 long range Tracker with serial 1

12.2 Simulator Window

The next sections will explain specialties on how the hardware is simulated using the features of the simulator window.

12.2.1 DRO

For the **continuous distance measurement profile** the coordinates will be send specially if changed during measurement. Incoming measurements will be delivered on the vector from the start to the end coordinate with the correct defined distance range.

12.2.2 Power Source

If both sources are set to **None** the simulator will disconnect.

12.2.3 Targets

There are different targets which can be chosen to be looked at. These targets can have details like tip and mounts. If the selected target shows a "*" at the end of the name, this means it is not compensated and therefore cannot be used to measure.It will also be signalized with a red dot right next to the drop down. Tips can also show a "*" which again means it is not compensated. Additional there is an "!", this means the shank of the tip is not compensated.

12.2.4 Persistency

Every setting in the simulator window, which is written in blue or has a blue border, can be saved. It is saved by type and serial number. It can also be reset to the defaults.

Part III

Working with the tracker

Chapter 13

Configuring Network

The ethernet interface can be configured either by static ip or dynamically through DHCP. To connect the tracker to an pre-existing wired network the supplied straight RJ45 cable should be used. To connect directly to the tracker without a switch/hub you can use the supplied crossed-over RJ45 network cable.

Alternatively, a wireless connection over an external AP can be established. Please consult Tracker Pilot documentation for further information.

Chapter 14

Connecting to a Tracker

To connect to a tracker can be done in two different ways. Either by connecting to an previously known IP address directly or by letting LMF search for available trackers on your network and selecting one of them to connect afterwards.

14.1 Connecting via 'Connection' class

The following code example shows how to directly connect to a tracker from which the IP address is known. First instantiate an object of type **Connection** and use its only method **Connect** with the IP address of the tracker. If no exception gets thrown (i.e. Tracker not connected, etc.) the resulting object is the tracker object and ready to work with.

Listing 14.1: Connecting via 'Connection' class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.ErrorHandling;

namespace SDK.Samples.CodeSamples
{
    class ConnectingWithConnection
    {
        public void DirectConnectionWithIPAddress()
        {
            try
            {
                var connection = new Connection();
                var tracker = connection.Connect("192.168.0.1");
            }
            catch (LmfException ex)
            {
                Console.WriteLine("Could not connect to tracker => " + ex.Description);
            }
        }
    }
}
```

14.2 Connecting via 'TrackerFinder' class

The second code example shows a more convenient way of connecting to a tracker. It first uses an object of type **TrackerFinder** to get a list of all available trackers in the network and you can then choose one to connect to it without directly knowing its IP address. The list of found **TrackerInfo** objects, which can be reached by the property **Trackers**, contain some information about the found trackers (like Name, IPAddress, etc.).

Listing 14.2: Connecting via 'TrackerFinder' class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.ErrorHandling;

namespace SDK.Samples.CodeSamples
{
    class ConnectingWithTrackerFinder
    {
        public void GettingAllTrackersInNetwork()
        {
            // Instantiate the TrackerFinder object
            var found = new TrackerFinder();

            // Check if any tracker have been found in the network
            if (found.Trackers.Count > 0)
            {
                // Take the first one and connect
                var trackerinfo = found.Trackers[0];
                try
                {
                    var tracker = trackerinfo.Connect();
                }
                catch (LmfException ex)
                {
                    Console.WriteLine("Could not connect to tracker => " + ex.Description);
                }
            }
        }
    }
}
```

This method can be used to simple iterate over available trackers and i.e. show them in a specific GUI to let the customer choose a tracker without manually typing in the IP address of the tracker.

Chapter 15

Tracker

15.1 Tracker Object

The Tracker object is the starting point of all operations that can be done with LMF. Most of the commands and properties are located on the Tracker class itself. But if you need to access tracker specific features like for example the GotoStandBy, you will need to cast the Tracker object into the more specific Tracker class. Another common usage is to cast to a tracker family class to get a grouping of trackers belonging to a family if you have more than one family of trackers available.

15.1.1 Class Hierarchy

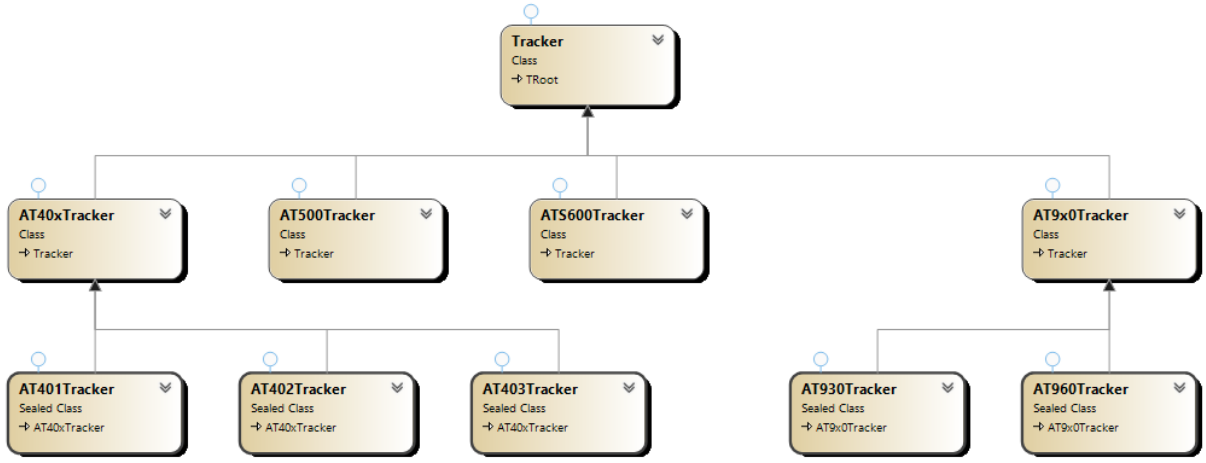


Figure 15.1: Tracker class hierarchy

Listing 15.1: The following example demonstrates how to access Tracker specific features

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;

namespace SDK.Samples.CodeSamples
```

```
{
    public class TrackerTypes
    {
        public void ConnectToTracker()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            //Check if we are connected to a At9x0Tracker
            if (tracker is AT9x0Tracker)
            {
                //Cast the tracker object into the correct Subclass
                AT9x0Tracker at9X0 = (AT9x0Tracker)tracker;

                if (at9X0.QuickRelease.Value)
                {
                    //QuickRelease is closed
                }
                else
                {
                    //QuickRelease is open
                }
            }
        }

    }
}
```

15.2 Commands

Commands in LMF can be executed synchronously or asynchronously. The only difference between the two is that if you have a set of commands, asynchronous commands will execute one after the other without waiting for the previous command to complete, whereas synchronous commands wait for each previous command to complete before executing the next. Synchronous commands have the drawback that control flow does not continue until the command finishes which is not optimal if the command takes long or times out. The asynchronous commands in LMF have the 'Async' postfix to distinct themselves from the synchronous commands.

Listing 15.2: The following example demonstrates the usage of an asynchronous command

```
using LMF.Tracker;
using LMF.Tracker.ErrorHandling;

namespace SDK.Samples.CodeSamples
{
    class AsyncCommand
    {
        private readonly Tracker _Tracker = new Connection().Connect("192.168.0.1");

        public void InitializeAsync()
        {
            _Tracker.InitializeFinished += TrackerOnInitializeFinished;
            _Tracker.InitializeAsync();
        }

        public void InitializeSync()
        {
            _Tracker.Initialize();
```

```
        }

        private void TrackerOnInitializeFinished(Tracker sender, LmfException lmfException)
        {
            // ...
        }
    }
}
```

Chapter 16

Targets

16.1 Overview

To handle the targets there is a class which passes his properties and methods to the other subclasses. Following properties and methods are implemented in the class

- IsSelectable (ReadOnlyBoolValue, Property)
- Select() (Method)

16.1.1 Selected Target

T-Products like the T-Scan, T-Probe or T-Macs are not selectable because they are detected automatically by the tracker. As soon as the tracker locks on a T-Product, the **Selected** property of the targets list will be updated. Reflectors on the other hand need to be selected manually by the user using the **Select()** method of the Target. If the tracker is locked on a T-Product while you select a reflector, it will be preselected instead.

How to get the selected Target

The following code sample shows how to get the currently selected target or preselected reflector of the tracker.

Listing 16.1: How to get the selected target

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.Targets;

namespace SDK.Samples.CodeSamples
{
    class GetSelectedTarget
    {
        public void GetSelectedTargetSample()
        {
```

```
            var tracker = new Connection().Connect("192.168.0.1");

            //Gets the currently preselected Reflector
            Target preSelectedTarget = tracker.Targets.PreSelected;

            //Retrieve the currently selected Target
            Target selectedTarget = tracker.Targets.Selected;

            //Subscribe to the SelectedChanged event to get notified when selected target changes
            tracker.Targets.SelectedChanged += Targets_SelectedChanged;
        }

        void Targets_SelectedChanged(TargetCollection sender, Target target)
        {
            var newTarget = target;
        }
    }
}
```

16.2 Class Hierarchy

16.2.1 Reflectors

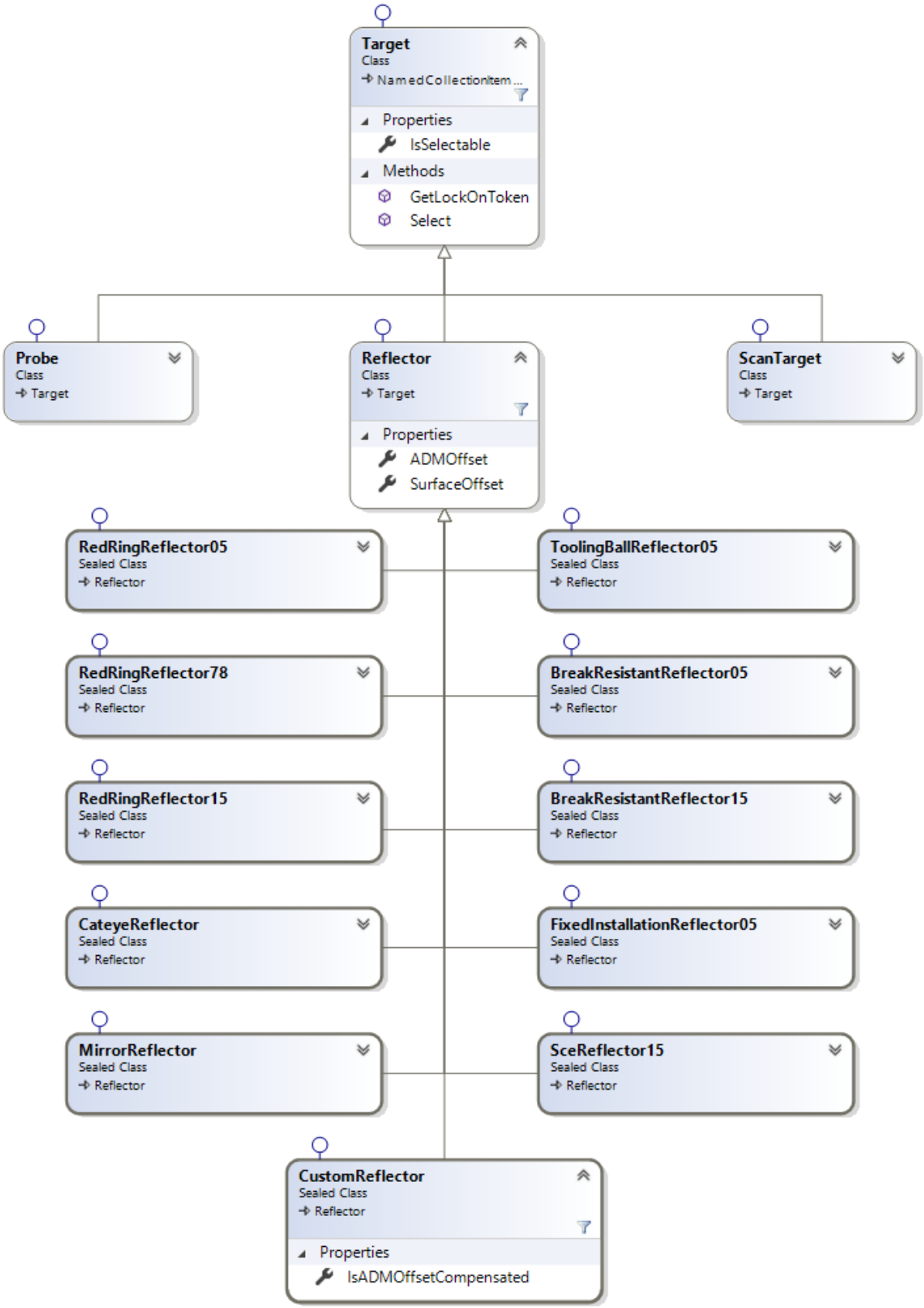


Figure 16.1: Reflector class hierarchy

16.2.2 Probes

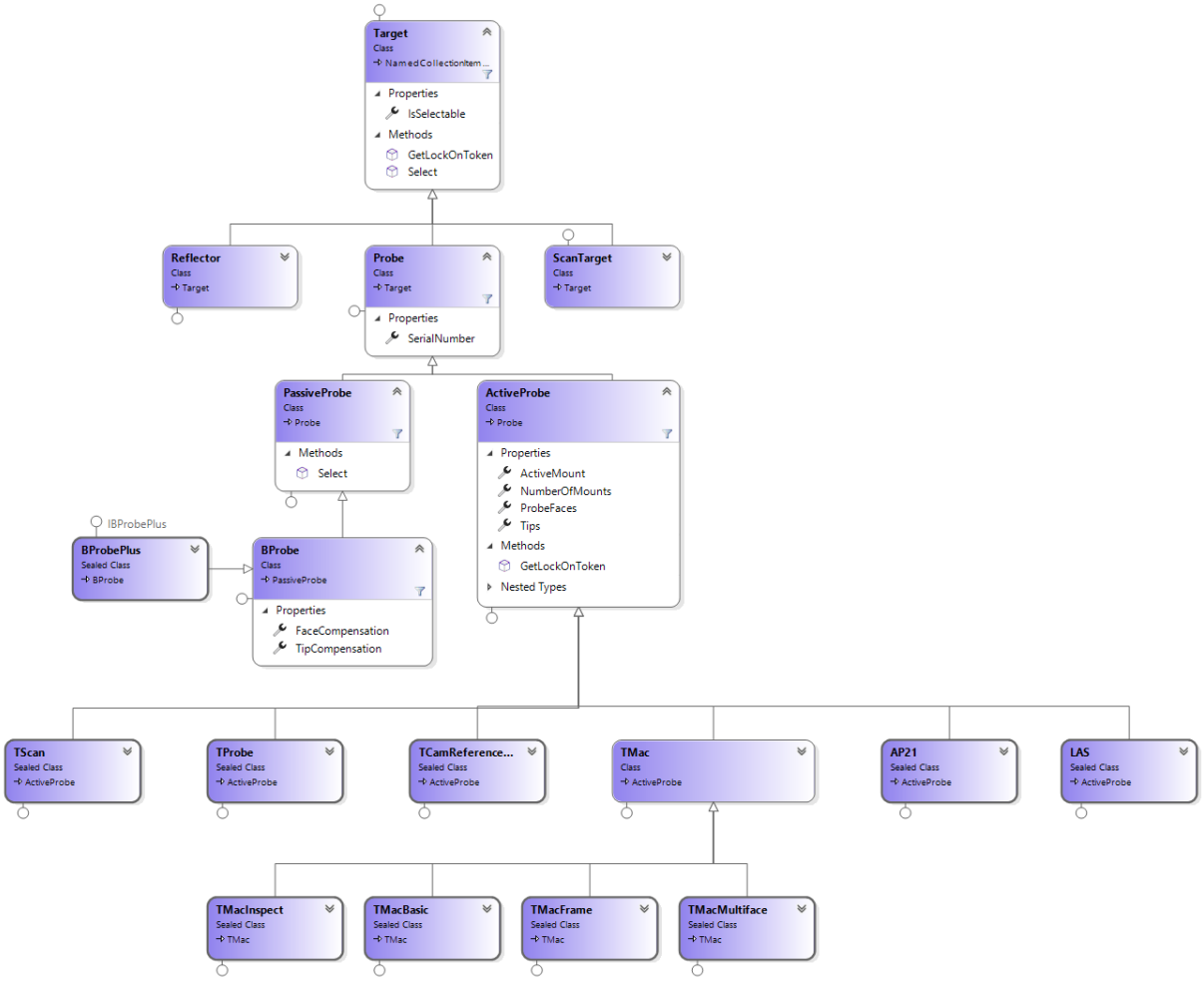


Figure 16.2: Probes class hierarchy

16.2.3 Scan Targets

All scan targets are selectable and therefore need to be selected with **Select()**. **PositionTo()** can be used the same way as with other targets.

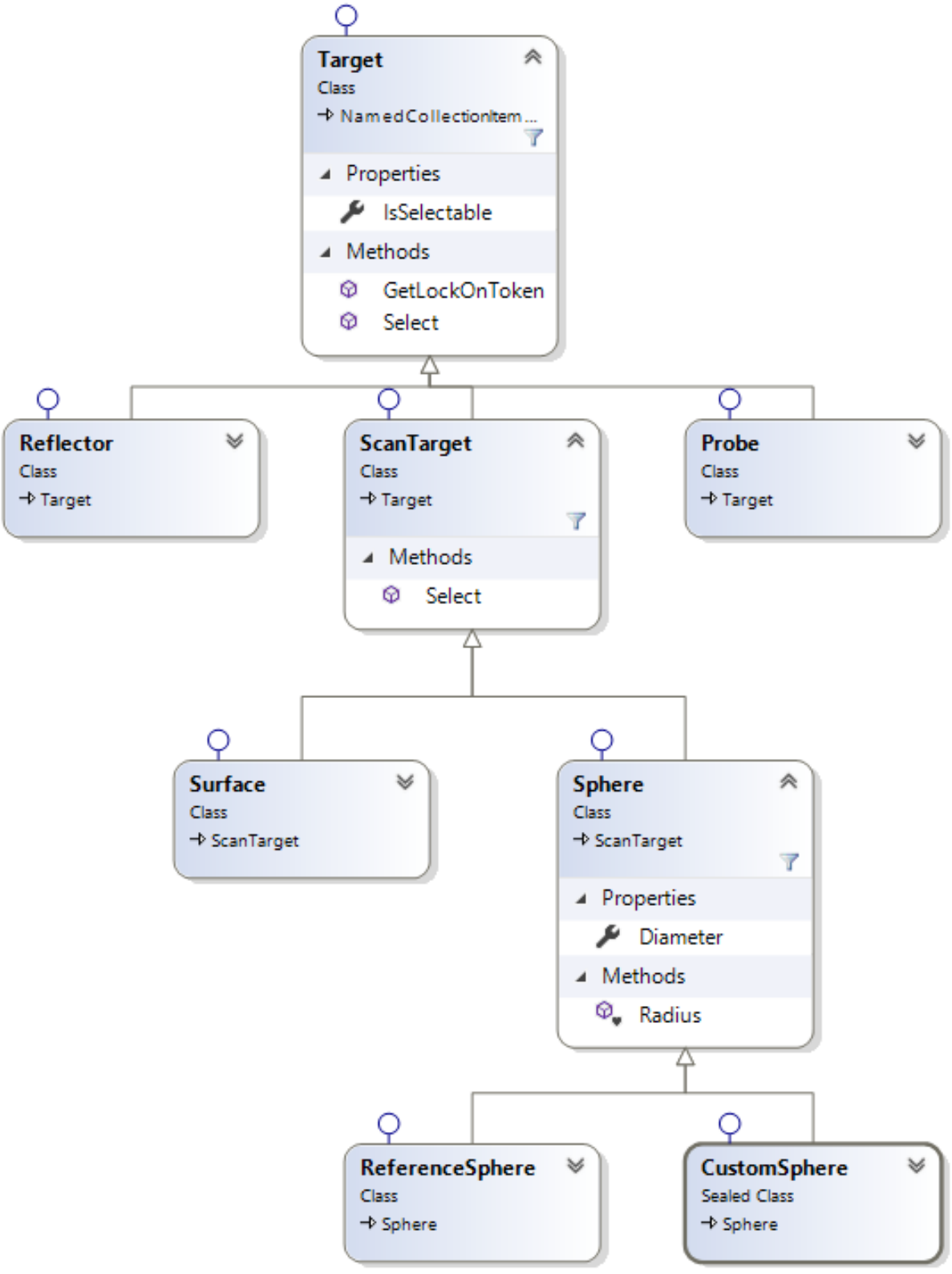


Figure 16.3: Scan targets class hierarchy

16.2.4 Surface

Surface is a default target. Each ATS600 Tracker has one. It is used for scan operations on surfaces. It is the only target which works with the Area Scan Profile.

16.2.5 Spheres

Only ATS600 Tracker are compatible with spheres. To measure a sphere select Stationary Measurement Profile and preferentially place the laser on the center of it.

16.3 B-Probes

The B-Probe / B-Probe ^{plus} is a passive probe and will not automatically be recognised, therefore it needs to be selected like a reflector through the **Select()** method.

16.4 Tips

Tips can be attached to probes allowing the system to provide measurements directly relating to the tip. Fot this reason the ActiveProbe class has a list of Tips. The TipCollection has a selected property which is updated automatically depending on the attached tip or the selected virtual tip. The Tip itself consists out of a translation a balldiameter and a length.

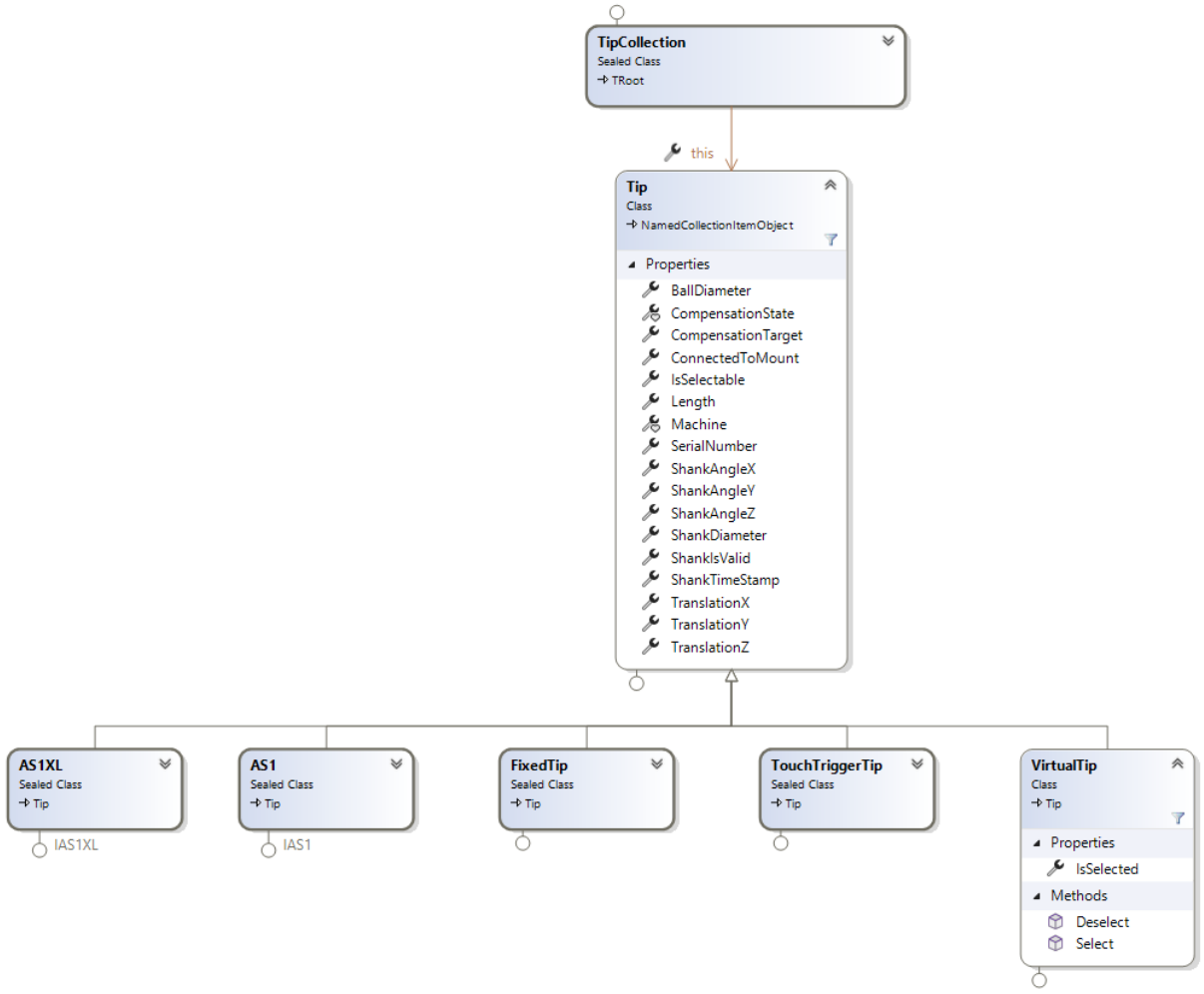


Figure 16.4: Tip class hierarchy

16.4.1 Virtual Tips

Virtual tips can be used in cases where no automatically detectable tip is attached. They are created and managed by the TrackerPilot and can be selected using LMF.
Virtual Tips have the following additional properties and methods

- Select() - Tries to select a Virtual Tip, if the Probe is not locked on it will preselect the Virtual Tip.
- Deselect() - Deselects this Virtual Tip. This will automatically activate any connected Tip of the currently active Probe.
- IsSelected - Returns true if the Virtual Tip is currently selected

The following code snippet shows how to check which tip is selected and how to select a different virtual tip.

Listing 16.2: How to use Tips

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.Targets;
using LMF.Tracker.Targets.Probes.ActiveProbes;
using LMF.Tracker.Targets.Probes.ActiveProbes.Tips;

namespace SDK.Samples.CodeSamples
{
    class TipSample
    {
        public void GetSelectedTipSample()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            var probe = tracker.Targets.Selected as ActiveProbe;

            if (probe != null)
            {
                //Get the selected tip of the active probe
                var tip = probe.Tips.Selected;
            }

            tracker.Targets.ActiveTipChanged += Targets_ActiveTipChanged;
        }

        public void SelectAVirtualTip()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            var probe = tracker.Targets.Selected as ActiveProbe;

            if (probe != null)
            {
                // Iterate over all Tips of this Probe
                foreach (VirtualTip virtualTip in probe.Tips.OfType<VirtualTip>())
                {
                    //Get the Tip you want to select
                    if (virtualTip.Name == "MyTip")
                    {
                        //Select the virtual Tip
                        virtualTip.Select();
                    }

                    //Use Tip.IsSelected to check wether a VirtualTip is preselected
                    //Use Tip.CompensationTarget to get the probe face the tip is compensated for.
                    if (virtualTip.IsSelected.Value)
                    {
                        Debug.WriteLine("VirtualTip " + virtualTip.Name +" is Selected for " +virtualTip.↩
                            CompensationTarget);
                    }
                }
            }

            tracker.Targets.ActiveTipChanged += Targets_ActiveTipChanged;
        }

        private void Targets_ActiveTipChanged(Tracker sender, Tip tip)
        {
            //The currently active Tip has changed.
        }
```

```
    }
}
```

- Virtual Tips can be selected on probes that are not connected to the tracker.
- It's possible to select different virtual tips for the different faces of a T-Mac.
- The virtual tip selections stay active, even if you reconnect to the tracker.

16.4.2 Measurement without Tip

When working with T-Macs, its possible to measure without any Tip connected or Virtual Tip selected. Measurements on probes without a real tip attached is in most of the cases possible but can sometimes be prohibited, depending on the probe type and the system software used. If that is the case a measurement impediment will indicate that a tip must be used to the probe. If attaching a real tip is not possible a virtual tip can be defined and selected on this probe to still allow measurements without a real tip attached.

16.5 TargetSearch

Finding a target is accomplished using the TargetSearch class. The following properties are provided and help the search process.

- ApproximateDistance
- Radius
- Timeout

Please note, that the rectangular or spiral search is only performed if PowerLock is off or the target is at a distance where it works better than PowerLock. If scan targets are selected, **TargetSearch** will have no effect. Spheres are not detectable and surface can point on everything.

Listing 16.3: How to find a target over TargetSearch

```
using System.Linq;
using LMF.Tracker;
using LMF.Tracker.ErrorHandling;
using LMF.Tracker.Targets;

namespace SDK.Samples.CodeSamples
{
    class TargetSearch
    {
        public void SearchTarget()
        {
            var tracker = new Connection().Connect("192.168.0.1");
            tracker.TargetSearch.Radius.Value = 5;
            tracker.TargetSearch.Timeout.Value = 12000;
```

```
            tracker.TargetSearch.ApproximateDistance.Value = 1;
            tracker.TargetSearch.StartAsync();

            tracker.TargetSearch.Finished += TargetSearchFinished;
        }

        private void TargetSearchFinished(LMF.Tracker.Targets.TargetSearch sender, Target ↩
            foundTarget, LmfException lmfException)
        {
            var target = foundTarget;
        }
    }
}
```

Chapter 17

Positioning

17.1 PositionTo

The PositionTo command can be used to send the laser beam to a specific position. It has the following parameters:

17.1.1 SearchTarget

If set to true, the tracker will search for a target after the specified position has been reached. Powerlock will be used to find the Target. If Powerlock is disabled a spiral search will be performed.

17.1.2 IsRelative

Determines whether the specified position is interpreted as relative to the absolute.

17.1.3 Pos1,Pos2,Pos3

The desired position using the coordinate system and unit settings as specified on the tracker settings object. If **IsRelative** is set to true, the specified position is required to use the spherical coordinate system.

17.1.4 Return Value

In case the target search is active the command either returns with the detected target or an exception. In case the target search is not active the command returns with the detected target or null to indicate that no target has been found.

Listing 17.1: PositionTo Sample

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
```

```
using LMF.Units;

namespace SDK.Samples.CodeSamples
{
    class PositionTo
    {
        public void Example1()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            tracker.Settings.CoordinateType = ECoordinateType.Spherical;
            tracker.Settings.Units.AngleUnit = EAngleUnit.Degree;

            // Positions the Tracker relative by 90 degrees on the horizontal axis without locking on
            tracker.PositionTo(false, true, 90, 0, 0);

            // Positions the Tracker absolute to the position 30,20,5000 (H,V,D) and searches for a Target.
            tracker.PositionTo(true, false, 30, 20, 5000);
        }

    }
}
```

17.2 PositionToTarget

This command is similar to the PositionTo command with the difference that it always does the target search and sets the preselected target to the one provided through the LockOnToken. Explicitly setting a preselected target before the positioning is not required with this command. An active probe is detected and cannot be preselected. So it is not preselected in that case. The command is not suitable for a positioning without locking onto a target, therefore. Using this command the lock-on process is accelerated by providing a LockOnToken that directly tells the tracker what the type of the target to lock on is. The acceleration is only possible if a valid LockOnToken matching the real target at the given position is provided. Otherwise the standard (slower) lock-on process is used as fallback.

17.2.1 Parameter lockOnToken

The LockOnToken is an object that can be obtained from the tracker tree by either calling GetLockOnToken on a target or by calling CreateLockOnToken on the TargetCollection. Using this token the tracker is able to do a faster lock-on process in case it is valid and matches the target at the provided position.

17.2.2 Other Parameters

All other parameters are the same as for the PositionTo command.

17.2.3 Return Value

This command always returns an exception or a detected target. In case of an active probe (e.g. T-Probe) it is the probe that is detected by the tracker. In case of a reflector

or a passive probe (e.g. B-Probe) it is the target that is preselected as the tracker cannot detect the reflector type or the type of a passive probe.

17.3 PositionTo vs PositionToTarget

Both commands may be used to move the tracker to a given position but they slightly differ in their behavior. Consider the following differences before choosing one of them:

- PositionToTarget always searches for a valid target using the target search while PositionTo can be used for both: Positioning with and without locking onto a target.
- PositionToTarget does an optimized lock-on process if a valid LockOnToken is provided while PositionTo always uses the standard lock-on process.
- PositionToTarget adapts the preselected target to the one denoted by the LockOnToken while PositionTo never changes the preselected target. Using PositionToTarget does not require to select the corresponding target before doing the positioning, therefore.

17.4 Positioning and Lock-On Process

Using the commands PositionTo and PositionToTarget the tracker can be moved to a given position. In case a target is detected at this position or near to it the tracker locks onto this target. Depending on the type and the position of the target the lock-on process is immediately started or delayed until a target is found by the target search (Power Lock or Spiral Search). These following cases are to be handled correctly:

- Precise target position: The given coordinates move the tracker closely enough to the target that the laser is pointing to the reflector center of the target. In that case the lock-on process may immediately start once the given position is reached. So in case of an asynchronous call to PositionTo or PositionToTarget the event telling that the tracker is ready to measure is arriving almost at the same time as the PositionTo(Target)FinishedEvent is arriving. There arrival order in the client application is non-deterministic, therefore. Do not rely on any order of them.
- Imprecise target position: The given coordinates move the tracker to a place where the laser is not pointing to the reflector center of the target but either the Spiral Search or the Power Lock is able to detect a target that is near and moves the tracker to it. In that case the lock-on process cannot immediately start once the positioning is finished. In case of an asynchronous call to PositionTo or PositionToTarget the event telling that the tracker is ready to measure may arrive some time after the PositionTo(Target)FinishedEvent is arriving as the tracker needs to find the target first.

- No target found: The given coordinates move the tracker to a position where no target can be found. So in case of an asynchronous call to PositionTo or PositionTo-Target only the PositionTo(Target)FinishedEvent is arriving. In case a target search is done (always the case for PositionToTarget) the event returns an exception.

17.4.1 3D vs. 6D Lock-on Process

Using the command PositionToTarget the tracker always locks onto a target. Some targets (Reflectors and passive probes) may be selected and are represented by the preselected target. Other targets (Active probes) are detected by the tracker and cannot be manually selected. The interesting special case is when the tracker is moved to an active probe using the command PositionToTarget but the provided LockOnToken denotes a reflector. For the command PositionToTarget the detected active probe overrides the reflector denoted by the given LockOnToken. The returned target is the active probe then. For the command GoAndMeasureStationary this is not the case: Here the active probe is measured as the preselected reflector preventing the tracker from doing any probe detection to allow the maximum speed.

17.5 GoAndMeasureStationary

This command combines the two actions:

- Positioning as done by the PositionToTarget command
- Stationary measurement

into a single command to speed up the whole procedure. The parameters are the same as for the PositionToTarget command. But the return value is different: It returns a stationary measurement taken at the given position. In case a stationary measurement profile is selected this profile and its settings are used for the measurement. In case a different profile is selected the command takes the first available stationary measurement and its settings for the measurement. During the command execution the selected profile is changed in this case. But once the command is finished the originally selected profile is selected again. The command behaves slightly different depending on the LockOnToken:

- Denoting a reflector by the LockOnToken prevents the tracker from detecting any active probe to speed up the target search to the maximum (Even more than for PositionToTarget where the standard target search is available as fallback option to always detect any present active probe). So the possibly present active probe is measured as the preselected reflector resulting in an unusable measurement in that case.
- Providing no LockOnToken (null) lets the tracker do the standard target search, which is not optimized but recognizes all targets correctly.
- Denoting an active probe by the LockOnToken lets the tracker do the same optimized target search as for PositionToTarget that allows to fallback to another target in case another one is present.

For the trackers that are not able to detect any target itself these differences are not relevant. But the command still adapts the preselected target to the one denoted by the LockOnToken. So no explicit selection is required before. The command allows to do a fast autoinspect loop if all targets to be measured are well-known.

17.6 GoHomePosition

GoHomePosition sends the laser beam to the 6Dof zero position.

Chapter 18

Triggers

18.1 TriggerCollection

The TriggerHappened Eventhandler is used to receive events of every enabled trigger in the collection. For example if you wan't to receive notifications of probe button events, you only need to enable the specific trigger and register to the Collection TriggerHappened event like the following code illustrates.

Listing 18.1: How to receive Probe Button events

```
using System.Linq;
using LMF.Tracker;
using LMF.Tracker.Triggers;

namespace SDK.Samples.CodeSamples
{
    class HandlingTriggers
    {
        public void Example1()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            tracker.Triggers.OfType<ProbeButtonTrigger>().First().IsEnabled.Value = true;
            tracker.Triggers.TriggerHappened += OnTriggerHappened;
        }

        private void OnTriggerHappened(Trigger trigger, TriggerEventData data)
        {
            if (trigger is ProbeButtonTrigger)
            {
                var button = data.Button;
            }
        }
    }
}
```

18.2 Trigger Types

18.2.1 ProbeButton

T-Probe buttons down and up click events allow to perform user defined actions on AT960 Tracker. With AT960 Trackers an additional double click event can be used.

B-Probe ^{plus} buttons down click events allow to perform user defined actions on AT500 Tracker.

18.2.2 RemoteControl

Remote Control buttons down click events allow to perform user defined actions on AT40x Tracker.

18.2.3 StableProbing

Stable Probing events are used to trigger a measurement when the target is treated as stable. Typically a measurement, either stationary or continuous is then performed to allow measurement acquisition without the need of a remote control.

Chapter 19

Measurements

19.1 Measurement Status

The tracker can have three different measurement states.

- ReadyToMeasure
- NotReady
- MeasurementInProgress
- Invalid

These states indicate if the tracker is ready to measure or not. You can register yourself to a changed event, so you are notified as soon as the measurement status has changed.

19.1.1 Measurement Preconditions

There can be many different reasons why a tracker is not ready to measure. In order to identify the exact reasons you can access a list of preconditions that need to be fullfilled before you are able to measure.
For example if you start the tracker, the measurement status will be NotReady with a precondition that the tracker needs to be initialized before you are able to measure. As soon as you initialize the tracker the measurement status will change to ReadyToMeasure if there are no more preconditions.
Each precondition has a unique number, a title, a description and a solution text. In the following sample code you can see how to access the measurement status and the measurement preconditions.

Listing 19.1: How to use the Measurement Status

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.Enums;
```

```
using LMF.Tracker.MeasurementResults;
using LMF.Tracker.Measurements.Profiles;

namespace SDK.Samples.CodeSamples
{
    class UsingTheMeasurementStatus
    {
        public void QueryMeasurementStatus()
        {
            //Connect to a Tracker
            var tracker = new Connection().Connect("192.168.0.1");

            //Get the measurement status
            var status = tracker.Measurement.Status.Value;

            //Print the details of the first preconditions if the measurement status is NotReady
            if (status == EMeasurementStatus.NotReady)
            {
                var firstPrecondition = tracker.Measurement.Status.Preconditions[0];
                Console.WriteLine(firstPrecondition.Number);
                Console.WriteLine(firstPrecondition.Title);
                Console.WriteLine(firstPrecondition.Description);
                Console.WriteLine(firstPrecondition.Solution);
            }

        }
    }
}
```

19.2 Measurement Profiles

There are multiple profiles for different measurement applications you can choose from. Each measurement profile has its own settings. In order to set a measurement profile active you have to call the select method on the profile you want to select, this will automatically stop the current measurement process.

19.2.1 StationaryProfile

The StationaryProfile is used to do stationary measurements with a specific accuracy. The measurement results will be 6D or 3D depending on the selected Target.

Accuracy

There are three levels of accuracy you can choose from.

- Fast
- Standard
- Precise

19.2.2 OutdoorProfile

The OutdoorProfile is used to do stationary outdoor measurements. The measurement results will be 6D or 3D depending on the selected Target. This measurement profile is only available when working with At40x trackers.

19.2.3 ContinuousTimeProfile

The ContinuousTimeProfile is used to do dynamic measurements with a given time separation. For example if you use a time separation of 20ms the Tracker will deliver 50 measurements per second. However, LMF does not trigger an event for each single measurement, they are grouped together and delivered at a rate of 5 times per second.

TimeSeparation

Time Separation between the measurements

19.2.4 ContinuousDistanceProfile

The ContinuousDistanceProfile is also used to do dynamic measurements with a given distance separation. This means if you move the target for set distance a measurement will be triggered automatically.

DistanceSeparation

Distance between the Measurements

19.2.5 TouchTriggerProfile

The TouchTriggerProfile is used for TMacs with an attached Touchtrigger. If this profile is selected each touch with the TouchTrigger stylus will produce a measurement.

19.2.6 CustomTriggerProfile

The CustomTriggerProfile is used in automation scenarios where external trigger signals are used to trigger single shot measurements on the tracker. Detailed information to the hardware are found in the **Appendix External Trigger Interface.pdf** in the SDK folder. Several properties can be configured to match the hardware trigger environment of the customer setup.

ClockSource

Choose between *internal* and *external* source for the trigger clock

ClockTransmission

Defines if the clock signal triggers a measurement on the *negative* or *positive* edge of the signal

MinimalTimeDelay

Defines the minimal time after the last measurement in which a new triggers signal does not trigger a new measurement

StartStopActiveLevel

Defines if measurements are allowed when the startstop signal is on *high* or *low* level

StartStopSource

Defines if the startstop signal is *active* or *ignored*. Depending on the source the *FirstMeasurementAfterSignal* of the *MeasurementInfo* is flagged differently. If the source is active, the first measurement after receiving the start signal of a transition is flagged as true, else the first of the incomming stream.

TriggeredMeasurementsArrived

IMPORTANT: All measurements taken with CustomTriggerProfile selected are coming via a separate event called **TriggeredMeasurementsArrived**.
TriggeredMeasurementsArrived delivers measurements of the type **TriggeredMeasurement**. Unlike other measurement results, the **TriggeredMeasurement** is independent of the coordinate / rotation type and the unit settings of the tracker. The position and rotation of a TriggeredMeasurement are always using the settings below:

- Position: HVD (meters and radians)
- Rotation: Quaternions

19.3 Measurement Results

Every measurement taken by the tracker object will result in a resulting object of base class type **Measurement**. Dependent if the measurement has been stationary/single shot, 3D/6D or simple/extended the user has to cast the individual measurement into its specialized instance. The LMF.Tracker framework support differentiation into the following measurement types:

- 3D & 6D measurements
- Stationary & single shot measurements

19.3.1 3D and 6D

This differentiation separates measurements into objects with / without 6D information based on which target type the measurement has been taken. For example, measurements taken on reflectors are always 3D measurements, as far as probes usually deliver 6D measurements. In some special cases there can be both types (3D/6D) of measurements on the same probe, i.e. if one instructs the framework to measure onto the prism of a TProbe/TMac for example, which would results in a 3D measurement on a 6D capable probe.
In general do all 3D measurements have information about its position in the coordinate system, whereas 6D measurement complete this by information about the rotation around the position in space.

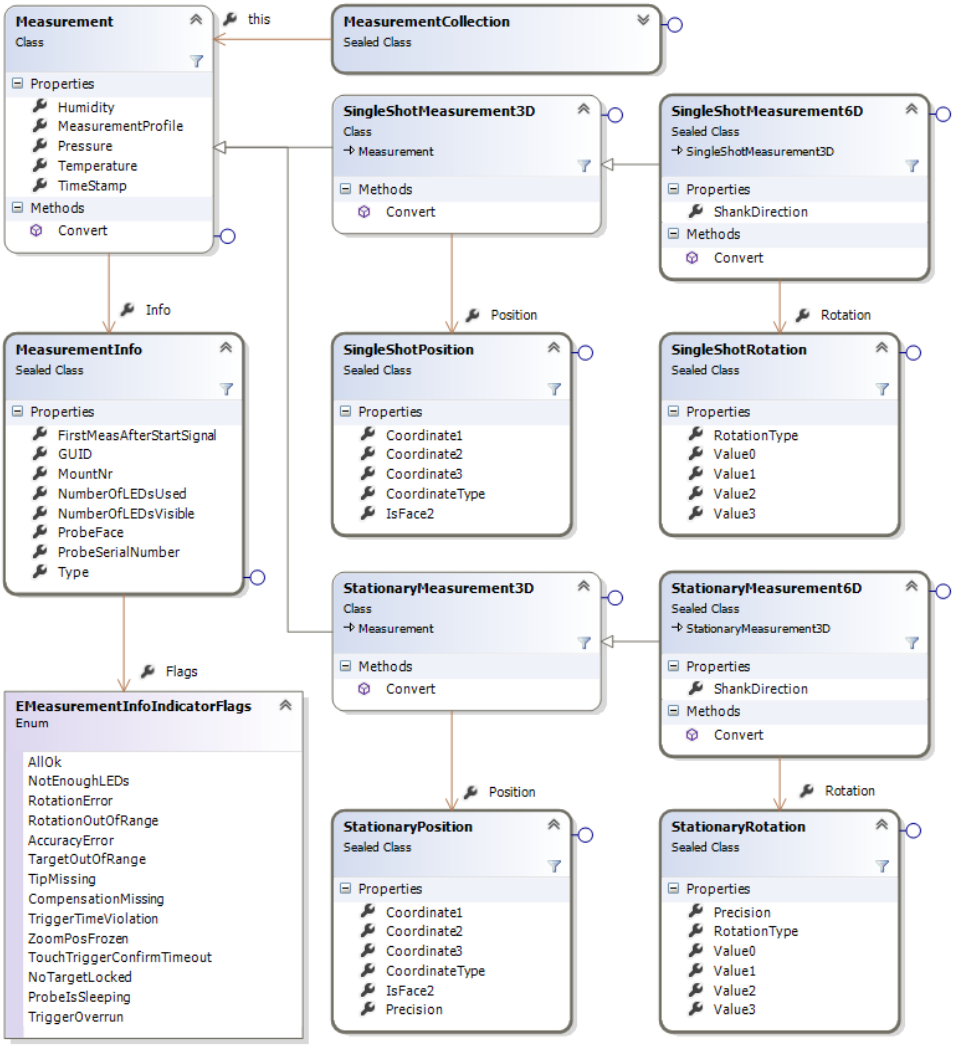


Figure 19.1: Measurement Structure

Besides the 6DoF rotation matrix, a ShankDirection property is provided that delivers the UnitVector of the currently detected tip. The Unit vector is representing direction of shank (pointing from tip along shank) with respect to tracker coordinate system and takes

orientation and transformation parameters which are activated in LMF into account. This is also considering a shank compensation if available.

Figure 19.2: Shank Direction

19.3.2 Stationary and Continuous

The LMF.Tracker framework is capable of basically doing two different kind of measurements: Single Shots and Stationary. Single shot measurements are - as the name suggests - measurements of one single shot in the smallest possible time resolution the tracker is capable of. They deliver exactly one complete measurement datum. On the other side there are stationary measurements which represent an averaged measurement over a longer period of time based on multiple single shot measurements. For this purpose they have additional information with them to represent the statistical data calculated in averaging these measurements. The Precision represents the spatial uncertainty of a stationary point measurement. It is calculated as an RMS (Root Mean Square) based on the H, V and D samples within integration time.

19.3.3 MeasurementInfo

The MeasurementInfo Object holds additional information about the validity of each measurement. This information is mainly needed for external triggered measurments, because stationary or continuous measurements are always valid.

- Type (Empty, NotLocked, Measurement3D, Measurement6D)
- Flags (See EMeasurementInfoIndicatorFlags)

- GUID (This matches the GUID of the selected Reflector, the selected Probe or the active Tip, depending on the measurement setup)
- ProbeSerialNumber
- ProbeFace
- MountNr
- NumberOfLEDsVisible
- NumberOfLEDsUsed
- FirstMeasAfterStartSignal

EMeasurementInfoIndicatorFlags

This bitflag indicates wether a measurement is valid or not and contains the corresponding reasons.

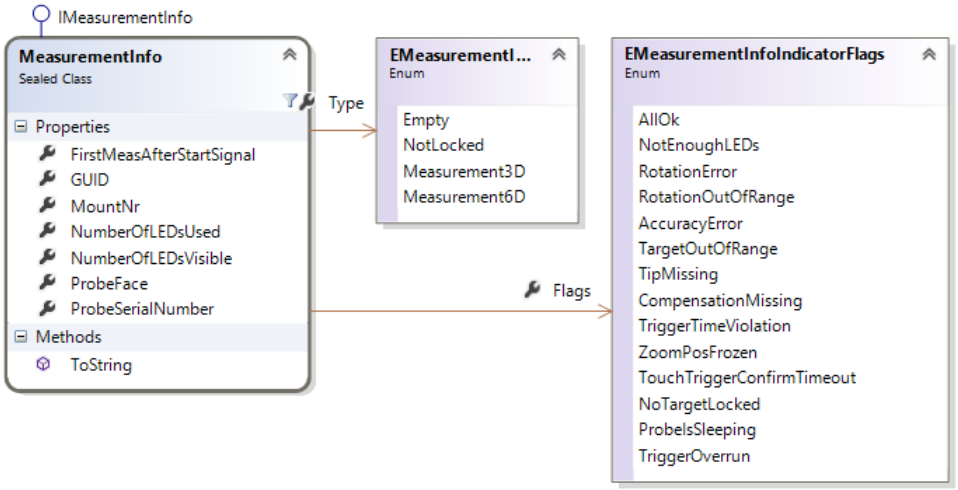


Figure 19.3: Measurement Info

Listing 19.2: How to use the MeasurementInfoIndicatorFlags

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.MeasurementResults;
using LMF.Tracker.Measurements.Profiles;

namespace SDK.Samples.CodeSamples
{
    class MeasurementBitFlags
    {
        public void Measure()
        {
```

```
            var tracker = new Connection().Connect("192.168.0.1");

            var m = tracker.Measurement.MeasureStationary() as StationaryMeasurement3D;

            //Use bitwise operations to check wether a flag is set or not.
            if ((m.Info.Flags & EMeasurementInfoIndicatorFlags.AllOk) == ↩
                EMeasurementInfoIndicatorFlags.AllOk)
            {
                //Measurement is Valid
            }
            else
            {
                //Measurement is not Valid
            }

            if ((m.Info.Flags & EMeasurementInfoIndicatorFlags.CompensationMissing) == ↩
                EMeasurementInfoIndicatorFlags.CompensationMissing)
            {
                //The CompensationMissing Flag is set, this means the Probe or Tracker compensation is missing.
            }

        }

    }
}
```

19.4 Taking Measurements

In order to start a measurement process you have to select a measurementprofile first. There are multiple measurement profiles for different applications. (See 19.2)

19.4.1 Synchronous Measurements

Listing 19.3: My first measurement

```
using System;
using System.Linq;
using LMF.Tracker;
using LMF.Tracker.Measurements;
using LMF.Tracker.Measurements.Profiles;
using LMF.Tracker.MeasurementResults;

namespace SDK.Samples.CodeSamples
{
    class TakingFirstMeasurement
    {
        public void Measure()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            //   Initialize the tracker before the first measurement
            tracker.Initialize();

            // Select profile
            tracker.Measurement.Profiles.OfType<StationaryMeasurementProfile>().First().Select();

            //  Now we are ready to measure
            var m = tracker.Measurement.MeasureStationary() as StationaryMeasurement3D;
```

```
            // Print out some measurement data
            Console.WriteLine("Position X" + m.Position.Coordinate1.Value);
            Console.WriteLine("Position Y" + m.Position.Coordinate2.Value);
            Console.WriteLine("Position Z" + m.Position.Coordinate3.Value);
        }
    }
}
```

19.4.2 Asynchronous Measurements

Listing 19.4: My first async measurements

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.MeasurementResults;
using LMF.Tracker.Measurements.Profiles;

namespace SDK.Samples.CodeSamples
{
    class TakingFirstMeasurementAsync
    {
        public void MeasureAsync()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            //  Initialize the tracker before the first measurement
            tracker.Initialize();

            // Select profile
            tracker.Measurement.Profiles.OfType<ContinuousTimeProfile>().First().Select();

            tracker.Measurement.MeasurementArrived += Measurement_MeasurementArrived;
            // Now we are ready to measure
            tracker.Measurement.StartMeasurement();

            // Do something in the meantime ...


            // Stop the measurements
            tracker.Measurement.StopMeasurement();
        }

        void Measurement_MeasurementArrived(LMF.Tracker.Measurements.MeasurementSettings sender, ↩
             MeasurementCollection measurements, LMF.Tracker.ErrorHandling.LmfException exception)
        {
            foreach (var measurement in measurements)
            {
                if (measurement is SingleShotMeasurement3D)
                {
                    var m = (SingleShotMeasurement3D) measurement;

                    // Print out some measurement data
                    Console.WriteLine("Position X" + m.Position.Coordinate1.Value);
                    Console.WriteLine("Position Y" + m.Position.Coordinate2.Value);
                    Console.WriteLine("Position Z" + m.Position.Coordinate3.Value);
                }
            }
        }
    }
```

```
}
```

Chapter 20

Overview Camera

The LMF offers a predefined dialog that can be called to locate targets in the field of view of the tracker. The dialog has all the needed functions like point and click or the capability to save images already implemented. Alternatively a independent video stream can be used. The following examples show the usage in case of doing the processing of the incoming video stream on your own and using the predefined standard OVC dialog to be used directly in your applications:

Listing 20.1: OVC common use cases

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.Enums;

namespace SDK.Samples.CodeSamples
{
    class OVCSample
    {
        public void Example1()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            // Notify me when the next image frame comes in
            tracker.OverviewCamera.ImageArrived += OverviewCamera_ImageArrived;

            // Get the overview camera running
            tracker.OverviewCamera.StartAsync();

            // Change some settings
            tracker.OverviewCamera.Brightness.Value = 70;
            tracker.OverviewCamera.Contrast.Value = 60;

            // Get an high resolution screen shot of the overview camera
            var highResImage = tracker.OverviewCamera.GetStillImage(EStillImageMode.High);

            // Stop the overview camera
            tracker.OverviewCamera.Stop();
        }

        void OverviewCamera_ImageArrived(LMF.Tracker.OVC.OverviewCamera sender, ref byte[] image, LMF.↩
            Tracker.OVC.ATRCoordinateCollection atrCoordinates)
```

```
        {
            //Do something with the "image" data ...
        }

        public void OVCDialogSample()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            // The standard LMF overview camera dialog can be displayed in many different ways:

            // Non blocking, asynchronously ...

            // Show the ovc window as child of the actual process
            tracker.OverviewCamera.Dialog.Show();
            // Do something ...
            tracker.OverviewCamera.Dialog.Close();

            // Show the ovc window as child on the process with pid = 345 <- should be from your desired ↩
                 process
            var pid = Process.GetProcessById(345).Id;
            tracker.OverviewCamera.Dialog.ShowOnProcess(pid);
            // Do something ...
            tracker.OverviewCamera.Dialog.Close();

            // Show the ovc window NOT as child of actual process, BUT topmost over all other windows
            tracker.OverviewCamera.Dialog.ShowTopmost();
            // Do something ...
            tracker.OverviewCamera.Dialog.Close();


            // Blocking, synchronously ...

            // Show as blocking child of the actual process
            tracker.OverviewCamera.Dialog.ShowDialog();
            // Do something after the dialog has closed
        }
    }
}
```

Chapter 21

Face

The Face shows the orientation of the tracker head. It can be Face I or Face II. There are two possibilities to change it, either with the change command or by turning the tracker head manually. To receive the information when it's changed, register to the Changed event. The following code sample shows the common handling.

Listing 21.1: Face examples

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;

namespace SDK.Samples.CodeSamples
{
    class FaceExample
    {
        public void Example1()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            Console.WriteLine("The tracker head is actualy in: " + tracker.Face.Value);
            Console.WriteLine("The actual face is the first face: " + tracker.Face.IsFace1);

            // Register to the "Changed" event (e.g. turn the tracker head vertically into the other face by ↩
                hand)
            tracker.Face.Changed += new Face.ChangedHandler(Face_Changed);

            // Register to the "ChangeFinished" event ...
            tracker.Face.ChangeFinished += new Face.ChangeFinishedHandler(Face_ChangeFinished);

            // ... and change the tracker to the other face
            tracker.Face.Change();

            // Do something in the other face, typically take a measurement ...

            // ... and change the tracker back to the original face
            tracker.Face.Change();
        }

        void Face_ChangeFinished(Face sender, LMF.Tracker.Enums.EFace newvalue, LMF.Tracker.↩
            ErrorHandling.LmfException ex)
        {
            if (ex != null)
            {
```

```
                // There has been some error here
            }

            Console.WriteLine("The tracker finished turning into the other face and is now in face: " + ↩
                newvalue);
        }

        void Face_Changed(Face sender, LMF.Tracker.Enums.EFace newvalue)
        {
            Console.WriteLine("Tracker head changed into face: " + newvalue);
        }
    }
}
```

Chapter 22

Laser and System shutdown behaviour

22.1 Laser

The Laser object can be used for turning on or shutting down the laser beam of the laser tracker. Also you can use the Laser object to define when the laser beam has to go into the sleep mode. When working with AT40x trackers the laser object is null and cannot be accessed, use **Tracker.GotoStandBy** instead.

22.1.1 Turn on/shut down the laser beam manually

In the following code sample you can look up how the turn on/shut down the laser beam:

Listing 22.1: Laser examples I

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;

namespace SDK.Samples.CodeSamples
{
    class LaserManually
    {
        public void ExampleLaserOnOffManually()
        {
            //connect to a tracker
            var tracker = new Connection().Connect("192.168.0.1");

            // initialize the laser object
            var laser = tracker.Laser;

            //turn the laser on (default value)
            laser.IsOn.Value = true;

            //turn the laser off
            laser.IsOn.Value = false;
        }
    }
}
```

22.1.2 Turn on/shut down the laser beam with a timer

As a developer you have also the possibility to shut down the laser beam with a timer which determines when the laser beam has to wake up.
NOTE: If you are shutting down or turning on the laser beam manually the timer will be reset.

Listing 22.2: Laser examples II

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;

namespace SDK.Samples.CodeSamples
{
    class LaserOnTimer
    {
        public void ExampleLaserOnTimer()
        {
            //connect to a tracker
            var tracker = new Connection().Connect("192.168.0.1");

            // initialize the laser object
            var laser = tracker.Laser;

            //turn off the laser and set a wake up time when the laser has to turn on
            //it is needed to set the parameter "wakeUpTime" with a DateTime-object
            laser.GoToSleep(DateTime.Now);

            //detect the wake up time of the laser (DateTime-object)
            DateTime wakeUpTime = laser.WakeUpTime;
        }
    }
}
```

22.2 Tracker

22.2.1 Shutdown

Tracker.Shutdown() can be used to completely shutdown the tracker and the controller.

22.2.2 GoToStandby

AT40xTracker.GoToStandby(DateTime wakeupTime) sets the system into a standby mode and restarts it automatically at the specified time. This function is exclusive to the AT40x tracker types.

Chapter 23

Quick Release

The Quick Release connects the sensor with the stand. To grant stability it needs to be closed. Therefore the Quick Release has two states closed and open. It can be only accessed on an ATS600 and AT9x0 Tracker. A warning is thrown if its state changes to open while connected to a Tracker. Make sure to check the state after connecting with a Tracker, so the user is alarmed if the Quick Release is open.

Chapter 24

Power Source

The LMF offers the information with which power source the tracker is running. The user can work with the power supply or with a battery.
To detect which power source is connected there is a property on the **Tracker.PowerSource.ControllerPowerStatus** object

- RunsOn

which delivers a EPowerSource (Enum) with three possible variations:

- EPowerSource.Battery
- EPowerSource.Mains
- EPowerSource.BatteryProblem

Also there is the possibility to detect how the level of the battery is if there is one connected. You have also to look in the **Tracker.PowerSource.ControllerPowerStatus** object to find the property:

- Level (**ReadOnlyDoubleValue**) -> unit type is percentage

Chapter 25

Inclination Sensor

25.1 Orient to Gravity

The Orient to Gravity (OTG) sets the Tracker to gravity. As a result, the OTG process delivers the rotation angles InclinationRotX and InclinationRotY representing the position of the Tracker to the gravity plane. Typically, InclinationRotX and InclinationRotY are small angles as the internal nivel only delivers values within working range.

Listing 25.1: GetInclinationToGravity

```
Tracker.InclinationSensor.GetInclinationToGravity()
```

GetInclinationToGravity starts the OTG process. The internal nivel readings are taken at 4 different positions (0, 90, 180, 270 degree), resulting in the CurrentInclinationToGravity result with the rotation angles InclinationRotX and InclinationRotY.

Listing 25.2: GetInclinationToGravityAsync

```
Tracker.InclinationSensor.GetInclinationToGravityAsync()
```

The same as GetInclinationToGravity() but asynchronous.

Listing 25.3: CurrentInclinationToGravity

```
Tracker.InclinationSensor.CurrentInclinationToGravity
```

Represents the current rotation angles InclinationRotX and InclinationRotY from the OTG process.

Listing 25.4: InclinedToGravity

```
Tracker.InclinationSensor.InclinedToGravity
```

If set, CurrentInclinationToGravity is applied on the controller, i.e. the measurements are corrected to gravity. In this case, InclinationRotX and InclinationRotY are applied as orientation parameter on the controller.

25.1.1 Formula

If InclinedToGravity is not set, you have to apply the rotation angles by the following formula:

- T(x) = RTrackerToGravity *x
- T(x): position of target to gravity
- x : position of target to Tracker (measured 3D position)
- RTrackerToGravity: Rotation matrix with rotation angles InclinationRotX and InclinationRotY

$$R = \begin{bmatrix} cz * cy & -sz * cy & sy \\ sz * cx + cz * sy * sx & cz * cx - sz * sy * sx & -cy * sx \\ sz * sx - cz * sy * cx & cz * sx + sz * sy * cx & cy * cx \end{bmatrix}$$

- cx = cos(InclinationRotX)
- sx = sin(InclinationRotX)
- cy = cos(InclinationRotY)
- sy = sin(InclinationRotY)
- cz = 1
- sz = 0

25.2 Inclination Monitoring

The Inclination Monitoring is a process that runs in background to detect whether the Tracker is stable or not. If the Tracker is moved or sinks slowly and exceeds a given Threshold, a warning is thrown to inform the user on the Tracker movement. The delivered InclinationMeasurement is always converted to the Tracker coordinate system, independently of the horizontal position of the Tracker.
Please note that access to inclination data and monitoring is NOT available before performing an OTG for all families excluding ATS600.

Listing 25.5: Active

```
Tracker.InclinationSensor.Monitoring.Active
```

If activated, an inclination sensor stream is started. If disabled, the inclination sensor stream is stopped.

Listing 25.6: Current

```
Tracker.InclinationSensor.Monitoring.Current
```

Delivers the current inclination absolute value in the Tracker coordinate system. The current inclination is updated constantly, depending on the defined Interval. The user has to calculate the drift on his own, by computing the difference to the initial value on startup of the monitoring.

Listing 25.7: Interval

```
Tracker.InclinationSensor.Monitoring.Interval
```

Defines the update interval of the current inclination. Typically, an interval of 15 s to 60 s is reasonable.

Listing 25.8: Threshold

```
Tracker.InclinationSensor.Monitoring.Threshold
```

The Threshold of the inclination sensor monitoring. If the Threshold is exceeded, an InclinationChanged event is fired.

Listing 25.9: ThresholdExceeded

```
Tracker.InclinationSensor.Monitoring.ThresholdExceeded
```

Flagged if the Threshold has been exceeded.

Listing 25.10: WorkingRangeExceeded

```
Tracker.InclinationSensor.Monitoring.WorkingRangeExceeded
```

Flagged if the working range of the inclination sensor has been exceeded.

Listing 25.11: Reset

```
Tracker.InclinationSensor.Monitoring.Reset()
```

Sets the initial inclination to the current inclination

Listing 25.12: InclinationChanged

```
Tracker.InclinationSensor.Monitoring.InclinationChanged
```

Event that is fired frequently when the 'Interval' value is reached and the 'Current' inclination value is delivered.

Listing 25.13: Measure

```
Tracker.InclinationSensor.Monitoring.Measure()
```

Measures the current inclination in the Tracker coordinate system (one single measurement). It is recommended to rather use the nivel stream for monitoring applications.

Chapter 26

Tracker Alignment

26.1 Orientation and Transformation

The orientation takes the instrument coordinate system to the world coordinate system and the transformation takes the world coordinate system to an object coordinate system. The two sets are used either by them selves or together to show coordinates in the required coordinate system. The major input to these calculations are the coordinates of a set of reference points (nominals) together with the corresponding measured coordinates (actuals). The result of the calculation is a seven parameter transformation of the measured points onto the reference points.

26.2 Orientation

Orientation refers to the alignment of a tracker with respect to a world coordinate system (WCS). The world coordinate system is typically defined by the principal measurement station. The coordinates of a point with respect to the principal station are called nominal or reference coordinates. The coordinates as measured by the active station are called actual coordinates (Figure 26.1).

26.2.1 Orientation parameters

The orientation is a six parameter transformation consisting of three translation and three rotation parameters. The scale is fixed to 1 in case of an orientation. The orientation describes the mapping of a given set of actual points (actuals) onto a given set of reference points (nominals). The mapping is calculated such as to minimize the deviation between the transformed points and the corresponding reference points in a least squares sense. In the orientation case the map assumes the form:

- T(x) = t + R*x/s
- T(x): transformed position, nominals
- x : input coordinates, actuals

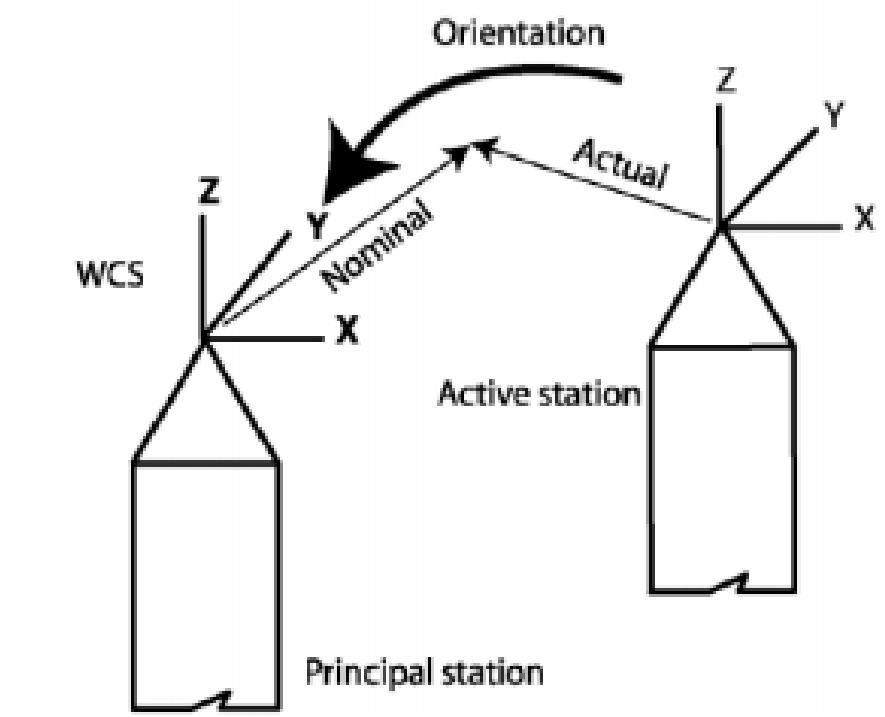


Figure 26.1: Orientation

- t: 3D translation vector
- R: 3x3 Rotation matrix
- s: scale, fixed to 1 for the orientation

$$R = \begin{bmatrix} cz * cy & -sz * cy & sy \\ sz * cx + cz * sy * sx & cz * cx - sz * sy * sx & -cy * sx \\ sz * sx - cz * sy * cx & cz * sx + sz * sy * cx & cy * cx \end{bmatrix}$$

- cx = cos(xAngle)
- sx = sin(xAngle)
- cy = cos(yAngle
- sy = sin(yAngle)
- cz = cos(zAngle)
- sz = sin(zAngle)

The corresponding residuals are:

- residual = T(actual) - nominal

26.3 Transformation

A transformation defines a local object coordinate system. In this case the object coordinates play the role of nominals (Figure 23.2). Activating the calculated transformation parameters and remeasurement yields actual coordinates approximately equal to the nominal coordinates .

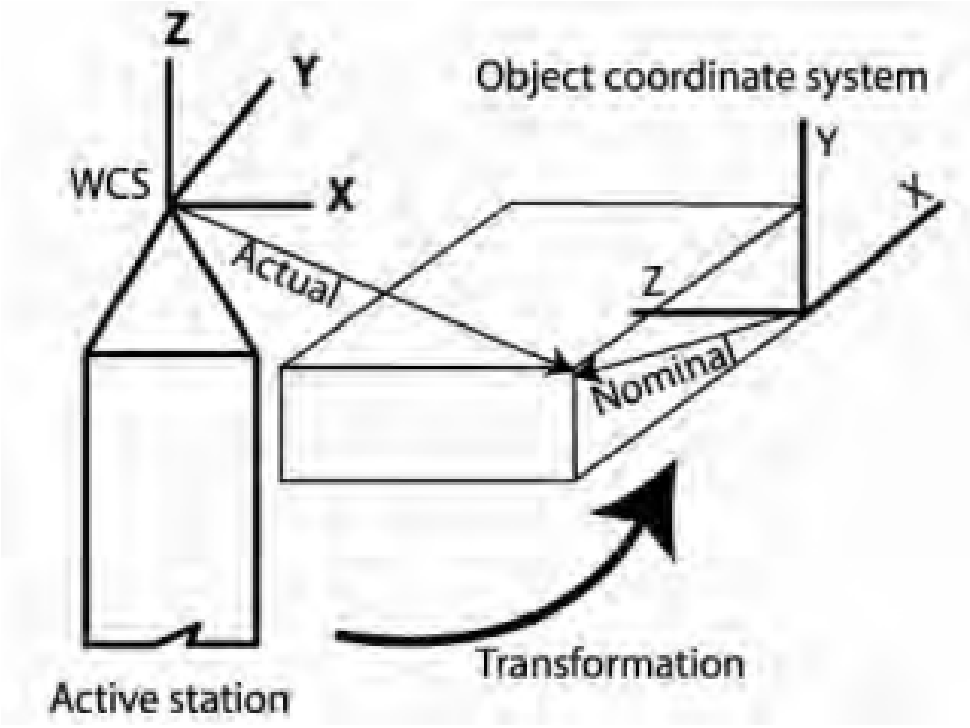


Figure 26.2: Transformation

26.3.1 Transformation parameters

The transformation is a seven parameter transformations consisting of three translation, three rotation and one scale parameter. Typically, the scale is close to one, e.g. when describing a temperature dependent dilation on the object. The transformation is the inverse equation of the orientation. Thus, for the transformation the map assumes the form:

- T(x) = s*R-1*(x-t)
- T(x): transformed position, nominals
- x : input coordinates, actuals
- t: 3D translation vector
- R-1: inverse 3x3 Rotation matrix
- s: scale

26.4 Applications

A common use case is a 'move station'. In this case the transformation describes the relation between the reference station (WCS) and the object coordinate system. The orientation in turn describes the relation between the current station and the reference station (WCS). The tracker is then moved around the object (e.g. an airplane) and the orientation has to be computed for each move station. When applying both orientation and transformation sets to the tracker, the measured coordinates of the current station yields object coordinates.

26.5 Code samples, document usage

ATTENTION: In contrast to all other classes on the tracker object tree which use the units as set in the Tracker.Settings.Units object, all TrackerAlignment units and coordinate systems are fixed to the Leica base units as follows:

- Coordinate type = Cartesian
- Rotation type = RotationAngles
- Angle unit = Radian
- Length unit = Meter

Please be aware that all properties on the TrackerAlignment objects use these base units and can not be changed.

The orientation and transformation can be computed and applied separately or together. Instead of passing the actuals as doubles in AddPoint(), the method AddPointFromMeasurement() can be used which accepts a StationaryMeasurement3D for the actuals.

- oi.Points.AddPoint(0, 0, 0, 0, 0, 0, 1, 1, 1, 1E-5, 1E-5, 1E-5)
- var m = t.Measurement.MeasureStationary();
- oi.Points.AddPointFromMeasurement(0, 0, 0, 0, 0, 0, m as StationaryMeasurement3D);

Constraints can be added as well: For example if the scale shall be fixed in the transformation, use the method:

- ti.SetScale(1.0, ti.FixedAccuracy)

which fixes the scale to 1. In order to calculate the orientation and transformation use the methods:

- var orientationParameters = t.TrackerAlignment.CalculateOrientation(oi);
- var transformationParameters = t.TrackerAlignment.CalculateTransformation(ti);

respectively. In order to apply the calculated orientation and transformation on the tracker use the methods:

- t.Settings.SetCalculatedOrientation(orientationParameters)
- t.Settings.SetCalculatedTransformation(transformationParameters)

respectively. The usage of the orientation and transformation is documented below:

Listing 26.1: Tracker Alignment, orientation, transformation

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LMF.Tracker;
using LMF.Tracker.MeasurementResults;
using LMF.Tracker.TrackerAlignments;

namespace SDK.Samples.CodeSamples
{
    public class TrackerAlignment
    {
        // The user can use any positive value to keep a certain parameter more or less fixed, i.e.
        //
        // accuracy = 0 means that the user wants to fix the parameter exactly to the provided value
        // accuracy = 1mm means that the provided value is known with an uncertainty of +/-1mm in
        //            the sense of a standard deviation
        //
        // In the calculation of the orientation or transformation the square inverse of the accuracy
        // is used as a weight for an additional equation parameter = provided value.
        //
        // Similarly any positive value can be used to indicate the accuracy of the provided nominal
        // and measured coordinates in the "AddPoint" method. The square inverse of these accuracies
        // are used as weights during the adjustment.

        public void DocumentUsageOrientation()
        {
            //connect to real Tracker or Simulator
            using (Tracker t = new Connection().Connect("AT960Simulator"))
            {
                var oi = new AlignmentOrientationInput();

                //add nominals and actuals, must be cartesian coordinates in [m]
                double n = oi.FixedAccuracy; //a priori accuracy nominals
                double a = 1E-5; //a priori accuracy actuals
                oi.Points.AddPoint(0, 0, 0, n, n, n, 1, 1, 1, a, a, a);
                oi.Points.AddPoint(0, 1, 0, n, n, n, 1, 2, 1, a, a, a);
                oi.Points.AddPoint(1, 0, 0, n, n, n, 2, 1, 1, a, a, a);

                //add constraints, if any
                //oi.SetRotationX(0.0, oi.FixedAccuracy);
                //oi.SetRotationY(0.0, oi.FixedAccuracy);
                //oi.SetRotationZ(0.0, oi.FixedAccuracy);

                //compute orientation and get AlignmentOrientationResult
                var oParams = t.TrackerAlignment.CalculateOrientation(oi);

                //apply the calculated transformation on the Tracker object
                t.Settings.SetCalculatedOrientation(oParams);
            }
        }


        public void DocumentUsageTransformation()
        {
            //connect to real Tracker or Simulator
            using (Tracker t = new Connection().Connect("AT960Simulator"))
            {
                var ti = new AlignmentTransformationInput();

                //add nominals and actuals, must be cartesian coordinates in [m]
                double n = ti.FixedAccuracy; //a priori accuracy nominals
                double a = 1E-5; //a priori accuracy actuals
                ti.Points.AddPoint(0, 0, 0, n, n, n, 1, 1, 1, a, a, a);
                ti.Points.AddPoint(0, 1, 0, n, n, n, 1, 2, 1, a, a, a);
```

```
                ti.Points.AddPoint(1, 0, 0, n, n, n, 2, 1, 1, a, a, a);

                //add constraints, if any
                ti.SetScale(1.0, ti.FixedAccuracy);

                //compute transformation and get AlignmentTransformationResult
                var tParams = t.TrackerAlignment.CalculateTransformation(ti);

                //apply the calculated transformation on the Tracker object
                t.Settings.SetCalculatedTransformation(tParams);
            }

        }


        public void DocumentUsageAddPointFromMeasurement()
        {
            //connect to real Tracker
            using (Tracker t = new Connection().Connect("192.168.0.1"))
            {
                var oi = new AlignmentOrientationInput();

                //add actuals from measurement
                double n = oi.FixedAccuracy; //a priori accuracy nominals
                var m = t.Measurement.MeasureStationary() as StationaryMeasurement3D;
                oi.Points.AddPointFromMeasurement(0, 0, 0, n, n, n, m);

                m = t.Measurement.MeasureStationary() as StationaryMeasurement3D;
                oi.Points.AddPointFromMeasurement(0, 1, 0, n, n, n, m);

                m = t.Measurement.MeasureStationary() as StationaryMeasurement3D;
                oi.Points.AddPointFromMeasurement(1, 0, 0, n, n, n, m);

                //calculate and apply
                var oParams = t.TrackerAlignment.CalculateOrientation(oi);
                t.Settings.SetCalculatedOrientation(oParams);
            }
        }
    }
}
```

26.6 AT9x0 Tracker Coordinate Systems

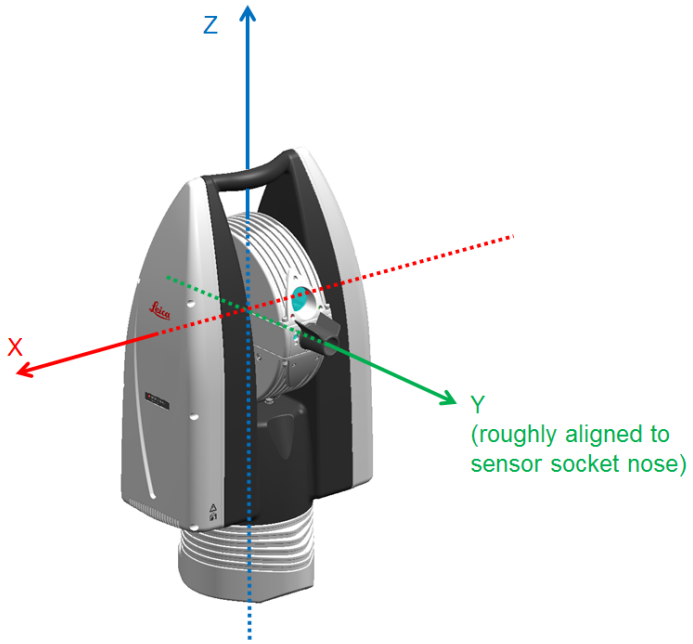


Figure 26.3: Cartesian Coordinate System

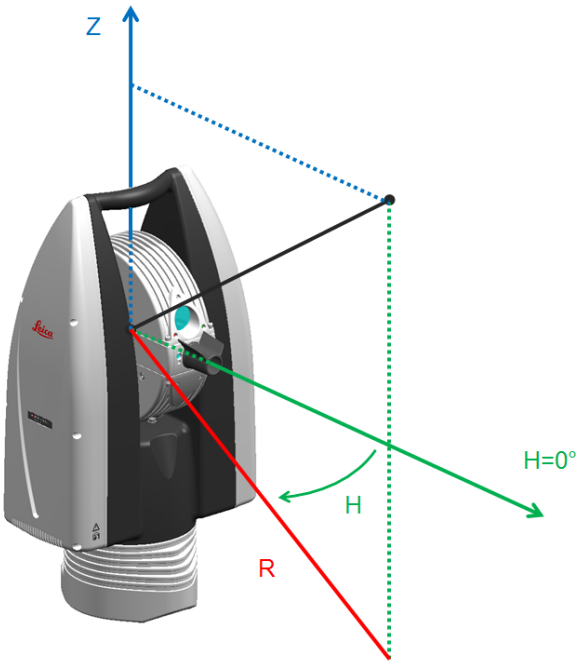


Figure 26.4: Cylindrical Coordinate System

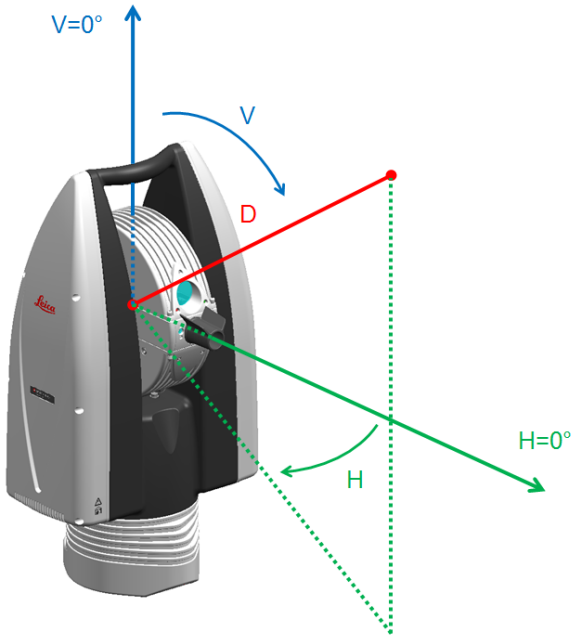


Figure 26.5: Spherical Coordinate System

26.7 AT40x Tracker Coordinate Systems

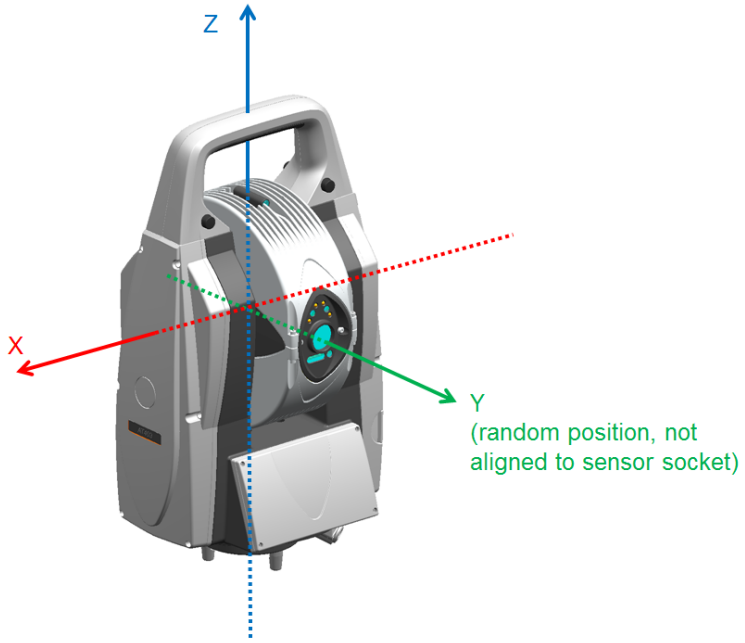


Figure 26.6: Cartesian Coordinate System

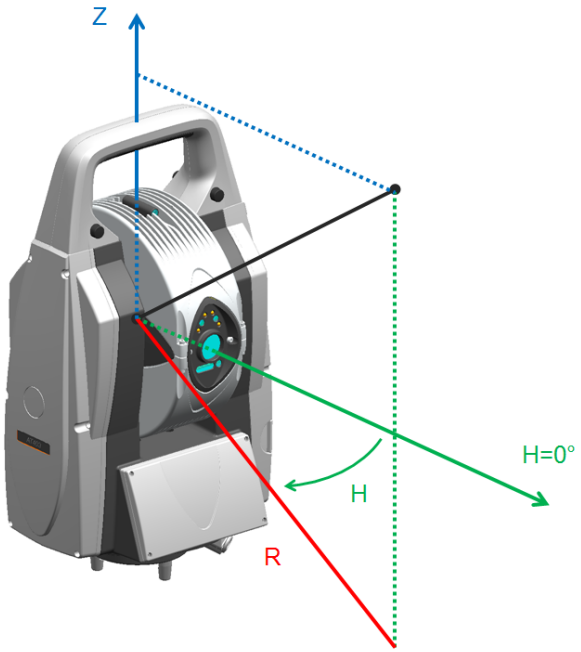


Figure 26.7: Cylindrical Coordinate System

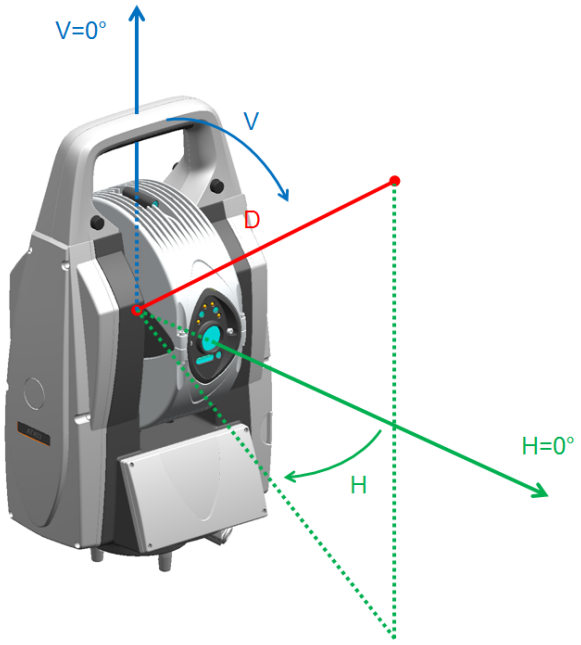


Figure 26.8: Spherical Coordinate System

Chapter 27

Scanning

The main classes for scanning use cases are the **AreaScanProfile**, **LineScanProfile** and **RingScanProfile** class. They hold the needed scanning mode parameters and also the scanning area defined by regions and lines. The next class diagram gives an overview of the scanning profiles sub tree and its possibilities.
In the **AreaScanProfile** is supported only raster scan, where the PointToPointDistance and LineToLineDistance have the same value. The LineToLineDistance value is ignored, and only PointToPointDistance is used.

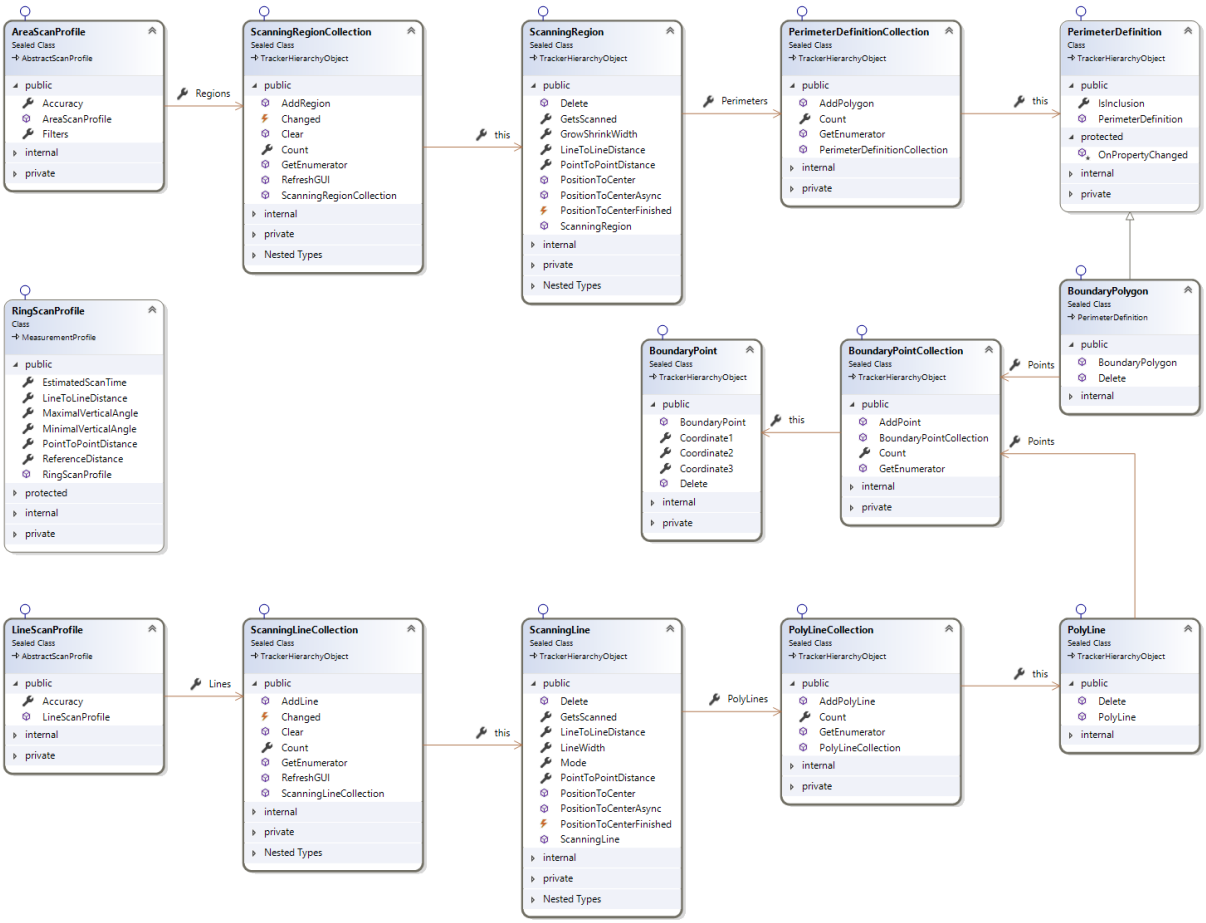


Figure 27.1: AbstractScanProfile sub tree classes

27.1 Defining A Scan Region And Scanning With The Tracker

The following code snippets will show you an example of how to use the AreaScanProfile to define a region by creating a perimeter consisting of polygons defining the outer and inner boundary of the scanning area.

27.1.1 Overview

Listing 27.1: Overview

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void Demo()
        {
            Connect();
            OpenOvcScanningDialog();
            SelectProfileAndAccuracy();
            SelectScanningTarget();
            CreateScanningRegion();
            UpdateOvcScanningDialog();
            Scan();
        }
    }
}
```

27.1.2 Connecting to the tracker

Listing 27.2: Connect to the tracker

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.Tracker;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void Connect()
        {
            // _Tracker = new Connection().Connect("192.168.0.1");
            _Tracker = new Connection().Connect("ats600simulator");
        }
    }
```

```
}
```

27.1.3 Open scanning OVC dialog

Listing 27.3: Open scanning OVC dialog

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void OpenOvcScanningDialog()
        {
            // Starting the OVC dialog
            _Tracker.OverviewCamera.Dialog.Show();
        }
    }
}
```

27.1.4 Selecting the AreaScanProfile and configuring it

Listing 27.4: Selecting the AreaScanProfile

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.Tracker.Enums;
using LMF.Tracker.Measurements.Profiles;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void SelectProfileAndAccuracy()
        {
            // Getting the AreaScanProfile
            var asp = _Tracker.Measurement.Profiles.OfType<AreaScanProfile>().SingleOrDefault();

            if (asp != null)
            {
                // Choosing the accuracy
                asp.Accuracy.Value = EAccuracy.Precise;

                // Do not forget to select the profile as active one!
                asp.Select();
            }
        }
    }
}
```

27.1.5 Selecting the right target

Listing 27.5: Selecting the right target

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.Tracker.Targets.ScanTargets;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void SelectScanningTarget()
        {
            // Getting the scan target of type "Surface"
            var surface = _Tracker.Targets.OfType<Surface>().SingleOrDefault();

            if (surface != null)
            {
                // and select it to be the active one
                surface.Select();
            }
        }
    }
}
```

27.1.6 Creating the desired scan region

LMF supports the definition of one ore more scan regions. Each scan region can hold one or more polygons to be able to define complex scan regions. These polygons are than merged into one polygon depending on their position in the z-order (starting at index 0 in the collection as bottom polygon). The **IsInclusion** flag defines the way the polygons are merged as union (IsInclusion = true) or as subtraction (IsInclusion = false) respectively.

Adding points to a polygon uses the coordinate system and physical units from the **Tracker.Settings** object as well as alignment (transformation/rotation) if any is set. So a good advice is to first select the proper coordinate system and physical units before any points are added. Changing the systems and units on the **Settings** object is allowed, but be aware that new points are using the then changed coordinate system and units. Older points **DO NOT** change its system after they have been added. So creating polygons in different systems can be done but will not change already existing points.

Listing 27.6: Creating the desired scan region

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;
using LMF.Tracker.Measurements.Perimeters;
using LMF.Tracker.Measurements.Profiles;
using LMF.Units;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void CreateScanningRegion()
        {
            // Setting the desired coordinate system and units for defining the scan region
            // !!! Be aware that changing setting creates the added points in the chosen physical units
            // and coordinate systems. Also optional alignments (orientations and transformations)
            // will be taken into account if they have been definde previousely on the -Settings- object.
            // Creating polygons in different systems can be done but will not change already existing
            // points. !!!
            _Tracker.Settings.CoordinateType = ECoordinateType.Spherical;
            _Tracker.Settings.Units.AngleUnit = EAngleUnit.Radian;
            _Tracker.Settings.Units.LengthUnit = ELengthUnit.Meter;

            // Optionally: To set alignment parameters, please uncomment the following lines
            //_Tracker.Settings.SetOrientation(...);
            //_Tracker.Settings.SetTransformation(...);

            // Getting the area scan profile
            var asp = _Tracker.Measurement.Profiles.OfType<AreaScanProfile>().SingleOrDefault();

            if (asp != null)
            {
                // Clear all existing scan regions
                asp.Regions.Clear();

                // Add a new region and define the distances
                var hexaRegion = asp.Regions.AddRegion();
                hexaRegion.LineToLineDistance.Value = 0.01;
                hexaRegion.PointToPointDistance.Value = 0.005;

                // Now get the actual direction the laser is pointing at ...
                var direction = _Tracker.GetDirection();

                // ... and define a scan regions around that point
                // by creating an outer hexagonal polygon
                double rRegion = 4.0 / 180.0 * Math.PI;
                var hexaPoly = hexaRegion.Perimeters.AddPolygon();
                CreateHexagonPointsInPolygon(
                    hexaPoly, rRegion,
                    direction.HorizontalAngle.Value,
                    direction.VerticalAngle.Value,
                    5.0);

                // with an inner hexagonal polygon as hole
                double rHole = 2.5 / 180.0 * Math.PI;
                var hexaHole = hexaRegion.Perimeters.AddPolygon();
                CreateHexagonPointsInPolygon(
                    hexaHole, rHole,
                    direction.HorizontalAngle.Value,
                    direction.VerticalAngle.Value,
                    5.0);
                hexaHole.IsInclusion = false; // this defines the hole
            }
        }

        // This is the helper function to create the hexagonal polygon
        private void CreateHexagonPointsInPolygon(BoundaryPolygon poly, double radius, double offsetH,↩
            double offsetV, double distance)
        {
```

```
            for (double angle = 0.0; angle < Math.PI * 2.0; angle += Math.PI * 2.0 / 6.0)
            {
                poly.Points.AddPoint(
                    offsetH + Math.Cos(angle) * radius,
                    offsetV + Math.Sin(angle) * radius,
                    distance);
            }
        }
    }
}
```

27.1.7 Visually check region in OVC

Listing 27.7: Visually check region in OVC

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.Tracker.Measurements.Profiles;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void UpdateOvcScanningDialog()
        {
            // Getting the area scan profile
            var asp = _Tracker.Measurement.Profiles.OfType<AreaScanProfile>().SingleOrDefault();

            // Updating the GUI in OVC dialog to show the added regions
            asp.Regions.RefreshGUI();
        }
    }
}
```

27.1.8 Starting the Scan

Listing 27.8: Starting the Scan

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.Tracker.ErrorHandling;
using LMF.Tracker.MeasurementResults;
using LMF.Tracker.Measurements;

namespace SDK.Samples.CodeSamples.Scanning
{
    partial class ScanningDemo
    {
        public void Scan()
        {
            // Registering the event handler for incoming scan data
```

```
            _Tracker.Measurement.MeasurementArrived += Measurement_MeasurementArrived;

            //  Starting the scan process
            _Tracker.Measurement.StartMeasurement();
        }

        private void Measurement_MeasurementArrived(
            MeasurementSettings sender,
            MeasurementCollection paramMeasurements,
            LmfException paramException)
        {
            foreach (var m in paramMeasurements)
            {
                //  Do something with the measurements
            }
        }
    }
}
```

Chapter 28

Command Execution Time

Most of the synchronous LMF commands return quite fast. But there are some (positioning, measuring) that take longer. This chapter describes the times for the long running commands of LMF. Using these times an application using LMF can set the required timeouts to handle the case correctly where something is going wrong and getting stuck.
The asynchronous variants of the LMF commands return immediately and send an event in case they are finished. So waiting for these events may use the same timeouts as calling the synchronous commands.

28.1 Tracker Initialization

The initialization of an AT40x may take up to 80s depending on the stability of the environment. The initialization of an AT930 / AT960 may take up to 30s.

28.2 Positioning and Target Search

The commands:

- PositionTo
- PositionToTarget

are also long running commands. For PositionTo command this only holds if the flag to search the target is set. Otherwise the tracker is just positioned as fast as possible. Typically this positioning time without the target search is around 1s. In case the target needs to be detected the positioning commands respect the timeout that is set in Tracker.TargetSearch.Timeout. This also holds for the command to search the target without doing a new positioning. So the application may set its own timeout accordingly. In the asynchronous case the command is terminated in case the SelectedChanged event arrives. But in case no target may be found this event does not arrive. In case the tracker was locked onto a target before calling the positioning command the SelectedChanged event arrives twice: Once for leaving the previous target and once for locking onto the new target.

The command to change the face may be considered as a positioning command. In case it leaves a target that is locked on it tries to lock on again. Otherwise it just changes the face. So the Target Search timeout needs to be respected in the first case only.

28.3 Measurements

The measurement times are different for the different tracker types. So the timeout for a synchronous MeasureStationary or GoAndMeasureStationary command need to be set accordingly.
For the asynchronous variants the measurement command is finished at the moment where the MeasurementArrived event arrives.
For the AT930 / AT960 we recommend the following timeouts based on the measurement profile:

- StationaryMeasurementProfile: Using this profile the timeout is dependent on the selected accuracy of the measurement:

  Fast 2s

  Standard: 12s (used for the maximum distance)

  Precise: 12s

- Other profiles: 7s

For the two face measurements the measurement timeout is to be doubled plus 2s additional time to change the face.
For the command GoAndMeasureStationary using AT930 / AT960 the following conditions need to be considered:

- This command combines the positioning and measurement. So the positioning time either using Power Lock or Spiral Search needs to be included:

  Power Lock: For the Power Lock 3s should be added to the profile dependent measurement timeout to be on the safe side.

  Spiral Search: The time the user sets for the Spiral Search timeout is to be added to the profile dependent measurement timeout plus 1s overhead.

- In case no stationary measurement profile is selected (CustomStationary is ok) this command always uses the StationaryMeasurementProfile and the standard accuracy as fallback so that it does not fail if the wrong profile is selected. The timeout needs to be set accordingly.

For the AT403 we recommend the following timeouts based on the measurement profile:

- Fast: 4s
- Standard: 6s
- Precise: 16s

For the AT402 / AT401 we recommend the following timeouts based on the measurement profile:

- Fast: 4s
- Standard: 8s
- Precise: 16s

For the command GoAndMeasureStationary using AT40x the timeout for the Spiral Search needs to be added to the measurement timeout plus 1s overhead.
For the two face measurements the timeout needs to be doubled plus 2s additional time to change the face.

28.4 L-File Generation

The execution time of the GenerateLFile command is highly dependent on the speed of the network connection to the tracker. It may take up to several minutes. In case the L-File is generated when no tracker is connected only the local log files of the LMF on the PC are included. In that case the time is typically below one minute depending on how many commands LMF already executed.

Chapter 29

Time Server

LMF offers the possibility to connect an AT9x0 tracker to an external time server using the Precision Time Protocol (PTP). Therefore, an additional time stamp property **TimeStampExternal** is available in the Tracker.MeasurementResults.Measurement object:
If PTP is disabled or the connection to the time server is lost, the property **TimeStampExternal** is set to the minimum value by default (Monday, 01. January 0001 00:00:00.000000). Otherwise, the object **TimeStampExternal** contains the time stamp received from the external time server.
The required configuration can be made in the **Tracker.TimeSync.TimeServer** object. There is a property **Mode** which delivers a **ETimeServerMode** with two possible values:

- ETimeServerMode.Off
- ETimeServerMode.PTP

If Mode is set to **ETimeServerMode.PTP**, you can access the **PTP** property to configure Precision Time Protocol Settings. You have the following setting options:

- Master IP Address
- Domain
- Announce Interval
- Sync Interval
- Delay Request Interval
- Announce Receipt Timeout

In the property **Status**, you see the current status of the PTP connection. The status delivers a **EPrecisionTimeProtocolStatus** with following possible values:

- EPrecisionTimeProtocolStatus.NotInitialised: Happens only on startup.
- EPrecisionTimeProtocolStatus.AllOk: All good, ready to go.
- EPrecisionTimeProtocolStatus.NoMasterCommunication: The Slave is working but does not receive master clock messages.

- EPrecisionTimeProtocolStatus.FirmwareError: The Slave has a problem. Most likely, the Tracker needs to be rebooted.
- EPrecisionTimeProtocolStatus.FilterError: The data processing had an error. Non-fatal, will recover.
- EPrecisionTimeProtocolStatus.NoisyData: The time sync data is rejected due to high noise.

You can also subscribe the **Changed** event of the **Status** property to get notified when the status changes.

Chapter 30

Connect Box

The LMF offers the information if the AT960 tracker is connected with a connect box or not. If a connect box is connected to the tracker, information like name, ip address, serial number and installed firmware of the connect box are available. The **IsConnected.Changed** event can be registered to get the information when a connect box has been connected or disconnected from the tracker.

ConnectBox object in AT960 tracker

The following code sample shows ConnectBox object and properties of it

Listing 30.1: Getting information if connect box is connected to AT960 tracker

```
using LMF.Tracker;
using LMF.Tracker.BasicTypes.BoolValue;

namespace SDK.Samples.CodeSamples
{
  class TrackerConnectBoxObject
  {
        private AT960Tracker at960;
        private bool isConnectBoxConnected;
        private string connectBoxIpAddress;
        private string connectBoxName;
        private string connectBoxSerialNumber;
        private string connectBoxInstalledFirmware;
        public void GetConnectBoxInformation()
        {
            var tracker = new Connection().Connect("192.168.0.1");

            //Check if we are connected to a At960Tracker
            if (tracker is AT960Tracker)
            {
                //Cast the tracker object into the correct Subclass
                at960 = (AT960Tracker)tracker;

                at960.ConnectBox.IsConnected.Changed += ConnectBoxIsConnectedChanged;
                //update current connect box infromation
                ConnectBoxIsConnectedChanged(at960.ConnectBox.IsConnected, at960.ConnectBox.↩
                    IsConnected.Value);
            }
        }

        //event to handle connect box is connecte value
```

```
    private void ConnectBoxIsConnectedChanged(ReadOnlyBoolValue sender, bool paramNewValue)
    {
            //get is connected value
            isConnectBoxConnected = paramNewValue;
            // if ConnectBox is connected
            if (isConnectBoxConnected)
      {
                //get information about connected connect box
                connectBoxIpAddress = at960.ConnectBox.IPAddress.Value;
                connectBoxName = at960.ConnectBox.Name.Value;
                connectBoxSerialNumber = at960.ConnectBox.SerialNumber.Value;
                connectBoxInstalledFirmware = at960.ConnectBox.InstalledFirmware.Value;
      }
      else
      {
                //if no connect box connect reset properties
                connectBoxIpAddress = string.Empty;
                connectBoxName = string.Empty;
                connectBoxSerialNumber = string.Empty;
                connectBoxInstalledFirmware = string.Empty;
            }
    }
  }
}
```

Part IV

Working with the connect box

Chapter 31

Configuring Network

The ethernet interface can be configured either by static ip or dynamically through DHCP. To connect the connect box to an pre-existing wired network the supplied straight RJ45 cable should be used. To connect directly to the connect box without a switch/hub you can use the supplied crossed-over RJ45 network cable.
Alternatively, a wireless connection over an external AP can be established. Please consult Tracker Pilot documentation for further information.

Chapter 32

Connecting to a Connect Box

To connect to a connect box can be done in two different ways. Either by connecting to an previously known IP address directly or by letting LMF search for available connect boxes on your network and selecting one of them to connect afterwards.

32.1 Connecting via 'Connection' class

The following code example shows how to directly connect to a connect box from which the IP address is known. First instantiate an object of type **IOConnection** and use its only method **Connect** with the IP address of the connect box and the bool argument if you want to connect in read only mode or not. If you want to be able to set the RobotGo digital output the second bool argument has to be false (default). In this case the connect box will acquire the exclusive connection to this digital output. Only one client is allowed to controll the RobotGo output bit at a time. Every other connection can only connect if the connection is started in read only mode (second argument set to true). If no exception gets thrown (i.e. ConnectBox not connected, etc.) the resulting object is the tracker object and ready to work with.

Listing 32.1: Connecting via 'IOConnection' class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.IO;
using LMF.Tracker.ErrorHandling;


namespace SDK.Samples.CodeSamples
{
  class ConnectBoxConnectingWithConnection
  {
    public void DirectConnectionWithIpAddressWithRobotGoControl()
       {
     try
     {
       var connection = new IOConnection();
       var connectBox = connection.Connect("192.168.0.2");
```

```
      }
      catch(LmfException ex)
            {
                Console.WriteLine("Could not connect to connect box => " + ex.Description);
            }
        }
         public void DirectConnectionWithIpAddressInReadOnlyMode()
        {
          try
          {
            var connection = new IOConnection();
            var connectBox = connection.Connect("192.168.0.2", true);
          }
          catch (LmfException ex)
          {
            Console.WriteLine("Could not connect to connect box => " + ex.Description);
          }

        }
  }
}
```

32.2 Connecting via 'ConnectBoxFinder' class

The second code example shows a more convenient way of connecting to a connect box. It first uses an object of type **ConnectBoxFinder** to get a list of all available connect boxes in the network and you can then choose one to connect to it without directly knowing its IP address. The list of found **TConnectBoxInfo** objects, which can be reached by the property **ConnectBoxes**, contain some information about the found connect boxes (like Name, IPAddress, etc.).

Listing 32.2: Connecting via 'ConnectBoxFinder' class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using LMF.IO;
using LMF.Tracker.ErrorHandling;


namespace SDK.Samples.CodeSamples
{
  class ConnectBoxConnectingWithConnectBoxFinder
  {
      public void GettingAllConnectBoxesInNetwork()
      {
        // Instantiate the TrackerFinder object
        var found = new ConnectBoxFinder();

        // Check if any tracker have been found in the network
        if (found.ConnectBoxes.Count > 0)
        {
          // Take the first one and connect
          var connectBoxInfo = found.ConnectBoxes[0];
          try
          {
            var connection = new IOConnection();
```

```
            var connectBox = connection.Connect(connectBoxInfo.IPAddress);
          }
          catch (LmfException ex)
          {
            Console.WriteLine("Could not connect to connect box => " + ex.Description);
          }
        }
      }
    }
  }
}
```

This method can be used to simple iterate over available trackers and i.e. show them in a specific GUI to let the customer choose a tracker without manually typing in the IP address of the tracker.

Chapter 33

Connect Box

33.1 ConnectBox Object

The ConnectBox object is the starting point of all operations that can be done with LMF when connecting to a CB21. Most of the properties are located on the ConnectBox class itself. There is a CB21ConnectBox class derived from ConnectBox but at the moment it is the only one and has no specific features.

33.1.1 Class Hierarchy

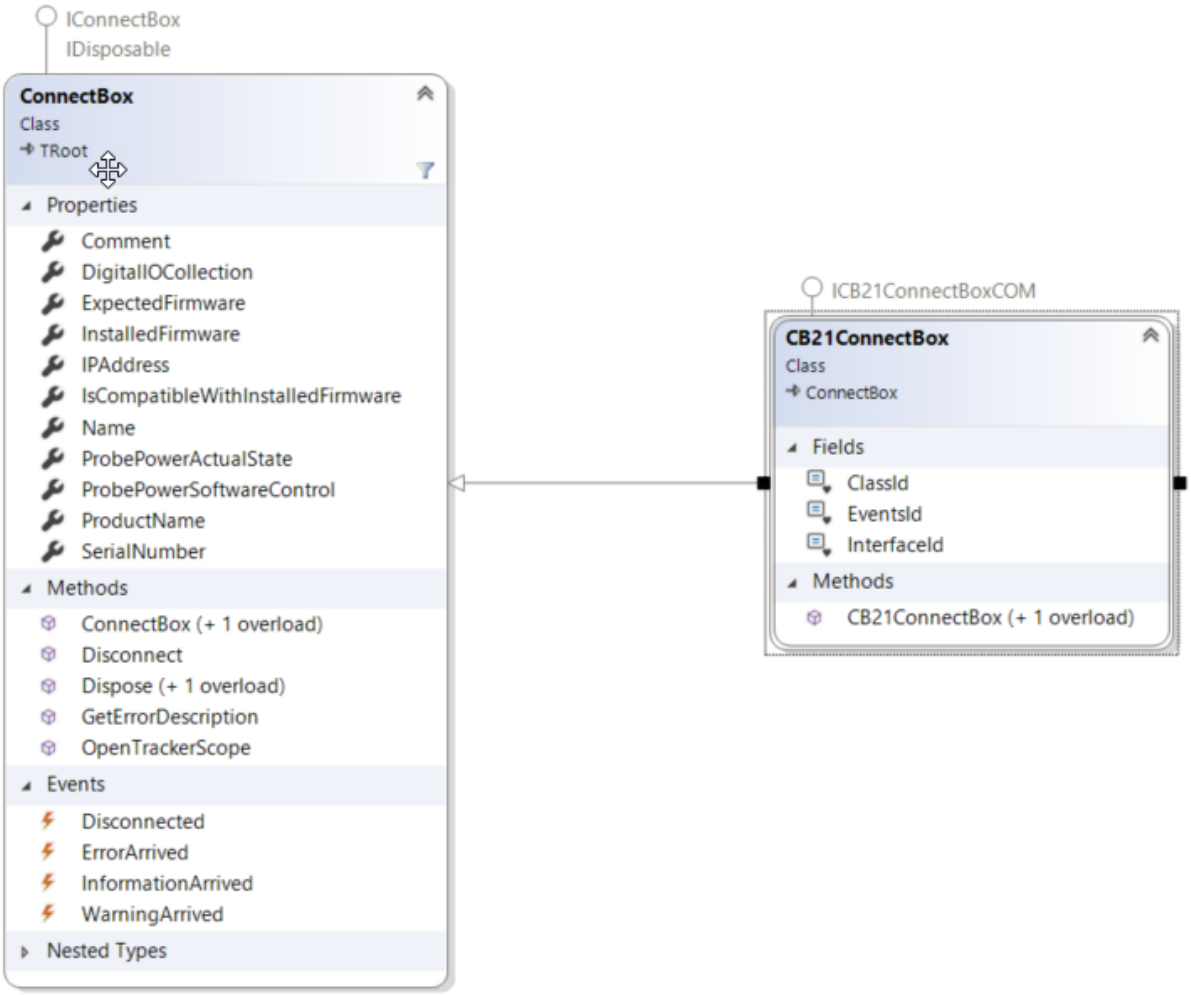


Figure 33.1: ConnectBox class hierarchy

Chapter 34

DigitalIOCollection

34.1 Overview

To be able to get the state of the Digital IO's there is a collection with objects for every Digital Input and Output available on the Connect Box.

34.2 DigitalIO

The Base class is DigitalIO and every derived class represents one digital Input or Output available in the hardware.

34.2.1 Class Hierarchy

Figure 34.1: DigitalIO class hierarchy

There is a **Changed** event triggered on every available DigitalIO when the value of the input/output pin is changed.

How to get the digital IO's

The following code sample shows how to get the digital IO's and register to the Changed event

Listing 34.1: How to get the DigitalIO and register to event

```
using LMF.IO;
using LMF.IO.DigitalIOs.DigitalIO;
using LMF.Tracker.BasicTypes.BoolValue;
using LMF.Tracker.ErrorHandling;
using System;

namespace SDK.Samples.CodeSamples
```

```
{
  class ConnectBoxDigitalIOCollection
  {
        public void DirectConnectionWithIpAddressInReadOnlyMode()
        {
            try
            {
                var connection = new IOConnection();
                var connectBox = connection.Connect("192.168.0.2", true);

                foreach(DigitalIO digitalIO in connectBox.DigitalIOCollection)
        {
                    if (digitalIO is ClientStatusBitReadOnly)
                        (digitalIO as ClientStatusBitReadOnly).Changed += ClientStatusBitChanged;
                    if (digitalIO is ClientWaitBitReadOnly)
                        (digitalIO as ClientWaitBitReadOnly).Changed += ClientWaitBitChanged;
                    // continue with every derived class
        }
            }
            catch (LmfException ex)
            {
                Console.WriteLine("Could not connect to connect box => " + ex.Description);
            }

        }

        //event to handle changed client wait bit output
        private void ClientWaitBitChanged(ReadOnlyBoolValue sender, bool paramNewValue)
    {
            //do something with value
        }

        //event to handle changed client status bit output
        private void ClientStatusBitChanged(ReadOnlyBoolValue sender, bool paramNewValue)
    {
            //do something with value
        }
    }
}
```

34.2.2 RobotGo output

RobotGo output can be changed and is used to send the go signal to the robot. Only one client is allowed to controll this DigitalOutput at a time, that is the reason why when connecting to the ConnectBox there is the option to specify if connection should be in read only mode (cannot set RobotGo output). If connection to ConnectBox with acquring the RobotGo output is successful then the signal can be triggered by changing the **Value** property of the object.

Using RobotGo output

The following code sample shows how to get RobotGo output and use it

Listing 34.2: How to get RobotGo and change it

```
using LMF.IO;
using LMF.IO.DigitalIOs.DigitalIO;
using LMF.Tracker.ErrorHandling;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SDK.Samples.CodeSamples
{
  class ConnectBoxRobotGo
  {
    private RobotGo robotGoOutput;
    private RobotProgramActiveReadOnly robotProgramActive;
    private RobotPositionReachedReadOnly robotPositionReached;
    public void DirectConnectionWithIpAddressWithRobotGoControl()
    {
      try
      {
        var connection = new IOConnection();
        var connectBox = connection.Connect("192.168.0.2");

        //get robot go object
        robotGoOutput = connectBox.DigitalIOCollection.OfType<RobotGo>().FirstOrDefault();
        //get robot program active input to see it robot is ready
        robotProgramActive = connectBox.DigitalIOCollection.OfType<RobotProgramActiveReadOnly>()↩
            .FirstOrDefault();
        //get robot position reached to know when robot movement is done
        robotPositionReached = connectBox.DigitalIOCollection.OfType<↩
            RobotPositionReachedReadOnly>().FirstOrDefault();

        //check if robot program is active
        if(robotProgramActive.Value)
        {
          //set the robot go signal to true
          robotGoOutput.Value = true;
          //wait until robot has reached position
          while(!robotPositionReached.Value) { Task.Delay(1000); }
          //set the robot go signal to false
          robotGoOutput.Value = false;
        }
      }
      catch (LmfException ex)
      {
        Console.WriteLine("Could not connect to connect box => " + ex.Description);
      }
    }
  }
}
```

Chapter 35

Probe Power

The probe power can be controlled by using the **ProbePowerSoftwareControl.Value** property. By setting this property to **false** the probe is shut down. Setting it to **true** should turn the probe on again. However the power to the probe can also be controlled by the robot through the **ProbeShutdownReadOnly** digital input. Because of this the probe is powered only if **ProbePowerSoftwareControl.Value** is set to **true** and the robot did not activate the **ProbeShutdown** input. To have the actual state of the probe power there is the read only bool property **ProbePowerActualState.Value**. This property is showing if the probe is powered or not.

How to use ProbePowerSoftwareControl

The following code sample shows how to use ProbePowerSoftwareControl

Listing 35.1: How to use ProbePowerSoftwareControl

```
using LMF.IO;
using LMF.Tracker.ErrorHandling;
using System;

namespace SDK.Samples.CodeSamples
{
  class ConnectBoxProbePowerSoftwareControl
  {
    public void DirectConnectionWithIpAddressWithRobotGoControl()
    {
      try
      {
        var connection = new IOConnection();
        var connectBox = connection.Connect("192.168.0.2");

        //getting information if probe is powered or not
        var probePowerActualState = connectBox.ProbePowerActualState.Value;

        if(probePowerActualState)
        {
          //turnning probe power off
          connectBox.ProbePowerSoftwareControl.Value = false;

          //turnning probe power on
          connectBox.ProbePowerSoftwareControl.Value = true;
        }
      }
```

```
      catch (LmfException ex)
      {
        Console.WriteLine("Could not connect to connect box => " + ex.Description);
      }
    }
  }
}
```