emScon – Tracker Programming Interface

User Manual

Leica Geosystems

User Manual

emScon TPI

Metrology Division

Preface

These are original instructions and part of the product. Keep for future reference and pass on to subsequent holder/user of product. Read instructions before setting-up and operating the hard- and software. The emScon TPI reference manual and the emScon TPI user manual should always be used together.

Feedback

Your feedback is important as we strive to improve the quality of our documentation. We request you to make specific comments as to where you envisage scope for improvement. Please use the following E-Mail address to send in suggestions:

documentation.metrology@leica-geosystems.com

Software and version	emScon TPI;1.5
Manual release	June 2003
Manual order number	None

Preface

Contact

Leica Geosystems AG
Metrology Division
Moenchmattweg 5
5035 Unterentfelden
Switzerland

Phone ++41 +62 737 67 67

Fax ++41 +62 737 68 34

www.leica-geosystems.com/ims/index.htm

Contents

1.Introduction

Prerequisites

Hardware

The tracker-programming interface (TPI) supports the following laser trackers:

- LT300
- LT500 & LTD500
- LT600 & LTD600
- LTD700
- LT800 & LTD800

Programming Environment

This manual (notation, samples/tutorials) is based on Microsoft Visual Studio 6.0 (VC++ 6.0, Visual Basic 6.0) running on Microsoft Windows (98/NT/2000).

For Unicode applications, install VC++ with Unicode libraries (custom installation). Linker/runtime errors, such as: *mfc42u.lib, mfc42ud.lib or mfc42u.dll* missing, indicate that VC++ was installed without Unicode support.

TCP/IP Communication

TCP/IP communication functions are not part of the TPI and have to be provided, except when using the high-level TPI.

The TCP/IP API functions of your operating system (OS) can also be used. Keywords under VC++ include *Win32 Sockets API*, or (if using MFC) *CAsyncSocket* and *CSocket*. Visual Studio contains a TCP/IP communication library,

MSWinsck.ocx, as an ActiveX control (COM object).

Applications with ActiveX controls must be Windows based, i.e. with a graphical user interface (GUI). Console applications are not suitable for ActiveX controls.

The use of a static TCP/IP library (for example Win32 Sockets API), or a TCP/IP communication DLL enables console (DOS) applications. Console applications have the advantage of comprising minimal overhead and are often 'single source file' applications.

The sample codes have examples of both GUI applications, with *MSWinsck.ocx* as the TCP/IP communication library with VC++ and DOS applications.

 See also the Reference Manual.

Version Compatibility

Future Compatibility

This is a very important issue in order to prevent client application software adjustments upon future emScon server software upgrades.The coming versions of emScon will include new data over the TCP/IP connection, such as new packet types, status messages and new error messages. The existing applications will be valid in future emScon versions with one important caveat.

 See "Backward Compatibility" on page 12 for details.

Backward Compatibility

Backward compatibility will be provided, in that existing packets/information structure are neither changed nor removed. In practice, this generally means that the default case in switch statements should always be treated as 'neutral' (no action).

Example

The enum ES_SystemStatusChange in v1.2 contains only two members.

```
enum ES_SystemStatusChange
{
    ES_SSC_DistanceSet,
    ES_SSC_LaserWarmedUp,
};
```

A future programming statement as follows would cause an 'Unexpected Status' message, with future emScon upgrades.

```
switch (status)
{
   case ES_SSC_DistanceSet:
     MessageBox("ADM Distance
                re-established");
    break;

   case ES_SSC_LaserWarmedUp:
     MessageBox("Laser is now
                 ready");
    break;

   default:
           MessageBox("Unexpected Status");
     break; // WRONG!!!
};
```

Solution

Ignore the default case with no 'default' entry tag or one that just has an effect to debug versions.

```
  default: // No action at all
     break;

or

  default: // no effect to retail versions
     TRACE("Unexpected Status");
     ASSERT(false);
     break;
```

Backward Compatibility v1.0/1.1

A minor change to version 1.2 may cause backward compatibility problems, in rare cases, if all of the following conditions apply:

- Application is based on LTControl (COM-Interface).
- EmScon server and LTControl have been upgraded (there is no problem if only the server is upgraded but the client still uses v1.x LTControl).
- One of MeasureStationaryPoint/MeasureStationaryPointEx methods is used.

- The Measurement mode before calling these functions was one of the 'Continuous' modes (i.e. not explicitly set to 'Stationary').
  - In this case, v1.x emScon versions implicitly changed to 'Stationary'. This is no longer the case in v1.2. There will be an error message, 'that it is not possible to make a stationary point measurement, while not being in stationary mode'.

Solution

If this situation should occur to your v1.x emScon client, try the following workaround:

1. A compatibility switch is required, in order to make v1.2 LTControl behave the same as v1.x.
2. Place file LTControl.ini in Windows directory (e.g. C:\WinNT).
3. Make following entry :
   [Settings]
   V1BackwardCompatibility=1


Do not forget to delete this file or to set the Entry to ='0' if the backward compatibility behavior is no longer wanted. It is recommended to change applications to new v1.2. behavior, as soon as changes are made to the client software.

Sample Code

The samples/tutorials show the principles of TPI programming. However, the sample applications may not be of real practical use, in the specific TPI commands they implement.


In a practical application, in order to get accurate results, it is crucial to implement all the steps as listed under 'Initial steps'.

See "Integration in Application Software" on page 15 for details.

The number of files and overhead has been kept to a minimum. Code generated from wizards, such as *recompiled headers, icon, res2 includes* and 'cosmetic functions', have been stripped off.

Following the tutorial instructions may result in some blown-up code compared with the samples bundled with this manual/SDK. However, the essential code remains the same.

See also the numerous comments in the sample source files.

See "Appendix " on page 140 for a complete list of sample source files.

Error Handling

The samples do not always implement complete error handling and may need to be run through the debugger.

Interface Design

The user interface design is kept at a minimum level (for example, unavailable buttons are not grayed out). Such items are general issues of Windows programming.

Hard Coded Information

The samples contain some hard coded information (IP address/coordinate values) that must be adapted to the local environment.

Integration in Application Software

The emScon graphical user interface, emScon Base User Interface (emScon BUI), provides a browser based access from the Application Processor. Access over the browser requires the IP address of the emScon Tracker Server.

The BUI includes:

- A Toolbar for sensor control and display of results and their statistics.
- Web pages providing access for sensor and server settings.

Integration of the application software running on the Application Processor with the emScon BUI is explained in Sample 13.

 See the emScon Reference Manual for BUI details.

Initial steps

Essential Steps

A client application must carry out all steps upon startup. Omitting some of these steps will prevent the tracker from measuring or lead to inaccurate results. Inaccurate results are difficult to detect.

 Setting correct environment parameters (temperature, pressure) or configuring the system for automatic, environment parameter reading is crucial.

 The environment parameter setting needs to be done before tracker initialization.

 Sample 7 implements all essential steps. Other samples are not exhaustive and show programming principles only.

 See also Leica Tracker/Training Manual.

Command Sequence

Steps	TPI command
1. Establish TCP/IP connection.	Depends upon TCP/IP communication – See different samples
2. Set units (length, angle, temperature and pressure)	ES_C_SetUnits

3. Set current environmental temperature, pressure and humidity	ES_C_SetEnvironmentParams
4. Initialize the Laser Tracker	ES_C_Initialize
5. Measurement mode (stationary, continuous time…)	ES_C_SetMeasurementMode
6. Get reflectors	ES_C_GetReflectors
7. Set reflector	ES_C_SetReflector
8. Go bird bath	ES_C_GoBirdBath
9. Station parameters	ES_C_SetStationOrientationParams
10. Transformation parameters	ES_C_SetTransformationParams
11. Coordinate system type (RHR, LHR…)	ES_C_SetCoordinateSystemType

General information

Provides an overview of the parameters and their implications.

Initialize Laser Tracker	**Definition**	**Comment**
	Initialize encoders and internal components	This command has to be performed every time you set up a new Leica Tracker system station. It is strongly recommended to use this function 2-3 times a day to initialize encoders and its internal components. This is important due to thermal expansion of the tracker hardware, which has a direct influence on the measurements
Current Temperature and Pressure	**Definition**	**Comment**
	Set index of refraction	With the input of the environmental temperature, pressure and humidity, the system calculates the light refraction index of the interferometer (IFM) and the absolute distance meter (ADM). These parameters have a direct influence on the measurement accuracy A change of 1°C causes a measurement difference of 1ppm. A change of 3.5mbar causes a measurement difference of 1ppm.

	Definition	Comment
		Change environmental parameters when significant changes take place. Default values: 20.0 °C, 1013.3mbar
Set Reflector	**Definition**	**Comment**
	Select a specific reflector	A wrong reflector results in a wrong initial IFM distance, e.g. when using the *Go Birdbath* command. This has a direct influence on the absolute positioning accuracy of the measurement network. ADM measurements are inaccurate due to the different reflector constants. Tooling ball reflector (TBR) = 5.310 mm Cat eye = 59.114 mm
Set Temperature Range	**Definition**	**Comment**
	The laser's mode of operation depends on external temperature. Select one of the following temperature ranges corresponding to your application: Low: +5 - +20 °C Medium: +10 - +30 °C High: +30 - +40 °C	The Laser Startup Mode is set by default to medium (+10 - +30 °C). When the external temperature exceeds this range, the system tries to stabilize the interferometer. During this process (10 - 20 minutes), no measurements can be taken (Green LED on the

	Definition	Comment
		tracker blinks). Switch mode to low or high.
Station Parameters	**Definition**	**Comment**
	The station parameters describes the translation and rotation of the tracker station in a coordinate system: X, Y, Z, Omega, Phi, Kappa	In case the station parameters are not set explicitly, TPI will set the parameters as follows: (X=0/Y=0/Z=0/Omega=0/ Phi=0/Kappa=0).
Transformation Parameters	**Definition**	**Comment**
	A transformation describes a change into another coordinate system, which is different from the tracker coordinate system. It has the following parameters: X, Y, Z, Omega, Phi, and Kappa and scale factor.	The object coordinate system is located in the measured object, which may correspond to the coordinate system of the design. Either a controlled orientation or a transformation can create the object coordinate system. Data is created in the object coordinate system if the transformation parameters are applied to the TPI. In case these parameters are not set, the TPI will deliver the data based on the tracker coordinate system (X=0/Y=0/Z=0/Omega=0/ Phi=0/Kappa=0/Scale = 1.

 See "Mathematics" on page 129 for more information.

	Definition	Comments
Coordinate System Type	Selects the coordinate system type: RHR/LHR X, LHR Y, LHR Z/CCW/CCC/SCW/SCC	The TPI delivers the data in the current coordinate system type. By default the tracker system will work in the right handed rectangular coordinate system (RHR) type: 3D rectangular coordinates are defined by 3 mutually perpendicular axes X, Y and Z given in the order (X, Y, Z). Since the axes can be arranged in two different ways, right and left-handed systems are defined according to the convention illustrated in a simple 2D case. Cylindrical Clockwise (CCW) Cylindrical Counter Clockwise (CCC) In a cylindrical system the X and Y values are expressed in terms of a radial (distance) offset from the Z-axis and a horizontal angle of rotation. The Z coordinate remains the same. Spherical Clockwise (SCW) Spherical Counter Clockwise (SCC)

	Definition	Comments
		In a spherical system a point is located by a distance and two angles instead of the 3 coordinate values along the rectangular axes. For axes labeled XYZ, with Z vertical, the point is located by its distance from the origin, horizontal angle in the XY plane and zenith angle measured from the Z-axis.

2.C Interface

Low-level programming

Introduction

Before designing the client application, refer to the *ES_C_API_Def.h file*. The C-TPI is a pure collection of enumeration types and data structures. The data structures reflect the 'architecture' of the data packets (= byte arrays) sent and received over the TCP/IP network, between the Application Processor and the Tracker Server.

 No functions or procedures are defined.

 Since C++ is an extension of C, a C++ compiler can also be used for C programming.

TCP/IP Connection

1. Establish a TCP/IP connection to the tracker server. This is typically achieved by invoking a *Connect* function of the TCP/IP communication library or toolbox. This function will take the IP address (or its related hostname) of your Tracker Server.
2. Set the TCP/IP Port Number to 700 for the Tracker Server.

Sending Commands

3. Call a *SendData* function from the TCP/IP communication library or toolbox (Function name may differ). This function typically takes a pointer to a data packet and probably the size of it (unless the packet is wrapped

into a structure that knows its size implicitly, for example a *Variant* structure).

4. The architecture of the packets (TPI protocol) is defined by the data structures in the *ES_C_API_Def.h* file.

    See the Reference Manual for a detailed description of these structures.

Code Sequence

5. For invoking a *GoPosition* command. Assign appropriate initialization values. For example, assign an *ES_Command* and an *ES_C_GoPosition* to a *GoPositionCT* struct variable.

    The compiler will not detect, if, for example, an *ES_DT_SingleMeasResult* as type, or an *ES_C_SwitchLaser* as command is assigned to a *GoPositionCT* variable. Inappropriate initialization values cause the command to fail.

```
GoPositionCT data;  // declare packet variable
data.packetInfo.packetHeader.type = ES_DT_Command;
data.packetInfo.packetHeader.lPacketSize = sizeof(data);
data.packetInfo.command = ES_C_GoPosition;
data.dVal1 = -1.879;
data.dVal2 =  2.011;
data.dVal3 =  0.551;
data.bUseADM = FALSE;
```

Initialization Macros

6. To avoid such errors, which may happen through copy/paste errors and are difficult to trace, it is recommended to use initialization macros for correct assignment of type, size and command values.
   An *INITStopMeasurement* macro , for example, requires two statements, the parameter declaration and the parameter initialization (macro call). The *StopMeasurement* has no additional command parameters. If there are any, these can be incorporated into the macro.

```
StopMeasurementCT  cmdStop;      // declaration
INITStopMeasurement(cmdStop);    // initialization
```

C++ Initialization

C++ offers a much more elegant way for initialization – the 'constructor' approach, which eases the initialization issues.

See "C++ Interface" on page 56 for details.

7. After initialization of the data variable, send it to the tracker server using the TCP/IP *SendData()* function (or whatever this function is called). Depending on the TCP/IP communication library used, the data packet may need to be packed into a Variant *vrtData* variable, followed by a *SendData* (*vrtData*) call. Alternatively, a *Send()* function takes the address and size of the data packet variable, Send (&data, sizeof(data)).

Answers from Tracker Server

8. The *SendData()* function does not wait for the Tracker Server (tracker) to complete the requested action - *SendData()* will return immediately. On completion of the requested action, the tracker server sends an answer to the client. Depending on the command, it may take a few seconds between sending the command and receiving an answer. This requires some type of notification or callback mechanism. That is, as soon as data arrives from the Tracker Server, some sort of event needs to trigger a *ReadData()* procedure in the client application. Depending on the TCP/IP communication, this notification could be a Windows Message, an Event or a Callback Function.

This type of communication is asynchronous.

Asynchronous Communication

9. From the programmer's point of view, asynchronous communication is much more difficult to handle than synchronous communication. The programmer must ensure, not to send a new command until the answer of the previous one has returned (commands might be queued under certain circumstances).

DataArrived Notification

10. All TCP/IP communication libraries/toolkits contain either a *DataArrived()* notification or a similar function, which is called by the framework each time data has arrived. Depending on the toolkit:
    - The function may directly return a Variant type parameter that contains the data.
    - The function may deliver the data within a byte array.
    - The function returns the size of the data packet that is ready to be read. In this case, the *DataArrived()* function subsequently calls a *ReadData()* function immediately, in order to get the data into a local byte array.

DataArrived Notification Queue

11. If a 'traffic jam' occurs on the incoming TCP/IP line, i.e., if incoming data is being queued, a *ReadData()* call will read all the currently available data with no notification for each individual packet. Many packets may be queued and only one *DataArrived* notification might be issued. This means that the *byteArr* buffer will contain more than one packet. This may occur on high frequency, continuous measurement streams. The application has to make provisions to

correctly treat such cases. The *lPacketSize* value is most convenient when parsing the *byteArr* buffer.
If the *byteArr* buffer is completely filled with data, it is likely that the last packet in the *byteArr* is incomplete. The packet fragment needs to be saved and padded to complete upon the subsequent read-call.

 See "Queues and Scattered Data" on page 91 for details.

12. Assuming a received data block has been read into a byte buffer named *byteArr*. In order to interpret the data, a mask is required. This requires knowledge of the type of data packet (enum *ES_DataType*). A typical *PacketHeader* interpreting code is as follows:

PacketHeader Code

```
PacketHeaderT *pData = (PacketHeaderT*)byteArr;
```

13. Access the type and the size of the packet can be with:

```
pData->type;
pData->lPacketSize;
```

The packet size is only for convenience. *Sizeof(type)* also returns the packet size.

 This redundancy may be used for consistency checks and is helpful when using programming languages other than C that lack the *sizeof()* operator).

The packet size is reliable on received packets. When sending packets to the Tracker Server, it is recommended to initialize the *lPacketSize* variable correctly, although the Tracker Server ignores it. This approach has been chosen to reduce possible programming errors.

Command Subtype Switch

14. Command type answers require a switch statement to distinguish the command subtype. Non-data returning commands can all be treated the same and are handled in the default switch statement. All other command answers need to be masked with the appropriate result structure. The code fragment below demonstrates this with the *GetUnits* command, and shows part of the handling of a single measurement answer:

```
switch (pData->type)
{
   case ES_DT_Command: // 'command- type' answer arrived
   {
      BasicCommandRT *pData2 = (BasicCommandRT *)byteArr;

      // if something went wrong, no need to continue
      if (pData2->status != ES_RS_AllOK)
      {
         // TODO: evaluate and handle the error
         return false;
      }

      switch (pData2->command)
      {
         case ES_C_Initialize:
         case ES_C_PointLaser:
         case ES_C_FindReflector:
            break;

         case ES_C_GetUnits:
         {
            GetUnitsRT *pData3 = (GetUnitsRT *)byteArr;

            // Diagnostics - check whether packet size
            // as expected (in debug mode only)
            ASSERT(pData3->packetInfo.
                        packetHeader.lPacketSize ==
                                sizeof(GetUnitsRT));

            // now you can access Unit specific data.
            pData3->unitsSettings.lenUnitType;
            pData3->unitsSettings.tempUnitType;

            break;
         }

         // case XXX:
         // Todo: add other command type evaluations
         //    break;

         default:
            break;
      }
   }
   break;

   case ES_DT_SingleMeasResult: // single-meas-result-
   {                            //type answer has arrived
      SingleMeasResultT *pData4 =
                  (SingleMeasResultT *)byteArr;

      if (pData4->packetInfo.status != ES_RS_AllOK)
         return false;

      break;
   }

   // Todo: add further 'case' statements
   // for remaining packet types
}
```

- Declaring variables within case statements, which are suitable for masking data, require curly brackets around a particular case block. Otherwise the compiler will claim.

- If-then-else can be used instead of switch statements. However, switches are more efficient.

- Frequent items should be treated at the top of a switch statement, for example multi-measurement results (not covered above).

C Client Applications

Sample 1-Tutorial

The *EmsyCApiClient* sample is presented in tutorial form, based on Visual C++ v6.0, it uses MSWinsck.ocx for TCP/IP communication. The application is given a dialog box 'look and feel'.

 An OCX control requires a dialog/Windows based application, with VARIANT data types.

Step 1: Creating an Application Framework using AppWizard

1. Launch Microsoft Visual C++ (Visual Studio).
2. Select *File* > *New* > *Projects*.
3. Select *MFC AppWizard* (exe) from list.
4. Type *EmsyCApiClient* in the Project Name field.
5. Click *OK*.
6. Select *Dialog-based*. Click *Next*.
7. Uncheck *AboutBox*.
8. Check *3D controls* and *ActiveX Controls*. Click *Next*.
9. Select (optional) *No, thank you* under, "*Would you like to generate source file comments?"* Leave other settings unchanged. Do not change any file names.
10. Click *Finish*. Click *OK* on last pane.
11. Compile the application.
12. Run the application.

Step 2: Import the Winsock Control

1. Select menu *Project* > *Add to Project* > *Components and Controls*.
2. Change to Registered ActiveXControl folder.

3. Select Microsoft Winsock Control, version 6.
4. Click *Insert*, Click *OK*.
5. Reregister MSWinsck.ocx, if Microsoft Winsock control version 6.0 is not available, in the list of Registered ActiveX controls. Use *Regsvr32.exe* utility. Reinstall VC++, if file cannot be found.
6. Dialog box 'Confirm Classes' appears.
7. Check item 'CMSWinsockControl'. Leave proposed class and file names unchanged.
8. Click *OK*.
9. Close Components and Controls Gallery dialog. Two files *MSWinsockControl.h* and *MSWinsockControl.cpp* have been generated as a wrapper class to the OCX's COM-Interface and inserted in the project. No changes are to be made to these files. Only a few of the many functions of *MSWinsockControl* are required by the application.

Step 3: Create a Winsock Controls Instance

Knowledge about placing controls on dialog boxes with the Dialog Box editor is presumed. The Controls palette (the window from which buttons, edit fields etc. are dragged to the dialog template) is padded with a new icon at the bottom (two small computers are shown). Its tool-tip info reads 'MS Winsock control version 6.0'.

1. Create one such control in the dialog: Activate the resource editor and the dialog template. In the 'Controls' toolbox, click the *Winsock icon*.
2. Draw a rectangle on the dialog. The icon should now appear as a 32x32-pixel icon. Place icon out of the way, where it does not

disturb other controls.

 This icon will be invisible in the real dialog.

3. Right click to look at the properties: ID: IDC_WINSOCK1. Do not change the ID.
4. Click *Control TAB* of the property box. Add an empty string "" to *Remote Host* field. This prevents an exception when running the application.
5. Do not change any remaining properties. No Remote host ID or port #. This information will be added later.
6. Add member variable of type *CMSWinsockContro*l (see *MSWinsockControl.h* file for class definition).
   Manually: Continue with step 7.
   With Class Wizard: Continue with step 9.
7. Add following declaration:
   manually to the applications dialog class *CEmsyCApiClientDlg*.

```
CMSWinsockControl  m_winSockCtrl;
```

8. Create the control explicitly or make a correct entry to the *DoDataExchange()* function.
9. Let the *ClassWizard* do the work.
   Call it from the *View* menu or press *Ctrl+W*.
10. Activate the *Member Variables* TAB.
11. Select IDC_WINSOCK1 in the list. Click *Add Variable…* button. Name the variable *m_winSockCtrl*.
12. Click *OK,* close the *ClassWizard* window.

Step 4: User Controls on the Dialog

The *MSWinsock* control's functions take all string and data parameters as Variants.

Use of Variants (packing/unpacking before sending/receiving data) is necessary, due to the Variant parameters of the MSWinsck.ocx Send/Receive functions.

Use of an alternative TCP/IP library (with C or C++ Interface) or the native TCP/IP communication functions of the operating system enable design of non Win32- based or non-Windows (e.g. console applications, Apple OS/Linux etc) applications. These will not require Win32 specific Variants.

 See "Sample 3" on page 52 or "Receiving Data Sample 9" on page 66 for details.

InitDialog() Handler

 See "Sample 4" on page 57 for explanation of the *InitDialog()* handler with the implicit *Connect()* function and close with *WM_DESTROY*.

Connect/Disconnect Handlers

The ClassWizard's Message Maps TAB can be used to add push buttons to dialog templates and to attach appropriate message handlers to them, together with *Static* and *Edit* controls to dialogs with member variables.

 See "Sample 1-Tutorial" on page 30 for explicit *Connect* and *Disconnect* handlers, bound to appropriate push buttons with all required controls added to the dialog template.

 Refer to VC++ 'Scribble' tutorial for information on dialog box design.

1. Remove the *OK* button and the TODO label from the dialog template.
2. Replace the caption of the *Cancel* button with *Exit*.

3. Add the following 7 pushbuttons and the related message handlers to the dialog template.

 In order to correspond with the Sample 1 source code, the following names are recommended:

Button Caption	Button ID	Name of Message handler
Connect	DC_BUTTON_CONNECT	OnButtonConnect
Disconnect	IDC_BUTTON_DISCONNECT	OnButtonDisconnect
Init Tracker	IDC_BUTTON_INIT	OnButtonInit
Laser Pointer	IDC_BUTTON_POINTLASER	OnButtonPointlaser
Find Reflector	IDC_BUTTON_FINDREFLECTOR	OnButtonFindreflector
Set Reflector	IDC_BUTTON_SETREFLECTOR	OnButtonSetreflector
Start Meas	IDC_BUTTON_STARTMEAS	OnButtonStartmeas

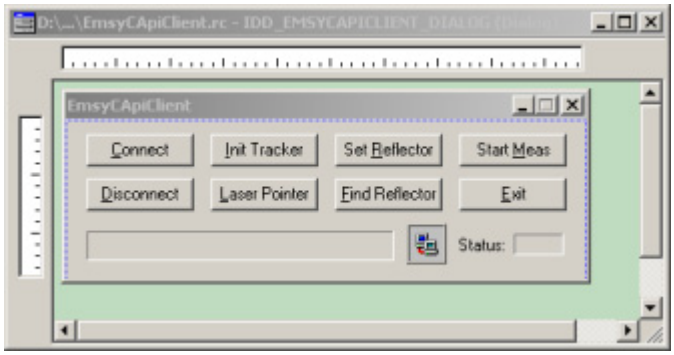
4. Add two 'Statics' and relate them to appropriate member variables (type Control):

Static ID	Member variable name
IDC_STATIC_RESULT	m_staticResult;
IDC_STATIC_STATUS	m_staticStatus;

5. Eliminate the 'captions' of these Statics.
6. Check the 'sunken' property (to make then visible). Their size should permit three (3)

coordinate values (Result) or a three (3) digit-number (Status).
A label (static) may be added, with caption 'Status' in front of the status field. No member variable need be added and no change of the ID is necessary.

7. The dialog template should now look as follows:



Step 5: Connect/Disconnect TCP/IP Handlers

The *EmsyCApiClientDlg.cpp.h* should now contain 'skeleton' code for all push button handlers (generated by the ClassWizard).

1. Implement the OnButtonConnect() and OnButtonDisconnect() handlers.
2. Include *atlbase.h* to know *CComBSTR* and *ES_C_API_Def.h*. Add these files to the project with Menu *'Project' > Add to 'Project' > Files*.

```
#include <atlbase.h>
#include "ES_C_API_Def.h"
```

3. The essential *OnButtonConnect()* code reads as follows (fragment only).

```
VARIANT vRemoteHostName;
VariantInit(&vRemoteHostName);
vRemoteHostName.vt = VT_BSTR;

// CComBSTR ensures allocating and freeing the string
CComBSTR bstr(_T("192.168.0.1")); // hostname or IP adress
vRemoteHostName.bstrVal = bstr;

VARIANT vRemoteHostPort;
VariantInit(&vRemoteHostPort);
vRemoteHostPort.vt = VT_I4;
vRemoteHostPort.lVal = 700; // port 700 reserved  for ES

// So far, all was Variant preparation. Do connect call now
m_winSockCtrl.Connect(vRemoteHostName, vRemoteHostPort);
```

 When *MSWinsock* control throws exceptions on failure use *try {}/catch {}* statements.

 See "Sample 1-Tutorial" on page 30 for the complete implementation of the *OnButtonConnect()* function with useful comments.

4. The essential code of the *OnButtonDisconnect()* handler:

```
m_winSockCtrl.Close();
```

5. To trap *MSWinsock* errors, the entire implementation of *OnButtonDisconnect()* is shown here:

```
void CEmsyCApiClientDlg::OnButtonDisconnect()
{
   try
   {
      m_winSockCtrl.Close();
   }
   catch(...)
   {
      AfxMessageBox(_T("Failed to close connection\n"));
   }
}
```

Build the application and connect/disconnect to the Tracker Server. The *netstat.exe* Windows tool checks whether the connection was established correctly.

Step 6: Implementing Command Handlers

1. Implement the remaining command handlers for the commands (Buttons) that have been previously added. Select the appropriate command structure (as defined in the *ES_C_API_Def.h* file) and initialize it.

2. Sample *Initialize Tracker* command handler:

```
void CEmsyCApiClientDlg::OnButtonInit()
{
   // clear status field, just cosmetics!
   m_staticStatus.SetWindowText(_T(""));

   try
   {
      InitializeCT data;

      data.packetInfo.packetHeader.type = ES_DT_Command;
      data.packetInfo.packetHeader.lPacketSize =
                                              sizeof(data);
      data.packetInfo.command = ES_C_Initialize;

      COleVariant variantData =
                     PackIntoVariant(&data, sizeof(data));
      m_winSockCtrl.SendData(variantData);
   }
   catch(...)
   {
      AfxMessageBox(
           _T("MSWinSockCtrl Exception (Initialize)\n"));
   }
}
```

PackIntoVariant() helper function is necessary due to the Variant type parameters of *MSWinsock*. The *PackIntoVariant()* needs to be defined in the application code (*for example locally in the EmsyCApiClientDlg.cpp file*)

PackIntoVariant() requires <atlbase.h>:

```
COleVariant PackIntoVariant(void *pData, UINT sizeOfData)
{
   BYTE* pByteArr = new BYTE[sizeOfData];
   memcpy(pByteArr, pData, sizeOfData);

   CByteArray byteArr;

   for (UINT i = 0; i < sizeOfData; i++)
      byteArr.Add(*(pByteArr+i));

   delete [] pByteArr;

   COleVariant vrt(byteArr);

   return vrt;
}
```

Tracker Initialization

The application should now be able to initialize the tracker.

1. Run the application and immediately click the *Init Tracker* button.
2. An error message reads: *MSWinSockCtrl Exception (Initialize)\n* (the message specified in the *OnButtonInit()* 'catch' part). The

Winsock control threw an exception because it was not connected before!

3. Press the *Connect* button, then the *Init Tracker* button again.

   Only one, single *Connect* at a time is possible. Pressing the *Connect* button more than once without a *Disconnect* in-between, will get an error message, although the connection is still OK.

4. Implement the remaining Handlers:
   OnButtonPointlaser()
   OnButtonFindreflector()
   OnButtonSetreflector()
   OnButtonStartmeas()
   Refer to Sample 1 code.

Hard Coded data

Some of these functions include hard coded data, for example coordinates for *OnButtonPointlaser()*. Invoking this command just toggles the beam between two locations. This is not of practical use, but for demonstration.

The coordinates may need to be changed to reflect your environment. It is recommended that a reflector is placed somewhere with approximate coordinates specified in (one of) the *OnButtonrPointlaser()* toggle positions. It is also assumed that a reflector with *ID 1* exists, as set in *OnButtonSetreflector(1)*. Change the ID as required.

Step 7: Receiving Data

In order to receive data, the events sent out by the Winsock control need to be caught, as soon as arriving data is signalled by the framework (by an Event).

1. Display the ClassWizard dialog, choose the *Message Maps TAB* and select the

IDC_WINSOCK1 item in the Object IDs table.

 Ensure the dialog class in the 'class name' list box is selected . The 'Messages' list shows all events the Winsock control is able to send.

2. Select *DataArrival,* press *Add Function...* and confirm the name *OnDataArrivalWinsock1*. Finish the ClassWizard with *OK*.
3. Searching all project files by 'arrival' shows, that the class wizard has inserted code in 3 locations:
   - Function declaration in the header files message map section.
   - An entry in the implementation files EVENTSINK map.
   - An empty function body.

Function Body

```
void
CEmsyCApiClientDlg::OnDataArrivalWinsock1(long bytesTotal)
{
    // TODO: Add your control notification handler code here
}
```

4. If a *Data Arrived* event happens, the number of bytes that are ready to be read is passed as a parameter to *OnDataArrivalWinsock1()* handler. An *OnDataArrivalWinsock1()* call requires the Winsock control's *GetData()* function.

GetData Function

```
m_winSockCtrl.GetData(&m_vtData, vtType, vtMaxlen);
```

5. The address of a variant must be passed as first parameter, since the data is delivered as a variant. To avoid frequent allocation/deallocation and initialization of (an automatic local) Variant variable, a Variant member variable is defined in the dialog class.

6. Add the following declaration to the *CEmsyCApiClientDlg* class (protected or private) in the *EmsyCApiClientDlg.h* file:

```
VARIANT m_vtData;
```

7. Initialize this variable once to the 'dialog class' constructor:
   Upon receiving data, the application may crash without initialized m_vtData

```
VariantInit(&m_vtData);
```

8. To maintain modularity, implement data-reading (GetData) in the *OnDataArrivalWinsock1()* function, and implement a separate processing (parsing) member-function named *ProcessData()*.
9. The *OnDataArrivalWinsock1()* implementation reads as follows:

```
void
CEmsyCApiClientDlg::OnDataArrivalWinsock1(long bytesTotal)
{
   TRACE(_T("OnDataArrival\n"));

   // m_vtData (output parameter variant) as member var
   // !!! Note: m_vtData must be initialized with
   // VariantInit; see constructor !!!

   // pass expected result-type through iVal
   VARIANT vtType;
   vtType.vt=VT_I2;
   vtType.iVal = VT_ARRAY|VT_UI1;

   // pass packet length through lVal (long) parameter
   VARIANT vtMaxlen;
   vtMaxlen.vt=VT_I4;
   vtMaxlen.lVal = bytesTotal;

   try
   {
      // read data from socket into variant
      m_winSockCtrl.GetData(&m_vtData, vtType, vtMaxlen);
   }
   catch(...)
   {
      AfxMessageBox(
              _T("MSWinSockCtrl Exception (GetData)\n"));
   }

   // check if all data has arrived - just diagnostics
   if (m_vtData.parray->rgsabound->cElements ==
                              (unsigned long)bytesTotal)
   {
      // now data can be processed
      if (!ProcessData())
         Beep(200, 100);
   }
   else
      ASSERT(false); // will only raise in debug mode
}
```

ProcessData()

To implement *ProcessData()* parse incoming data as described earlier in this document. The implementation of *ProcessData()* as a member function does not require passing data as a parameter. The *m_vtData* member variable can be directly accessed.

10. Mask the data buffer (byte-array) with a *PacketHeaderT* struct to identify the packet type.
11. Depending on the packet type, use the appropriate 'Sub-mask' (*BasicCommandRT*, *SingleMeasResultT* etc.).

This simple application handles only a few packet types/commands, and the *ProcessData* function remains compact. For applications that

implement virtually all Tracker Server answers, the parsing code must be split into smaller functions for ease of handling.

 Note the statements to display return status numbers (0 = OK, non zero = errors) and measurement results to the appropriate static controls of the dialog.

```
bool CEmsyCApiClientDlg::ProcessData()
{
   CString sStatus, s;

   // ProcessData() is a parser for the incoming data.
   // When ProcessData() is being called, we can assume
   // that 'm_vtData.parray->pvData' points to a valid and
   // complete 'answer block' (just an array of bytes).
   // Next we must mask this data block with BasicCommandRT
   // in order to figure out type. Once the type (for
   // example ES_DT_SingleMeasResult) is known, we can mask
   // it with the appropriate structure (e.g.
   // 'SingleMeasResultT'). mask arrived data with RT
   // structure in order to figure out type/status

   PacketHeaderT *pData =
                  (PacketHeaderT*)m_vtData.parray->pvData;

   switch (pData->type)
   {
      case ES_DT_Command: //'command- type' answer arrived
      {
         BasicCommandRT *pData2 =
                (BasicCommandRT *)m_vtData.parray->pvData;

         switch (pData2->command)
         {
            // here you may treat answers individually
            // as needed. Within this sample, we just do
            // a general handling for all command answers.
            // Thus the 'switch (pData->command)' block
            // could be entirely omitted. It is just here
            // for documentation purposes

            case ES_C_Initialize:
            case ES_C_PointLaser:
            case ES_C_FindReflector:
            case ES_C_StartMeasurement:
            case ES_C_SetReflector:
               // common error handling for all commands.
               // display error status. In a real program,
               // we would have to map these codes to text
               // strings describing the error

               sStatus.Format(_T("%d"), pData2->status);
               m_staticStatus.SetWindowText(sStatus);

               // handle error
               if (pData2->status != ES_RS_AllOK)
                  return false;
               break;

               // TODO: add further cases as other
               // commands become implemented.
               // See file ES_C_API_Def.h for further
               // commands and related answer structures.

               default:
                  return false;
         } // switch (pData->command)
      }
      break;

      case ES_DT_SingleMeasResult:
      {
         // A 'single-meas-result- type' answer
         // has arrived so mask it with
         // SingleMeasResultT structure

         SingleMeasResultT *pData3 =
             (SingleMeasResultT *)m_vtData.parray->pvData;

            // if something went wrong, there
            // is no reason to continue

            if (pData3->packetInfo.status != ES_RS_AllOK)
```

```
                return false;

             // Do something with the data (display it)

             s.Format(_T("X=%.3lf, Y=%.3lf, Z=%.3lf\n"),
               pData3->dVal1, pData3->dVal2, pData3->dVal3);

             m_staticResult.SetWindowText(s);

             // also display return status

             sStatus.Format(_T("%d"),
                            pData3->packetInfo.status);

             m_staticStatus.SetWindowText(sStatus);
          }
          break;

       // case ....
       // Todo: add other answer- cases

       default:
          return false;
          break;
    } // switch (pData->type)

    return true;
} // ProceessData()
```

Single points can be measured and displayed in the dialog. *Set Reflector* followed by a *Find Reflector* must be called before a measurement is triggered.

Asynchronous Communication

The application must have provisions to prevent pressing another button before an answer has arrived. (Some commands may be queued). An answer has arrived, when the status code is displayed. The command is OK when code is zero (0). The Tutorial sample does not fully handle these errors.

If the code is a number, refer to the Tracker/ Software reference manual.

Remarks

This sample demonstrates the C-TPI together with C++ clients and an MFC-using Windows application. Writing in C is still an option for non-Windows or non- Win32 applications. Further issues to be considered in a real application:

Interface Design

- User interface design: Grayed-out controls depending on context. Lock buttons in a situation when it is not appropriate to press them. Graying out all buttons during the time gap between commands sent and answer received is recommended for asynchronous communication.
- Other design issues include: proper TAB order of the controls and assigning shortcuts to dialog controls.

Error Handling

- Error handling: Errors should be handled exhaustively. Provisions should be made for all possible errors. Error messages should be defined as resources instead of hard coded strings. Implement exception handling where it applies (try/catch).

Network Traffic Jams

- Make provisions for 'network traffic jams' and receiving buffers of constant size. When the buffer is completely filled upon a read operation, it is very likely that part of the last packet will be 'distributed' over the buffer boundary. That is, part of the packet will not become available before the next 'Read' call.

   The buffer size (variant array) is allocated dynamically in this sample.

   See "Queues and Scattered Data" on page 91 for details.

MSWinsock Control

- Using the *MSWinsock* control is very convenient together with VC++ (ClassWizard etc.). A problem, however, may be its performance (COM interface and Variant copy). An *MSWinsock* based application may not be able to carry high frequency continuous measurements. For such high

performance applications, use of low-level C/C++ TPI is recommended in combination with a high performance TCP/IP communication library.

Essential code

- All essential code is concentrated in only two files *EmsyCApiClientDlg.cpp* and *EmsyCApiClientDlg.h* (apart from the generated wrapper classes for the OCX, which reside in *mswinsockcontrol.cpp/h*).

Build Sample 1

To build Sample 1 from the files in the SDK, open *EmsyCApiClient.dsw* with VC++ and choose 'Build' (or click *F7*).

TCP/IP address

- Change the (hard coded) TCP/IP default address ('192.168.0.1') according to the Tracker Server's address.

Visual Basic Client Application

Sample 2-Tutorial

The *ES_C_API_Def.h* cannot be directly included in Visual Basic (or Pascal, Java etc.) applications. It needs to be translated to other languages' syntax with the risk of errors (typing errors, different structure byte alignment, different sizes of basic data types etc.).

The use of languages other than C/C++ for using the low-level TPI is not recommended or supported.
This sample serves primarily to enable a better understanding of the TPI principles. It has functions limited to connecting to the tracker, initializing and disconnecting.

TCP/IP

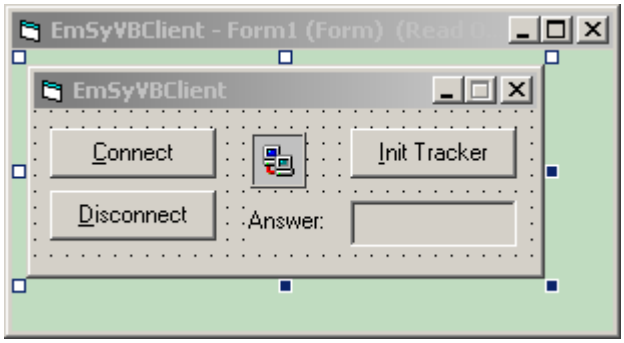
TCP/IP communication uses *MSWinsck.ocx*.

Step 1: Adding a Winsock Control and Designing a Form

1. Launch Visual Basic 6.0.
   Select *File* > *New Project* > *Standard.exe*. Click *OK*.

2. Save the form as *EmsyVBClient.frm* and the project as *EmsyVBClient.vbp*.
3. Select Menu *Project* > *Components* (or Ctrl+T).
4. In dialog list box, check *Microsoft Winsock Control 6.* Click *OK*.
5. Place an instance of the Winsock control icon in the application form (default name Winsock1).

   Do not change the default name.
6. Right mouse click the Winsock control icon on the form. Select *Properties*.
7. Enter the IP address or hostname of the Tracker Server in the *RemoteHost* field and the port number 700 to *RemotePort*. Leave the other settings as they are.
8. Add three buttons to Form Winsock1 *Connect*, *Disconnect* and *Init Tracker*. Add a (sunken) label named *AnswerStatus*.
9. The form reads as follows:



Step 2: Connect to the Tracker Server

1. Double click the *Connect* button. This adds a handler and switches the focus directly to the appropriate source location.
2. Insert one line of code, *Winsock1*:

```
Private Sub Connect_Click()
    Winsock1.Connect
End Sub
```

IP Address

3. If IP address and port number have not been passed through the properties.


See Point 7 "Step 1: Adding a Winsock Control and Designing a Form" on page 46.
Use the *Connect* call to insert address and port number of the Tracker Server: *Winsock1.Connect "193.8.34.133", 700*.

4. Double click the *Disconnect* button on the form and complete the handler as follows:

```
Private Sub Command1_Click()
    Winsock1.Close
    AnswerStatus.Caption = ""    'Just for cosmetics
End Sub
```

Winsock1

5. Entering Winsock1: As soon as '.' (dot) is entered a dropdown list offers all available methods and properties for the respective object. Typing a blank (or opening bracket) behind a method name gives IntelliSense support for all the method's parameters. Similar support is available in VC++.

Step 3: Translate the C- enums and Structs to VB

In order to create a command packet, the structures defined in the *ES_C_API_Def.h* file need to be translated to Visual Basic. Visual Basic uses four (4) Byte alignments by default and a VB 'long' is necessary for a C 'init'. A VB integer is only two (2) Bytes and therefore relates to a C 'short'. Doubles are eight (8) Bytes in both C and VB.

To implement the *Init Tracker* command, the following subset of 'enum' translations is provided. 'Enum' members VC start with zero (0) as default. Explicit values need to be assigned, since only subsets of the original C- enum types are translated (only as far as needed):

```
Enum ES_DataType
    ES_DT_Command
    ES_DT_Error
End Enum

Enum ES_Command
    ES_C_Initialize = 7
End Enum

Enum ES_ResultStatus
    ES_ES_AllOK
    ES_RS_NotImplemented = 2
End Enum

Further we need the following 'struct' translations:

Private Type PacketHeaderT
   lPacketLength As Long
   type As ES_DataType
End Type

Private Type BasicCommandCT
   packetHeader As PacketHeaderT
   command As ES_Command
End Type

Private Type BasicCommandRT
    packetHeader As PacketHeaderT
    command As ES_Command
    status As ES_ResultStatus
End Type

Private Type InitializeCT
   packetInfo As BasicCommandCT
End Type

Private Type InitializeRT
   packetInfo As BasicCommandRT
End Type
```

 See also detailed comments in the VB source code (File *EmsyVBClient.frm*)

Step 4: Implementing the Init Tracker Command

1. Create the function body for the command handler with a double click to the *InitTracker* button.
2. Declare a variable of *InitializeCT* and the *ES_C_Initialize* and *ES_DT_Command* tags filled in.
3. Send the data packet over the TCP/IP network.
4. The *InitializeCT* variable has to be copied to a Variant.
   The *SendData* function of the Winsock control takes a Variant parameter.

5. 'Borrow' the function *CopyMemory* from the Win32 API.
   Declare this function in the declaration part of the form:

```
Private Declare Sub CopyMemory Lib "kernel32" Alias _
     "RtlMoveMemory" (pDest As Any, pSource As Any, _
                      ByVal ByteLen As Long)
```

6. If *InitializeCT* variable is named *mt*. The sequence to copy *mt* into a variant reads as follows:

```
Dim bt() As Byte
ReDim bt(LenB(mt)) As Byte
Dim vtdata As Variant
CopyMemory bt(0), mt, LenB(mt)
vtdata = bt
```

Init (Tracker) handler

```
Private Sub Init_Click()
    AnswerStatus.Caption = ""  ' Just for cosmetics

    Dim mt As InitializeCT
    mt.packetInfo.command = ES_C_Initialize
    mt.packetInfo.packetHeader.type = ES_DT_Command

    ' This code-block should be rather put into a subroutine
    Dim bt() As Byte
    ReDim bt(LenB(mt)) As Byte
    Dim vtdata As Variant
    CopyMemory bt(0), mt, LenB(mt)
    vtdata = bt

    Winsock1.SendData vtdata ' Finally send the data
End Sub
```

7. An *Initialize Tracker* can now be invoked, however, no answers can be received.

Step 5: Implementing Answer Data Receiving

1. Select the *Winsock1* object in the code editor for Form1, top left drop down list.
2. Select *DataArrival* and *Error* , to create the bodies of these two handlers out of the Winsock control, top right drop down box.
3. Call *GetData*, which delivers the result in a Variant. Copy the variant into a variable of type *InitializeRT*.

```
Private Sub Winsock1_DataArrival(ByVal bytesTotal As Long)
    Dim vta
    Dim mt As InitializeRT
    Dim bt() As Byte
    Winsock1.GetData vta, vbArray + vbByte, LenB(mt)
    bt = vta
    CopyMemory mt, bt(0), LenB(mt)

    Beep 'Beep on Answer received

    If mt.packetInfo.status = ES_ES_AllOK Then
        AnswerStatus.Caption = " AllOK"
    Else
        AnswerStatus.Caption = " Unknown Error"
    End If
End Sub
```

4. Implement the error handler:

   Refer to comments in the source file.

```
Private Sub Winsock1_Error(ByVal Number As Integer, _
                           Description As String, ByVal _
                           Scode As Long, ByVal Source As _
                           String, ByVal HelpFile As _
                           String, ByVal HelpContext As _
                           Long, CancelDisplay As Boolean)
    Beep 'Beep on error received
    MsgBox "Winsock Error: " & Number
End Sub
```

Running the application

5. Set the IP Address before compiling and running the application.

   See Sample 2 folder in SDK for complete source code.

Remarks

This application can be extended to a full-featured Tracker Server 'controller' . It shows that the TPI is both language and platform independent (pure TCP/IP). Languages such as Pascal or Java can be used to build clients based on the C TPI directly.

The use of languages other than C/C++ for using the low-level TPI is not recommended or supported.

The Winsock control may not be efficient enough when dealing with 1000 points/second from a continuous measurement stream.

Structures

Structures are convenient for this purpose, and, in principle, pure Byte-arrays would also do,

which would lead to more complex initialization and interpretation.

COM interface

A COM interface can be used with virtually any programming language, without the hassle of translating the packet structures and Enums.

See "COM Interface" on page 69.

Sample 13 LT BUI Launch

This sample is a BUI launcher, which is to be used in the application, to launch the emScon BUI.

Refer to the documentation with the SDK for details.

Winsock 2.0 Client Applications

Sample 3

Implements a 'lightweight' C-TPI client application, with no graphical interface, variant overhead or MFC or ATL. This sample fits into a single file with 350 lines of code (including comments and empty lines), and compiles into a 48KB executable file.

This sample implements only *Initialize Tracker* and *Start Measurement* for single points, and requires events, threads and Winsock API functions.

Console application

This *EmsyCApiConsoleClient* is not presented as a tutorial. The VC++ AppWizard or a text editor can be used to create a 'Console Application' skeleton, and to implement the C standard entry function:

```
int main(int argc, char* argv[])
{
}
```

Add all the source code, save the file (.c or .cpp extension) and invoke the C compiler from the command line.

Comments

These comments refer to the file *EmsyCApiConsoleClient.cpp*.

The following include- files are required:

Windows.h need not be included

```
#include <stdio.h>     // standard C input/output
#include <Winsock2.h>  // win32 socket stuff
```

1. The *main()* function first does a TCP/IP connection by calling the function *TcpIpConnect()*, starts the *Data Receiver* thread and enters an endless 'User Interface loop'.
2. This loop looks for user input of one of the two TPI commands '*i*' for *Initialize Tracker* and '*m*' for *Start Measure*.
3. If the user enters *x*, the loop is stopped, the TCP/IP connection is closed and the application terminates.

   The *TcpIpConnect()* function is straightforward up to the call of *connect()*.
4. Call WSAStartup. After connecting, call *WSAEventSelect()*, which takes the following parameters:
   - A socket handle (that has been created before) as a global variable.
   - An event of type *WSAEVENT as* a global variable. This variable must be initialized with the return value of a *WSACreateEvent()* call.
   - A flags parameter. *FD_READ* is passed, indicating an interest in data-arrival events (a realistic application would have to also trap *FD_CLOSE* events).

5. Calling this function will cause the TCP/IP framework to signal the passed event, whenever data has arrived at the socket.
6. The *DataRecvThread()* has an infinite loop with the following statement:

```
WaitForSingleObject(g_hSocketEvent, INFINITE);
```

7. This is a blocking call and causes the loop to stop, until the event is signalled to be read. The blocking by the *WaitForSingleObject* is released and the loop passes on.
8. Reset the event before available data is read into a buffer.
9. Call a function *ProcessData()* that does the interpretation of the buffer.

Queuing (Traffic Jams)

There are no provisions to handle 'traffic jams' on the network. A real application needs to make provisions to handle such situations with a packet size transmitted in the header of each packet. The Winsock function *setsockopt()* may be used to 'tune' TCP/IP transmission rate by increasing buffer sizes.

 See Win32 documentation for more information about Winsock API (especially the WSA… function), threads and events.

 See "Receiving Data Sample 9" on page 66 for a more sophisticated data receiving thread.

 See also "Queues and Scattered Data" on page 91 and comments in source code.

Remarks

This sample can easily be ported to non-Win32 platforms (Unix, Linux, and Mac).

Creating a 'console' application requires the use of the *WSAEventSelect()* function with events and threads.

Windows application

For Windows applications, the *WSAAsyncSelect()* function is more appropriate. It issues Window messages instead of events and is simpler to handle. No separate thread is required (the window message loop takes this part).

See Win32 documentation on *WSAAsyncSelect()*.

Winsock API

The Winsock API functions are more efficient than the Winsock OCX control. The use of a MFC library permits a very convenient class wrapper around the Winsock API.

Refer to the *CAsyncSocket* and *CSocket classes* in “C++ Windows-MFC Application” on page 84 for details.

3.C++ Interface

Class Interface

Class design

The C++ interface does not provide any additional functions for the Tracker Server. It builds upon the C interface and is made up of one include file, *ES_CPP_API_Def.h* with the *ES_C_API_Def.h* as its basis. The C++ interface implements two classes *CESAPICommand* and *CESAPIReceive*, apart from wrapper classes for each data structure (from C-TPI). *CESAPICommand* handles sending commands from the client application to the TS and *CESAPIReceive* allows easy receiving and parsing of data sent by the TS to the client application.

 The advantage of a class design is the availability of constructors to perform (struct) initialization. A Tracker Server C++ interface is preferable to a C low-level interface, if a C++ compiler is available.

 See the Reference Manual for more information.

Platform Independent

Tracker Server client programming remains platform independent since C++ compilers are available for virtually every platform.

TCP/IP

This chapter does not touch TCP/IP issues. This sample uses the *MSWinsock OCX* for communication, as in Sample 1 & 2.

SendPacket()

The class *CESAPICommand* contains a pure-virtual function *SendPacket()*, which must be overwritten. This approach allows convenient 'Send…' command functions.

 Dealing with C data structures for sending commands is no longer required, as they are completely 'hidden'.

ReceiveData

In order to select the data the application is interested in, *CESAPIReceive* offers a method *ReceiveData*, which is called on data arrival events, as well as numerous virtual member functions.

All class member functions are defined 'inline'. Neither a library nor a .cpp file is required. One single include file suffices. The C++ interface is fully transparent with complete source code provided.

Sample 4

This sample is not a tutorial as the implementation process is essentially the same as Sample1. Sample 4 specific differences will be highlighted.

Sample 4 specifics

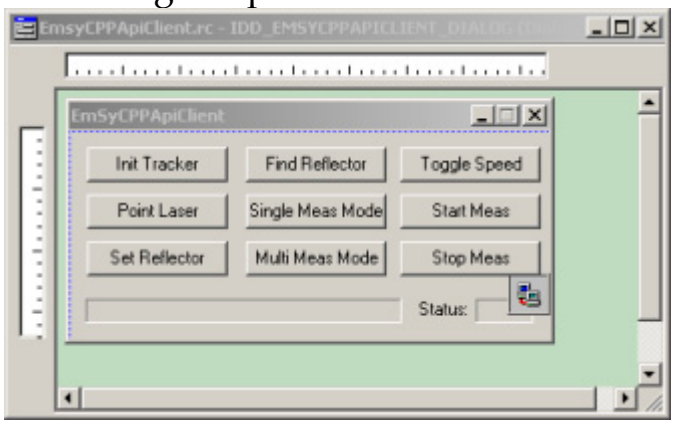
Application Framework

1. Creating an application framework and user-interface
    1. Name the project *EmsyCPPApiClient*.
    2. Do not add explicit *Connect /Disconnect* buttons.
    3. Implement the TCP/IP connection code to the *InitDialog()* function and use the *WM_DESTROY* handler to close the connection.

        Not a key difference. It performs the connection/disconnection 'automatically'.

See *EmsyCPPApiClientDlg.cpp/.h* files for details.

4. More buttons and appropriate handlers are added with the Class Wizard. The dialog template reads as follows:



Class for Commands

2. Deriving a class for sending commands.

Key difference to Sample 1.

1. Create a new class named, for example, *CESCppClientApiCommand*, which is derived from the base class CESAPICommand, and add to project. This gives a new file pair: *EsCppClientApiCommand.h /.cpp*
2. The core of this class declaration reads as follows:

```
#define ES_USE_EMSCON_NAMESPACE
#include "ES_CPP_API_Def.h"

class CMSWinsockControl;

class CESCppClientApiCommand : public EmScon::CESAPICommand
{
public:
    CESCppClientApiCommand();
    virtual ~CESCppClientApiCommand();

   void SetSocketPtr(CMSWinsockControl *pWinSockCtrl) {
      m_pWinSockCtrl = pWinSockCtrl;}

protected:
   CMSWinsockControl* m_pWinSockCtrl;

   COleVariant PackIntoVariant(void *pData, UINT sizeOfData);

   // virtual override
   bool SendPacket(void* pPacketStart, long lPacketSize);
};
```

3. For programming convenience, the class has a member variable pointing to the Winsock control, including an initialization function. The Variant packing helper has been designed as a (protected) member function of this class.

   See "Sample 1-Tutorial" on page 30 for more information.

Winsock Control

3. Based on the Winsock control, the implementation of *SendPacket* reads as follows:

   This function is dependent on *MSWinSockCtrl, SendPacket() function* and connect/disconnect code changes, if some other TCP/IP communication (*Winsock API, CAsyncSocket* or a third party library) is used.

```
bool CESCppClientApiCommand::SendPacket(void* pPacketStart,
                                        long lPacketSize)
{
   try
   {
      COleVariant variantData =
           PackIntoVariant(pPacketStart, lPacketSize);

      if (m_pWinSockCtrl)
         m_pWinSockCtrl->SendData(variantData);
      else
       ASSERT_VALID(m_pWinSockCtrl); // pointer valid?
   }
   catch(...)
   {
      TRACE(_T("MSWinSockCtrl Exception (SendData)\n"));
      return false;
   }

   return true;
}
```

1. In the dialog class *CEmsyCPPApiClientDlg*, declare an instance of type *CESCppClientApiCommand* :

```
private:
    CESCppClientApiCommand m_EsApiCommand;
```

2. The implementation of the command handlers (dialog buttons) reads as follows:

```
void CEmsyCPPApiClientDlg::OnButtonStartmeas()
{
   if (!m_EsApiCommand.StartMeasurement())
      TRACE(_T("StartMeasurement failed\n"));
}
```

3. OR

```
void CEmsyCPPApiClientDlg::OnButtonPointlaser()
{
   if (!m_EsApiCommand.PointLaser(1.342, 2.09, 0.5))
      TRACE(_T("PointLaser failed\n"));
}
```

 Textual Error messages, directed to debug window using TRACE statements, are not sufficient for an end-user application.

C TPI Source code

4. The same function designed with C, with a code reduction of about 80%, instead of C++ reads as follows:

 See "Sample 1-Tutorial" on page 30.

```
void CEmsyCApiClientDlg::OnButtonLaserpointer ()
{
   try
   {
      PointLaserCT data;

      data.packetInfo.packetHeader.type = ES_DT_Command;
      data.packetInfo.packetHeader.lPacketSize =
                                              sizeof(data);
      data.packetInfo.command = ES_C_PointLaser;

      data.dVal1 = 1.342;
      data.dVal2 = 2.09;
      data.dVal3 = 0.5;

      COleVariant vrtData =
               PackIntoVariant(&data, sizeof(data));

      m_winSockCtrl.SendData(vrtData);
   }
   catch(...)
   {
      TRACE(_T("MSWinSockCtrl Error (PointLaser)\n"));
   }
}
```

Receiving Data Sample 4

This Sample requires a client implemented *CDataReceive* class, which implements a limited reception of command-answers, as needed by the Sample. The *CESAPIReceive* class, which is provided in Sample 9, can be used to replace the *CdataReceive,* and thereby discard two application files.

 This replacement is to be treated only as an exercise for users. It is not recommended to implement your own receiver class (as part of the client application).

 See file ES_CPP_API_Def.h, for an example of *CESAPIReceive* class.

 See file DataReceive.h/.cpp, for an example of *CDataReceive* class.

CESAPIReceive class Sample 9

This Sample, *EmsyCPPApiConsoleClient*, with a *CESAPIReceive* class demonstrates Sending and Receiving features of the C++ TPI (among other features). This Sample is a simple console application and has no GUI or Winsock Control (Variants) overhead.

 Functions *OnDataArrivalWinsock1()* and *ProcessData()* can be copied from Sample 1, with minor extensions to the 'switch' statement of *ProcessData()*.

 See also "Receiving Data Sample 9" on page 66 for more information.

CESAPIReceive Class Description

The following describes the class *CESAPIReceive*.

 See file ES_CPP_API_Def.h .

This class is represented by the file pair *DataReceive.cpp/.h*. *CDataReceive* is a native class, i.e. it is not derived from some other base class.

ProcessData() is designed as a member function of the class. It takes the data as it comes from the TCP/IP network, and is the 'switch' statement for arrival data interpretation. Results are not written directly from the switch statement to the user interface. A virtual function is called for each type of arriving data. These virtual functions pass the data through their command type dependent parameter to the calling function.

 The interpretation of incoming data in this switch statement is implemented only as far as needed for this sample, which is not the case for *CESAPIReceive*.

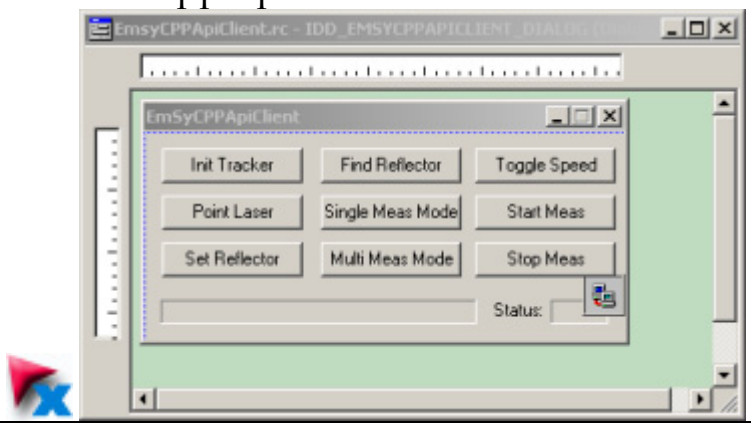
Procedure

1. Derive a class from *CdataReceive*.
2. Call the *ProcessData()* function of the class derived, where the data comes in from the network, only at one location in the *OnDataArival* notification handler.
   The derived class must implement only virtual functions for answer packet types the client application is expecting. Unrequired

data will just be ignored and the virtual function of *CDataReceive* will be called – with no effect. These functions are 'empty', apart from a Trace statement for developing purposes.

3. Using a CDataReceive class (CESAPIReceive class) allows hiding the awkward data arrival 'switch' statement from the main code. In other words: The arrival data parsing-function becomes a member function of CESAPIReceive, and is, therefore, completely hidden from the client applications.
4. This sample does not exclusively derive a class from *CDataReceive*. The dialog class inherits from *CDialog* and *CDataReceive* (multiple inheritance**).** This is appropriate because the dialog class comprises the *OnDataArrivalWinsock1()* arrival data notification handler, which has to call *ProcessData()*.

See "Class for Commands" on page 58 for the appropriate declaration.




```
class CEmsyCPPApiClientDlg: public CDialog,
                            public CDataReceive
```

5. The dialog class must, therefore, implement the *CDataReceive* virtual function overrides.

Single Point Measure Data Sample

```
void CEmsyCPPApiClientDlg::OnSingleMeasurementAnswer(
                                    SingleMeasResultT *pSingleMeas)
{
   CString s, sStatus;

   // Do something with the data (format it)
   s.Format(_T("X=%.6lf, Y=%.6lf, Z=%.6lf\n"),
            pSingleMeas->dVal1,
            pSingleMeas->dVal2,
            pSingleMeas->dVal3);

   m_staticResult.SetWindowText(s); // then display it

   // also display return status
   sStatus.Format(_T("%d"),
                  pSingleMeas->packetInfo.status);

   m_staticStatus.SetWindowText(sStatus);

   if (pSingleMeas->packetInfo.status != ES_RS_AllOK)
      Beep(100, 100); // command failed
}
```

6. The data is delivered through a pointer to a *SingleMeasResultT struct* function parameter. This function neither needs to perform any *DataRead*, nor any *ProcessData* (interpretation).

    It solely uses the data (just display it).

7. This function uses a pointer argument while the *OnCommandAnswer()* function has been designed to take a 'reference' argument (see comment in code). This constellation is used to demonstrate the two possibilities. The actual class should consistently use one or the other method.

 References should be preferred to pointers whenever possible in C++ programming.

 See the many comments in the sample source code.

Remarks

- Change the IP address and hard-coded numbers (coordinates for *PointLaser* command) in the sample code as required.
- Replace the Winsock control by some other TCP/IP communication API, as an exercise.

- If the Winsock 2.0 API is used, it is recommended to use the *WSAAsyncSelect()* function rather than the *WSAEventSelect()* for data receiving.

  In a Windows framework, using messages is simpler than using events and threads. Use of *MFC CAsyncSocket* on the other hand requires no such consideration – *CAsyncSocket* provides a message handler for data receiving by default.

- The function *Laser Pointer* toggles the location of the laser beam between two (hard-coded) positions. This is not of real practical use.
- The 'Toggle Speed', toggles the continuous measurement speed from 1/second to 100/second. To change speed during a continuous measurement, the measurement must be stopped before the speed is reset.

  The *Stop Meas* command is not valid in single measurement mode.

GUI Design

- Inactive buttons are not grayed out. The user must wait for arrival of the answer from the previous command, before sending a new command.
- Radio buttons or list boxes are recommended, to implement *Multi Meas Mode* (continuous) and *Single Meas mode* (stationary) mode, rather than simple push buttons.
- The parameters for *Pointer Laser* need to be entered in fields, in a dialog. The present design has not considered this aspect.

- Before leaving the application, make sure that the TS is set to *Stationary Measurements* (Single point measurement). Failure to do so may have unexpected effects, upon starting another client.

  Reboot the Tracker Server in such cases.

Receiving Data Sample 9

This Sample, an EmsyCPPApiConsole client with the *CESAPIReceive* class in the C++ TPI, implements a 'safe' data-reading thread (in order to handle 'clustered' and/or 'scattered' answer packets correctly, in case of a data 'traffic jam').

See explanations in "Sample 3" on page 52, on the multithreaded console application based on WinSocket API.

See also explanations in "Sample 4" on page 57 on using a Receiver class similar to CESAPIReceive.

This Sample is a minimal, 'single-source' file and easy to understand, in spite of being a multithreaded application.

Set the IP address to the actual TS address, before building the application.

ActiveX Component Sample 12

This *ReflectorCtl* sample provides an ActiveX component comprising the most common reflector commands.

This control skips building up a lookup table for ID/Name mapping, querying all the defined reflectors from the system and providing the appropriate user interface controls.

The Sample contains full source code (Visual C++) and has a compiled component *Reflector.ocx*, which allows use without a Visual C++ compiler.

 No support for this ActiveX component is provided.

Remarks

- The *Reflector.ocx* control must be registered before it can be used.
- Only one instance of such a control can be instantiated per Form/Dialog box.
- The properties 'ServerAddress' and 'PortNumber' can be specified at (Form/Dialog) design time. However, this only makes sense if these parameters are constant. The more common way is to set these properties programmatically.
- Call the method *Initialize* after having set the properties and not before the client application has successfully connected to the same address/port. This lets the client application, instead of the *Reflector.ocx*, handle any connecting problems.
- The client application must ignore answers from commands triggered by the *Reflector.ocx* (*Get Reflectors*, *GetReflector* and *SetReflector*).
- Do not implement an Error Event handler for *Reflector.ocx*. The control has a built- in handler. Visual Basic does not allow it– it causes a compiler error. If correctly applied, the component should never fire an error event.

- Here is a code sequence for a VB application. Typically executed in Form Load:

```
    Reflector1.ServerAddress = "193.8.34.213"
    Reflector1.PortNumber = 700
Reflector1.Initialize
```

- It is assumed that the client application has already successfully connected to the same address/port before these calls.

Keyboard Interface Limitation

- This component is primarily designed for mouse control and does not work properly with a keyboard interface (E.g. use of arrow keys in VB).

 See VC/VBA/VB documentation for general information on ActiveX controls, and how to include them in applications.

Sample 10

This Sample ('keasytracker') is an EmScon client application developed on SUSE 7.2 Linux (KDE) by a third- party provider. It has been published according to GNU General Public License (GPL).

 For further details see ‘README’ file in the Sample 10 project folder.

4.COM Interface

High-level Interface

Introduction

Unlike the C and C++ TPI, the COM TPI is a DLL library and not an include file. This DLL provides an easy to use programming interface for the Tracker Server. This makes it suitable for programmers with minimal programming expertise to design simple tracker applications.

The interface consists of a COM object. It is designed as an *ATL DLL COM* server and a *LTControl.dll,* as part of the tracker server SDK, with a built-in TCP/IP communication. The LTControl COM-object DLL is based on the tracker server C++ TPI, the Win32 Sockets 2.0 API and VC++ ATL. The *LTControl.dll* is, in a sense, a tracker server C++ client, allowing design of such a control.

 The programmer is not required to deal with TCP/IP communication libraries or system programming interfaces.

The high-level TPI supports both synchronous and asynchronous methods.

COM objects expose 'interfaces', described by a Type-Library, which is implicitly included in the DLL. A pure Type Library *LTControl.tlb* is also available, although not really needed. This High-level interface does not provide any additional functions (in terms of Tracker Server controlling

functions). *LTControl* is strictly based on the C++-TPI, with a high-level, convenient programming interface.

 See chapter 'COM Interface' in the Reference Manual TPI, for more information on the interfaces provided.

 COM interfaces work well together with Visual Basic and other programming languages on the Win32 platform, unlike the low-level interface.

 See "Sample 2-Tutorial" on page 46.

COM vs. Low-Level Programming

Advantages	Disadvantages
No include-file to deal with, therefore no translation required of C-structs and enums to VB syntax.	A DLL (ATL COM component). Its source code is not public
No TCP/IP library or function needs to be provided. All these functions are built-in. Only the IP address of the tracker server needs to be provided.	Is limited to Win32 platforms.
The high-level interface offers both synchronous and asynchronous communication support.	Due to the COM interface, the performance may be affected.
There are wide varieties of notification methods for arrival data when using asynchronous communication.	Since TCP/IP communication is built-in, there are no 'tuning' possibilities.
Supports various programming languages. Easy to use due to support of 'IntelliSense' for Microsoft Visual and Office programming tools.	The component needs to be registered on the client PC.

Interfaces and Notification Methods

All interfaces of the LTControl, including their methods and properties, are listed in the Reference Manual.

 See chapter 'COM Interface' in the Reference Manual, for more information on the interfaces provided.

LTControl COM Viewer:

In order to get detailed information about the Interfaces (including data types, properties, methods and events) exposed by a COM object, a COM viewer may be used. Visual Studio offers such a viewer: The OLE/COM Object Viewer can be launched from the *Tools* menu of VC++.

File > View Type Lib > LTControl.dll or LTControl.tlb.

Registering COM Objects

COM objects must be registered on the application PC before they can be used.

LTControl.dll Installation

1. Register *LTControl.dll* on the client PC (both developer and customer PCs).
2. If *LTControl.dll* is located in the *C:\WINNT\system32* directory, call *Regsvr32 C:\WINNT\system32\LTControl.dll* from the *Start/Run* menu of the explorer task-bar.

   
   The *LTControl.dll* does not depend on any other custom DLL, it can be registered anywhere. The Windows system directory is the common location.
3. A message box appears confirming registration – 'Registering of LTControl.dll succeeded'.
4. A message such as: Error 'Load Library failed, error 0x0000007e' most likely indicates that the PC lacks a correct ATL.dll installation (missing, wrong version or not registered).
   In this case, first install ATL.dll as described below. After that, repeat registering of LTControl.dll.

ATL.dll Installation

1. Copy *Atl.dll* from ES SDK 'Lib' directory to Windows system/system32 directory **OR** to *LTControl.dll* directory.

Unicode version for WinNT/Win2000. ANSI version for Win9x/Win ME.

 See properties of ATL.dll for operating systems supported.

2. Register *Atl.dll* – Regsvr32.exe <path>\Atl.dll.
3. Repeat registration of LTControl.dll.

Visual Basic client

Sample 5 Tutorial

The use of the LT Control provides the right tool to build a VB Tracker Server client, *LtcVBClient*. In contrast to the VB application in Sample 2 with the Tracker Server low-level interface, which was not recommended.

 For the application framework and for initializing COM objects: The LTControl.dll must be correctly registered before proceeding.

 See "LTControl.dll Installation" on page 72 for details.

1. Launch Visual Basic 6.0, choose from menu *File* > *New Project* > *Standard exe*. Click *OK*.
2. Save the form as *LtcVBClient.frm* and the project as *LtcVBClient.vbp*.
3. Choose menu *Project* > *References*.
4. In the dialog list box, check the entry *LTControl 1.2 Type Library*. Click *OK*.

    Ensure file path at the bottom of the dialog matches the control's registration location, browse for the correct location, using the 'References' dialog.

ATL Type COM object

The *LTControl.dll* is not an ActiveX (OCX) control. It is a general ATL type COM object, which can

be used in non-window based applications. It will also support, for example, pure C-clients (console applications).

 It is not possible to place an *LTControl* instance to the VB Form (as *MSWinsck.ocx* requires it).

Differences between LTConnect & ILTConnect

Differences between object *LTConnect* and the related interface *ILTConnect*.

- In VB clients, interfaces are not dealt with directly as objects are.
- The keyword *New* in the declaration of *LTConnect* creates this object within the client application.
- Local or remote object creation, in the COM server, depends on the design of a particular COM object.
- The *LTConnect* object needs to be created with *New*.

 Refer to a COM book for further details.

Accessing Interfaces

1. To access its interfaces, an object variable of type *LTConnect* is needed in the declaration part of the VB application:

```
Dim ObjConnect As New LTConnect
```

2. Declare an object for each one of the remaining types:

```
Dim WithEvents ObjAsync As LTCommandAsync
Dim WithEvents ObjSync As LTCommandSync
```

3. Provide either a synchronous or an asynchronous interface.

    Declaring both, as done here for demonstration purposes, will result in some duplicate data arrivals.

4. The two objects *ObjSync* and *ObjAsync* cannot be created with *New* (the VB compiler does not even allow this).

    This is also by design of the COM component, where these two objects have been designed as *Noncreatable* - the object instances are created in the LT Control's control scope and not locally in the application.

5. The keyword *WithEvents* makes the application recognize event notifications.

LTCommand Objects

An *LTConnect* object is always required, whereas only one of the *LTCommandSync* or *LTCommandAsync* objects is required. Depending on the selected notification mechanism, *LTCommandAsync* or *LTCommandSync* is to be declared with/without event support (*WithEvents* keyword).

1. The non-createable *LTCommandSync* and *LTCommandAsync* act like 'pointers'. The 'pointers' are initialized with the properties of *LTConnect* .
2. In the *Form_Load* function, connect to Tracker Server and select the desired notification method.
3. If *ConnectEmbeddedSystem* has succeeded (no exception thrown), *an LTCommand object has* been created.
4. Initialize the pointers by calling:

```
Set ObjSync = ObjConnect.ILTCommandSync
Set ObjAsync = ObjConnect.ILTCommandAsync
```

5. Do not reference ObjConnect.ILTCommandSync **OR** ObjConnect.ILTCommandAsync before the ObjConnectConnectEmbeddedSystem call.

6. The COM methods throw exceptions in case of failure. The Form_Load() subroutine shown below also shows how to handle them (try/catch for Visual Basic).
7. Add an error handler as shown below in every handler function that deals with the COM objects.

    Do not use a global error handler. Unhandled exceptions will lead to program abort.
8. The initial code for every Tracker Server Visual Basic client has to be as shown below, except the SelectNotificationMethod.
9. Use the Form_Load subroutine as 'template code' for other VB TPI clients.
10. Set the IP address of the Tracker Server.

```
Private Sub Form_Load()
    On Error GoTo ErrorHandler

    ObjConnect.ConnectEmbeddedSystem "127.34.8.161", 700

    ' This call may have different parameters in another project
    ObjConnect.SelectNotificationMethod LTC_NM_Event, 0, 0

    ' NEVER FORGET THIS! Note: In real applications there is
    ' usually only one of these:
    Set ObjSync = ObjConnect.ILTCommandSync
    Set ObjAsync = ObjConnect.ILTCommandAsync

    Exit Sub
ErrorHandler:
    MsgBox (Err.Description)

    End ' Exit application when connection fails
End Sub
```

 The *End* statement in the error case exits the application, when connection to the tracker server has failed.

11. Disconnect the Tracker Server in *Form_Unload()*. Implement the following handler:

```
Private Sub Form_Unload(Cancel As Integer)
    ObjConnect.DisconnectEmbeddedSystem
End Sub
```

 See chapter 'COM Interface' in the Reference Manual for an explanation of the function *ObjConnect.SelectNotificationMethod LTC_NM_Event, 0, 0*.

Synchronous/Asynchronous Interface

Differences between the synchronous and asynchronous interface.

- The functions of the synchronous interface do not return before the task is completed, while the asynchronous functions do so (see C/C++-TPI).
- In general, programming with synchronous functions is much easier. Handling *Data-Arrival Events* or *Notifications* is not required (except in some special cases).
- Use of either synchronous or asynchronous objects depends on the application.

   Use of both is not recommended.
- Running the application in the current state implicitly connects and initializes the tracker server upon *Form Load* and disconnects upon *Form Unload*.

Implementing Synchronous Commands

It is presumed that how to add buttons and their related command handler 'skeletons' are known.

 See "Sample 2-Tutorial" on page 46.

1. Add a button named *InitSync* (*caption Init Tracker (sync)*). The command handler should be completed with the following code:

```
Private Sub InitSync_Click()
    On Error GoTo ErrorHandler

    ObjSync.Initialize

    Exit Sub
  ErrorHandler:
    MsgBox (Err.Description)
End Sub
```

2. Since this is a synchronous call:
   - *ObjSync.Initialize* will not return before the tracker has finished initializing.
   - The *Exit Sub* statement will not be reached until initialization is finished. A real application would at least display an hourglass cursor while the program resides in the *InitSync* function.
   - The error handler is implemented in every single handler, otherwise the application will terminate in case of an error (unhanded exception).
3. Saving the *pointer* variables *ObjSync* and *ObjAsync*, and making direct calls such as *ObjConnect.ILTCommandSync initialize* will not work with the VB compiler because *ObjSync/ObjAsync* are not real pointers.

    The *Set ObjSync = ObjConnect.ILTCommandSync* statement is a *QueryInterface.*
4. Add another Button/Handler *Measure Single Point* and implement the handler as shown below. It is presumed the tracker server is set to 'stationary' when triggering this command, and the laser beam is attached to a reflector. The result – as a synchronous answer – can be shown directly in a message box (only x, y and z are shown).

```
Private Sub StartMeas_Click()
    Dim x As Double
    Dim y As Double
    Dim z As Double
    Dim d As Double 'd is a dummy variable
    Dim b As Boolean

    On Error GoTo ErrorHandler

    ObjSync.MeasureStationaryPoint x, y, z, d, d, _
                                   d, d, d, d, d, d, _
                                   d, d, d, d, d, d, b

    MsgBox (x & CStr(" , ") & y & CStr(" , ") & z)

    Exit Sub
ErrorHandler:
    MsgBox (Err.Description)
End Sub
```

 If this command was an asynchronous call, it would not be possible to display the result within this function. A result display is performed in the appropriate asynchronous answer handler.

 For other calls, refer to Sample 5 source code.

Implementing Asynchronous Commands

Visual Basic with 'IntelliSense' provides support for the available functions of an interface with the function parameters.

1. Add a button named *InitAsync* (*caption Init Tracker (async)*). The command handler should be completed with the following code:

```
Private Sub InitAsync_Click()
    On Error GoTo ErrorHandler

    ObjAsync.Initialize

    Exit Sub
ErrorHandler:
    MsgBox (Err.Description)
End Sub
```

In contrast to the synchronous initialize function, this one does not stop at the *Initialize()* function, *Exit Sub* is reached immediately. When tracker initialization is done, a notification or event is sent.

Catching Events and Messages

2. For asynchronous commands, the answers must be handled by some event mechanism. This could be *Events*, *Windows Messages*

(custom window-bound, registered, WM_COPYDATA).
For Visual Basic, *Events* are the right choice. The event mechanism is provided by the *_ILTCommandAsyncEvents* interface, which is a subsidiary of *ILTCommandAsync*. To activate this mechanism for a Visual Basic application, provide the keyword *WithEvents* upon the declaration:
When no requirements for catching events exists, omit the *WithEvents* keywords in order to save overhead.

```
Dim WithEvents ObjAsync As LTCommandAsync
```

3. When no requirement for catching events exists, omit the *WithEvents* keywords in order to save overhead.
4. When no requirement for catching events exists, omit the *WithEvents* keywords in order to save overhead.

The synchronous interface has an event interface, *_ILTCommandSyncEvents*. It is used for continuous measurements and (unsolicited) error messages, which cannot be handled synchronous by their nature.

Events are one of the notification methods of the LT Control. When using Windows messages for asynchronous notifications the keyword *WithEvents* is invalid. Windows messages are appropriate for VC++ clients and will be discussed later.

5. The application must declare what notification mechanism to use. Do this with the statement shown below. Without calling this function in the initialization part of the

application, no notification mechanism will be activated (when dealing with synchronous commands exclusively)

 See remarks on continuous measurement in “Handling Data Arrival – Continuous Measurements” on page 86.

```
ObjConnect.SelectNotificationMethod LTC_NM_Event, 0, 0
```

6. As soon as the *WithEvents* keyword is declared, the *ObjAsync* object (or whatever the variable is called) is listed in the top left list box of the *Form's* source code window.

    Remove *WithEvents* and save the code – the list entry will vanish.

7. If *ObjAsync* is selected in the list box, a list of all available event handlers is shown in the right drop-down list.

8. Select to generate the handler framework. Selecting *ErrorEvent* will generate a function named *ObjAsync_ErrorEvent*.

9. Complete this function with a message box to read as follows:

```
Private Sub ObjSync_ErrorEvent( _
          ByVal command As LTCONTROLLib.ES_Command, _
          ByVal status As LTCONTROLLib.ES_ResultStatus)
    MsgBox (command & CStr(" , ") & status)
End Sub
```

1. This event handler will now be called, for example on a Beam Broken Event.

Extended Synchronous Functions

ObjSync.MeasurStationaryPoint has 18 (basic data type) parameters. Basic data type parameters are a requirement in order to use these functions from VBA (Excel, Access…).

 For programming languages supporting user-defined data types (VC++, Visual Basic), having a function with only one struct parameter

would be more convenient. LTControl offers a collection of such 'extended' functions.

 See Reference Manual for details.

One of these functions is implemented in the sample, *MeasureStationaryPointEx*:

```
Private Sub StartMeasEx_Click()
    Dim result As SingleMeasResultT

    On Error GoTo ErrorHandler

    ObjSync. MeasureStationaryPointEx result

    ' display the result
    MsgBox(result.packetInfo.status & CStr(" , ") & _
              result.packetInfo.packetHeader.Type & _
              CStr(" , ") & result.dVal1 & CStr(" , ") & _
              result.dVal2 & CStr(" , ") & result.dVal3)

    Exit Sub
ErrorHandler:
    MsgBox (Err.Description)
End Sub
```

The data type *SingleMeasResultT* from the C-TPI is transparent through the COM interface. The VB application knows this type, through its reference to the *LTControl.*

Remark

Do not test explicitly against the VB keyword 'True', if using the *Get<FunctionName>Ex* methods of the LTControl, for those commands returning Boolean data within their result structure. This is because the Boolean member in these structures – if true – are (1). However, the VB keyword 'True' evaluates to (-1).

Always test the variable directly, or against 'Not False'.

Example

```
ObjSync.GetContinuousDistanceModeParamsEx dataout

If (dataout.bUseRegion) Then
    MsgBox "bUseRegion is True"
End If

or

If Not (dataout.bUseRegion = False) Then
    MsgBox "bUseRegion is True"
End If

are both correct. However, the following would evaluate to a
wrong result:

If (dataout.bUseRegion = True) Then
    MsgBox "bUseRegion is True" ' No message even flag true!
End If
```

C++ Console Application

Sample 6

This is only a test/demo application, made up of 'one source file/function', it should be easy to understand without further comments.

- The *LtcConsoleClient* provides a minimal Tracker Server application based on the *LTControl* COM component.
- It is designed as a so-called Console Application and consists of only about 15 lines of code, to direct the tracker to a certain point and measure it.
- Set the hard-coded values (IP address, coordinates) for the Tracker Server.

Synchronous Calls

This type of application only handles pure synchronous calls. Receiving Windows messages is not possible for a console application. Implementing an event sink is possible , but requires some advanced programming knowledge.

Include files from the C or C++ TPI are not required. TPI information is provided by importing LTControl.tlb.

C++ Windows-MFC Application

Sample 7

The *LtcCPPClient* provides a dialog- based MFC C++ application. It uses the synchronous interface, but also implements an event- sink to catch asynchronous answers (continuous measurements and error events).

ATL/COM

Programmers need to be familiar with ATL/COM in order to understand the event sink implementation.

Refer to a COM book for further details.

The *LtcCPPClient* covers all essential initial steps for a successful system start and accurate results, with some disabled code, which demonstrates all other variants of notification methods, which may be more familiar to programmers than event handling.

See comments in source code.

Message Notifications

The disadvantages of message notifications are:

- The result parameters cannot be received directly.
- There are only general messages for all types of answers.
- Usually only the size of a data block is passed with the message.
- The data block must be first read with *GetData()* (except for WM_COPYDATA) and then interpreted. Interpretation is done with a 'switch' statement with the *ProcessData()* sample code.

See “Handling Data Arrival – Continuous Measurements” on page 86.

This sample also shows one of the features not shown so far: How to retrieve the reflectors known to the system. It also demonstrates continuous measurements.

 View the source code for details. Note that this code contains a relatively big overhead needed for user interface issues. The Tracker Server specific part is not that dominant.

Source Code Description

- Information that is displayed in list boxes, such as units, CS type, is automatically read from the Tracker Server upon startup. What is seen has been actually selected.
- Changing the items of one list box automatically creates a 'Set' for the newly selected item.
- On changing units, CS type etc., some dependent information may vanish from the related edit fields to ensure consistency. This is due to the paradigm 'What you see is selected'. Do a 'Get' to recover it, which can also be done by the application.
- On setting new values, the 'Set' command is automatically followed by a 'Get' (two beep sounds). The 'Get' is not required (only for testing and demonstration purpose).
- Reflectors are read upon client startup. Can be heard by characteristic beeps. They must be selected in the reflectors list box.

   The *GetReflectors* button is only required in 'emergency' cases. If the client starts before the Tracker Server is ready and the client dialog shows up, but is not able to read the reflectors yet.

- The application is based on *LTC_NM_Event* notification selection. By changing the parameter of *SelectNotificationMethod* in *CCPPClientDlg::OnInitDialog()* (all variants are prepared), a different notification method can be activated. However, there is only an incomplete implementation of *ProcessData()* for these alternate methods (reflector processing, for example, is not yet complete).
- Only the *LTC_NM_Event* notification method is fully implemented in this sample. However, data transfer works with message methods. One or the other methods can be activated for test reasons (a good exercise would be to complete the missing implementation).

   Only the last call of *SelectNotificationMethod* is effective (there should be only one call to this function).

   See "Handling Data Arrival – Continuous Measurements" on page 86 for details on obtaining data in general and continuous measurements in particular.

Handling Data Arrival – Continuous Measurements

Continuous measurement streams are always handled asynchronous. That is, even if only a *LTCommandSync* is implemented (through which the *Start Measurement* command may be invoked), the continuous measurement packets will arrive asynchronously.

 A continuous measurement may last very long. It is not suitable to block execution all the time.

Methods to Catch Packets

- Provide a *LTCommandSync* object with a call to *SelectNotificationMethod*, with

LTC_NM_Event as first parameter.
This setting allows catching the continuous measurement packets through the event mechanism. This is especially convenient for Visual Basic.

- Use one of the Windows Messages notification methods.

   See "Sample 7" on page 84, as shown (disabled) in the code.
  These may be methods preferred with VC++ clients, especially if the programmer is not familiar on setting up event sinks. On the other hand, receiving Windows messages within VB application is permissible.

- The *MultiMeasResultT* structure only covers the first item of the array. The rest of the *lNumberOfResults - 1*-array elements are padded to the packet without gaps.

   Continuous measurement packets mostly contain more than one measurement value. Iteration through an array of measurements is necessary.

- A code fragment, on how to process a continuous measurement packet using the event mechanism, is shown below. This is a client implementation, stripped down and altered from sample 7, of the *ContinuousPointMeasDataReady* event, which exists for both *_ILTCommandSyncEvents* and *_ILTCommandAsyncEvents* interfaces

```
void __stdcall OnContinuousPointDataReady(long resultsTotal,
                                          long bytesTotal)
{
   CString s;
   VARIANT vt;
   VariantInit(&vt);

   if (m_pLTConnect == NULL)
     return;

   m_pLTConnect->GetData(&vt);

   MultiMeasResultT *pData =
               (MultiMeasResultT *)vt.parray->pvData;

   ASSERT(pData->lNumberOfResults == resultsTotal);

   for (int i = 0; i < pData->lNumberOfResults; i++)
   {
      s.Format(_T(" %.7lf, %.7lf, %.7lf"),
         pData->data[i].dVal1,
         pData->data[i].dVal2,
         pData->data[i].dVal3);

      // this is application dependent. May differ in your app
      m_pMainWnd->m_edit_Result.SetWindowText(s);
   } // for
} // OnContinuousPointMeasDataReady()
```

- On using a Windows message notification method, *LTC_NM_WM_Notify*, it looks quite similar. However, with the event method there is a unique event function for just receiving continuous results. With message notify methods, all types of data packets come in through the the same message handler. The data must be interpreted with a 'switch' statement. This is done in the *ProcessData()* function.

   Use of the *CESAPIReceive* class of the C++ interface is another possibility.

- The following implementation demonstrates receiving, not only data of continuous measurements, but also, any kind of data.

```
LRESULT CCPPClientDlg::OnNotifyMsg(WPARAM wParam, LPARAM lParam)
{
   CString s;
   VARIANT vt;
   VariantInit(&vt);
   m_pLTConnect->GetData(&vt);

   // wParam = msg ID = cookie!
   ProcessData(vt.parray->pvData, wParam);

   return true; // return non-zero if msg handled
}
```

- Activating this function calls *SelectNotificationMethod()* with the following parameters:

```
// cookie must be in the valid range for a user defined message
m_pLTConnect->SelectNotificationMethod(LTC_NM_WM_Notify,
                                       (long)m_hWnd,
                                        MY_NOTIFY_MSG);
```

- The message ID (which also acts as a cookie here) is defined as:

```
#define MY_NOTIFY_MSG (WM_USER+99)
```

- Entry in the message map must exist as follows:

```
ON_MESSAGE(MY_NOTIFY_MSG, OnNotifyMsg)
```

- Provide the *ProcessData()* subroutine.

   Not every type of data packet is fully implemented:

```
void CCPPClientDlg::ProcessData(void *ptr, int nCookie)
{
   CString s, s2;

   PacketHeaderT *pHeader = (PacketHeaderT*)ptr;

   switch (pHeader->type)
   {
      case ES_DT_MultiMeasResult: // most frequent ones on top
      {
         MultiMeasResultT *pData = (MultiMeasResultT *)ptr;

         for (int i = 0; i < pData->lNumberOfResults; i++)
         {
            s.Format(_T("%lf, %lf, %lf"),
                     pData->data[i].dVal1,
                     pData->data[i].dVal2,
                     pData->data[i].dVal3);

            // do something with data
          // application dependent
            m_staticContMeas.SetWindowText(s);
         } // for
      }
      break;

      case ES_DT_Error:
      {
         ErrorResponseT *pCmdData = (ErrorResponseT *)ptr;

    s.Format(_T("error: command=%d, status=%d\n"),
                  pCmdData->command,
                  pCmdData->status);

         AfxMessageBox(s);
      }
     break;

      case ES_DT_SingleMeasResult:
      {
         SingleMeasResultT *pData = (SingleMeasResultT *)ptr;
         ASSERT(pData->measMode == ES_MM_Stationary);

        // TODO: do something with data
      }
      break;

      case ES_DT_ReflectorPosResult:
      {
        // Not implemented
      }
     break;

      case ES_DT_Command:
         break; // nothing to do

      default:
         Beep(100, 100); // all other data currently unhandled
   } // switch
} // ProcessData()
```

For further details refer to the sample source code.

Limitations for high frequency continuous measurements (like loss of data) may occur due to hardware (LAN, PC performance) limitations. Tests have shown that under good conditions

(LAN, PC, Client program design), the LT Control is able to handle the maximum data rate of 1000 points per second, even through the event notification mechanism, which might have slightly less performance than the message methods – Low performance of IDispatch Interfaces.

Known Bugs in ATL Event Sink Implementation

There are currently two known bugs confirmed by Microsoft in VC++ 6.0 concerning event handlers.

- (Q237771): Events Fail in ATL Containers when Enum Used as Event Parameter.
- (Q241810) *IDispEventImpl* Event Handlers May Give Strange Values.

Apply one of the workarounds provided in MSDN and in Sample 7 (*file DataArrived.h*) for a practical application of one of the workarounds provided.

Queues and Scattered Data

When the Tracker Server delivers more data through the TCP/IP network than the client is able to process, it results in 'traffic jams'. Although, the TCP/IP network buffers such data (up to the configured buffer size), single data packets will be queued. That is, there are no more 'gaps' between the data packets. When the client is notified from the TCP/IP communication framework that data has arrived, it has to react to this notification by a *Read* call (depending on your communication tools, this can be *recv*, *GetData*, *CAsyncSocket::Receive() etc.*).

These read functions are not able to recognize packet boundaries. Read functions read all data that is currently available (In practice, the data will be read in one read- cycle, limited to a certain buffer size).

 These might be several combined packets or only a fraction of a (trailing) packet.

Problem Solution

1. Provide a sufficient read-buffer and read all that is currently pending. The client application parses the data block into packets, using the header information and size of each packet. With a fragmented last packet, the next read- cycle is started and the two fragments from the previous and the current reading are assembled together. This is probably the most efficient method, since it minimizes the number of reading interrupts. However, it is also the most complex one in terms of data parsing.

- Read only the header to determine the size of the first pending packet. The rest of the packet is estimated by reading (*packetSize* – *headerSize*) bytes.
  Variant method: 'Peek' (instead of Read) the header, without removing data from the socket. With known size, read as many bytes as indicated by *packetSize*. See code sample below.
- The sample code demonstrates a method to ensure complete packets (if data blocks arrive scattered) and to avoid data congestion (traffic jams). It is based on Winsock 2.0 API functions:

```
LRESULT CMsgSink::OnMessageReceived(UINT uMsg, WPARAM wParam,
                                    LPARAM lParam,
                                    BOOL& bHandled)
{
   // The read-buffer is kept static for performance reasons.
   // In a real application better make it a member
   // variable of CMsgSink
   //
   static char szRecvBuf[RECV_BUFFER_SIZE];

   bool bOK = true;
   long lReady = 0;
   int  nCounter = 0;
   long lMissing = 0;
   long lBytesRead = 0;
   long lBytesReadTotal = 0;
   int nHeaderSize = sizeof(PacketHeaderT);
   PacketHeaderT *pHeader = NULL;

   ATLTRACE(_T("CMsgSink::OnMessageReceived(%lu, %lu)\n"),
            wParam, lParam);

   if (WSAGETSELECTEVENT(lParam) == FD_READ)
   {
      // Just peek the header, do not remove data from queue
      lReady = recv((SOCKET)wParam, szRecvBuf,
                    nHeaderSize, MSG_PEEK);

      if (lReady < nHeaderSize)
      {
         if (lReady == SOCKET_ERROR)
         {
            if (WSAGetLastError() == WSAEWOULDBLOCK)
               Sleep(50); // busy - try later
            else
            {
               Beep(1000, 100);
               // not able to get header
            } // else
         } // if

         return true; // non-fatal only a peek, try next time!
      } // if

      pHeader = (PacketHeaderT*)szRecvBuf;

      bOK = bOK && lReady == nHeaderSize &&
             pHeader->lPacketSize >= nHeaderSize &&
             pHeader->lPacketSize < RECV_BUFFER_SIZE &&
             pHeader->type >= ES_DT_Command &&
             pHeader->type <= ES_DT_ReflectorPosResult;

      if (bOK)
      {
         do
         {
            nCounter++;

            if (lBytesRead > 0)
               lBytesReadTotal += lBytesRead;

            lMissing = pHeader->lPacketSize - lBytesReadTotal;

            lBytesRead = recv((SOCKET)wParam,
                              (szRecvBuf + lBytesReadTotal),
                              lMissing, 0);

            if (lBytesRead == SOCKET_ERROR)
            {
               if (WSAGetLastError() == WSAEWOULDBLOCK)
               {
                  Sleep(50); // busy - try later
                  continue;
               }
               else
                  Beep(1000, 100);
```

```
            } // if

            if (nCounter > 64) // emergency exit
            {
               if (lBytesReadTotal <= 0)
               {
                  ATLTRACE(_T("not able to read data
(recv)\n"));
                  return true; // nothing read, can leave safely
               } // if
               else
               {
                  bOK = false;
                  break;
               }
            } // if

            ATLTRACE(_T("Loop: BytesRead %ld, BytesReadTotal \
                      %ld, PacketSize %ld, Missing = %ld\n"),
                     lBytesRead, lBytesReadTotal+lBytesRead,
                     pHeader->lPacketSize,
                     lMissing - lBytesRead);

         } while (lBytesRead < lMissing);

         if (lBytesRead > 0)
            lBytesReadTotal += lBytesRead;
      } // if

      bOK = bOK && lBytesRead == lMissing &&
                   lBytesReadTotal <= RECV_BUFFER_SIZE;

      if (bOK)
      {
         // ProcessReceivedData() is assumed to take one single
         // (complete) data packet. It contains a 'switch'
         // statement to evaluate the packet (we have seen this
         // method several times in this manual / samples)

         if (lBytesReadTotal == pHeader->lPacketSize)
             ProcessReceivedData(szRecvBuf, lBytesReadTotal);
      } // if
   }
   else
      bOK = false;

   if (!bOK)
   {
      // make sure socket is cleaned up on data jam
      // in order to recover ordinary data receiving

      do
      {
         nCounter++;
         lBytesRead = recv((SOCKET)wParam, szRecvBuf,
                           RECV_BUFFER_SIZE, 0);

         ATLTRACE(_T("Recover in loop\n"));

      } while (lBytesRead > 0 && nCounter < 128);

      ATLTRACE(_T("Unexpected data - fatal error\n"));

      Beep(250, 10); // data lost
   } // else

   return bOK; // true when message handled
} // OnMessageReceived()
```

This code ensures that only complete packets are processed. However, the client may still not be fast enough to process all the incoming data. The

TCP/IP framework will buffer data, up to a limit. If such limits are reached, arbitrary data may arrive. The above function has (limited) recovery ability in case this should happen. Data will be lost in such situations.

Cause of Data Loss

- The network is not fast enough.
- The client PC is not powerful enough.
- The application is not able to process data fast enough.
- The application is not designed appropriately.

 The client application can still buffer incoming data, for example, in a FIFO list (taking the data packets as list elements). This approach can be chosen if the performance constraint is caused by intensive data processing. The Winsock API offers certain 'tuning' functions. These allow, for example, to alter internal network buffers. Increasing the receive- buffer with *setsockopt()*, for example, may increase data throughput significantly.

```
#define SOCKET_READ_BUFFER_SIZE (256 * 1024) // 256 KB buffer

int nBufSize = SOCKET_READ_BUFFER_SIZE;
int nVarSize = sizeof(nBufSize); // it's 4 byte, but sizeof is
better style!

nRet = setsockopt(m_sock, SOL_SOCKET, SO_RCVBUF,
                  (char *)&nBufSize, nVarSize);
ATLASSERT(nRet != SOCKET_ERROR);
```

 See documentation on *setsockopt()* for further details.

Reading Data Blocks with Visual Basic

Arrival data reading with C++, as shown in 'Handling Data Arrival – Continuous Measurements', can also be ported to VB. *Events* for VB are used here, with unique events for almost every type of arrival data (especially when using the asynchronous interface). Most of these

pass their results through basic data type parameters.

See “Handling Data Arrival – Continuous Measurements” on page 86.

Message notification methods with VB are not demonstrated here.

However, there are some exceptions where the data must be retrieved explicitly upon an incoming event. These types of events can be identified by the *DataReady* term in their names. The continuous measurement *events* are among these.

The code below shows an implementation of the *ContinuousPointMeasDataReady()* event handler. It does not demonstrate the processing of the data received. This handler does some diagnostics – checks whether the size of read data complies with the passed parameter. If OK, the size is displayed, otherwise an error message is shown.

By calling the *ObjConnectGetData()* function, the arrived data (that caused the *event*) is being read into a local buffer. The application interprets and processes the data. In order to get the measurement values, loop through the array and interpret the array elements with *MeasValueT* (not shown here).

VB may not be the right choice to process (high rate) continuous measurements, especially when running the interpreter. The VB project must be compiled first.

```
Private Sub ObjAsync_ContinuousPointMeasDataReady( _
                                    ByVal resultsTotal As Long, _
                                    ByVal bytesTotal As Long)
    Dim data As Variant
    Dim tp As VbVarType
    Dim sz As Long

    ObjConnect.GetData data
    tp = VarType(data) ' type; we expect a Byte arryay

    If (tp = vbArray + vbByte) Then ' Byte Array
        sz = UBound(data) + 1  ' index is zero based!

        If (bytesTotal = sz) Then
            MsgBox sz 'display # of bytes received
        Else
            MsgBox CStr("Unexpected size:") & sz _
            & CStr(", expected:") & bytesTotal
        End If

     End If
End Sub
```

 It is not necessary to read data here (with GetData). Answers may be filtered out and only those data packets of interest can be read.
With TCP/IP data must be read at socket level (see previous samples) otherwise no notification will arrive again.

The principles shown here also apply to message handlers, if one of the message notification mechanisms is selected.

 See “Answers from Tracker Server” on page 25 on how to mask/evaluate incoming data blocks.

VBA Macro-Language Support (Excel, Word, Access)

The *LTControl* COM component can also be used with VBA (Visual Basic for Applications), the built-in Macro language of MS Excel, Word and Access – with the exception that *structs* and *enums* are not fully supported with VBA that comes with Office 97. 'Ex' functions that take struct parameters cannot be used. VBA that comes with Office 2000 no longer has such limitations.

 It is highly recommended to use Office 2000 or higher for Tracker Server VBA Programming. Office 97 (Excel 97/Word 97) - apart from a

missing UDT - contain some bugs that make development of Tracker Server clients virtually impossible, as soon as *events* are involved. This bug leads to a completely corrupted file upon file saving, after an event has arrived.
For this reason, Excel samples delivered with the TPI-SDK are in Excel 2000 format. They may run with Excel 97 , but may be destroyed as soon as any changes are saved. Always maintain a safe (read-only) copy.
The following remarks only apply to Office 97 programming (Office 2000 VBA behaves as ordinary VB).

User-defined Types, the Differences between Visual Basic and VBA97

- Both allow defining user-defined *structs* locally. However, those *structs* exported by the LTControl (such as *PacketHeaderT*, *SingleMeasResultT*) are only recognized from within Visual Basic. VBA claims an error *Automation type not supported* if declaring, for example, a variable like:

```
Dim val As SingleMeasResultT // works with VB, but not VBA97
```

- Enums are not supported by VBA97. The compiler does not know the keyword *Enum*. User-defined *enums* cannot be defined locally, although this works with ordinary Visual Basic. It is also not possible to use *enum-* type variables that are exported by the LTControl. Declaration as follows are not possible in VBA97:

```
Dim cmd as ES_Command   // works with VB, but not VBA97
```

- When implementing an *EventHandler* that has *enum-type* parameters in Visual Basic will read as follows (only function header shown):

```
Private Sub CommandSync_ErrorEvent(_
                ByVal command As LTCONTROLLib.ES_Command,_
                ByVal status As LTCONTROLLib.ES_ResultStatus)
```

- When doing the same in VBA97 it will read as follows:

```
Private Sub CommandSync_ErrorEvent(ByVal command As Long, _
                                   ByVal status As Long)
```

- Visual Basic keeps the enum type information and recognizes the parameters with their correct *enum- types,* while VBA just passes them as long parameters.
  However, the symbols of the *enum* values are correctly recognized, although not checked by the compiler for correct typing (which can lead to errors, which are difficult to find).
  This problem is not specific to VBA, it also exists in VB. There are two different situations where *enums* and their value-symbols affect the interface:
- Method takes enum type parameters, for example, call *SetMeasurementMode* the same way for both VB and VBA:

```
ICommandSync::SetMeasurementMode(ES_MM_ContinuousDistance);
```

- *ES_MM_ContinuousDistance* will be correctly recognized as having the value '2' (see *enum* definition).
- Correct typing of values: VB as well as the VBA interpreter will not recognize typing errors in *enum* symbols here. However, both VB and VBA provide 'IntelliSense', providing for a selection from a list rather than having to type them in.
- *Event* handlers, as we have seen above, pass *enums* as long values in VBA. The incoming values can be tested against *enum* symbols. In an event handler, the following code

might be typical (example *ErrorEvent* in VBA):

```
Private Sub CommandSync_ErrorEvent(ByVal command As Long, _
                                   ByVal status As Long)
If (command = ES_C_Initialize) Then
   ' do something
End If

If (status = ES_RS_NoTPFound) Then
   ' do something
End If
End Sub
```

 Use extreme caution while typing the symbols with VBA 97. No 'IntelliSense' support is available.

Conclusion

- There is no problem with enums and VBA97. It is just a potential error source due to missing type checking.
- Structs (unless locally defined) are not supported in VBA97. LTControl offers an alternative to these functions offering no significant restriction on using VBA97.
- None of the e*vent* functions has *struct* parameters, and have, therefore, no restriction with VBA97 (both synchronous and asynchronous interface).

Continuous measurements and VBA

Events of continuous measurements do not directly pass the data.

 See “Handling Data Arrival – Continuous Measurements” on page 86 for details.

Handling continuous measurements within VBA requires care. *Events* can be 'subscribed' with the *WithEvent* keyword and pending data can be read with *GetData(),* as shown in:

 See “Reading Data Blocks with ” on page 95 for details.

Masking Data

The unavailability of (LTControl) structures prevents masking the data. With the byte-layout of the data blocks the appropriate bytes can be extracted 'manually' and assigned to basic data types.

 This is not convenient and exceeds the typical Excel programmer's expertise.

Even with VB, although *structs* are available, masking data is not that easy as in C++. By providing some helper functions, data blocks can be copied to appropriate *struct* parameters instead of pointer type-casts:

```
ILTConnect::ContinuousDataGetHeaderInfo()
ILTConnect::ContinuousPointGetAt()
ILTConnect::ContinuousPoint2GetAt()
ILTConnect::Continuous6DDataGetAt()
```

This allows extracting information of interest from data blocks of type *ES_DT_MultiMeasResult*, *ES_DT_MultiMeasResult2* and *ES_DT_Multi6DMeasResult*.

A VB (VBA) implementation, with comments, of the *ContinuousPointMeasDataReady* event handler that demonstrates usage of these functions reads as follows:

```
Private Sub LtSync_ContinuousPointMeasDataReady ( _
         ByVal resultsTotal As Long, ByVal bytesTotal As Long)

    ' a continuous point meas packet came in. Note that in
    ' case of continuous measurements (due to multiple points /
    ' variable size of packet) only # of results and packet size
    ' are passed in (which both are not really needed here)
    ' So we first must GET the data, then retrieve information
    ' out of the gotten block.

    ' since we are doing function calls to a COM object
    ' (LtConnect) that can throw exceptions, we need an error
    ' handler. Note we would not require an error handler in the
    ' other Event Handlers (LtSync_ReflectorsData,
    ' LtSync_ReflectorPositionData) because (usually) no COM
    ' functions are called there subsequently

    On Error GoTo ErrorHandler

    ' 1. Get the data

    Dim data As Variant
    LtConnect.GetData data

    ' 2. Get header info. Calling this function is optional.
    ' the only thing we need here is numResults. However,
    ' it's the same as resultsTotal passed to the functions.

    Dim numResults As Long
    Dim measMode As Long
    Dim temperture As Double
    Dim pressure As Double
    Dim humidity As Double

    LtConnect.ContinuousDataGetHeaderInfo data, numResults, _
                      measMode, temperture, pressure, humidity

    ' since we have numResults twice from different paths, lets
    ' check them for compliance!

    If Not (numResults = resultsTotal) Then
       MsgBox "Fatal Error - unexpected discrepancy"
    End If

    ' since we know how many results, we can loop over the index
    ' Note that index runs form 0 to numResults - 1

    For index = 0 To numResults - 1

        ' data and index are input parameters, rest  output

        LtConnect.ContinuousPointGetAt data, index, status, _
                                time1, time2, dVal1, dVal2, dVal3

        ' TODO: do something with each result here

    Next

    Exit Sub
ErrorHandler:
    MsgBox (Err.Description)
End Sub
```

ContinuousPointGetAt()/Continuous6DDataGetAt() may have an impact on performance. They have been primarily designed for use with VB(A). For C++ applications, more efficient ways to extract continuous measurements exist.

VBA applications, depending on data processing, may not have enough performance when using continuous high data rates. Always run compiled versions. In special cases the incoming results need to be buffered.

Use of values instead of symbols, in Visual Basic, avoids the problem of typing incorrect *enum* symbols, which cause errors difficult to detect.

 A complete *.tlh* file is automatically generated when importing *LTControl.tlb* into a VC++ project.

Scripting Language Support

Pure scripting languages VBS (Visual Basic Script), JavaScript etc. are currently not supported by the LTControl COM component.

This would require *IDispatch* interfaces rather than custom interfaces. Combinations of *IDispatch* and custom interfaces (dual interfaces) have the same disadvantage as *IDispatch* – lack of performance.

Excel Control for Tracker Server

Sample 8

This sample works only with Excel 2000, and consists of an Excel sheet with a VBA macro *LtcExcel*. Tracker server client VBA-programming with Excel 97 (Office 97) is not recommended. The variant *LtcExcelWithImage2.xls* is an extended version of the *LtcExcel.xls* application and includes 'Still Video Image' support. This feature requires the tracker being equipped with a video camera.

 See "VBA Macro-Language Support (Excel, Word, Access)" on page 97.

The essential difference between a VB client and an Excel client is that the Excel sheet takes the

role of a VB Form. That is, data input/output goes through cells.

Sample 11 *GetStillImage*

This LtcWin32Client sample demonstrates the *GetStillImage* command, which requires a camera mounted on the tracker. The application skips the physical disk bitmap file. The bitmap file contents is read directly into memory buffers and shown on screen. This also allows a simple full size scaling option.

Before application start, the system settings flag 'Has Video Camera' must be enabled.

Asynchronous interface

This sample implements the more complex (from programmer's point of view) asynchronous interface. The complexity comes from the fact that the programmer has to perform the task of synchronizing, because command calls are non-blocking. This is required for data that is non-synchronous (for example continuous measurements – not in this application, however.).


See implementation of synchronous command for *GetStillImage* in “Remarks” on page 105.


This sample uses a different method than the relatively complex connection point interface in Sample 7. The connection point interface is a solution, when using Visual Basic/VBA.

This is a pure Win32 application and does not use the MFC library. This was intended in order to make everything as transparent and 'lightweight' as possible. The implemented bitmap (file) reading algorithm is not intended for general purpose other than the simple b/w camera image format currently provided.

Remarks

When designing a client application using the LTControl COM component, either the synchronous or the asynchronous command interface must be used.

- With the asynchronous interface and the events notification (that is, calling *SelectNotificationMethod* with *LTC_NM_Event*), an Event- Sink must be implemented. In VB, this is done by defining the *WithEvents* keyword, but in C++ this is a bit more complicated. In addition, the appropriate event handlers must be implemented .

   With any other notification mechanism, the event sink is not required and the *WithEvents* keyword must be removed. Implement Windows message handlers and not event handlers, in this case.

- With the synchronous interface, some answers remain asynchronous by their nature - *continuous measurement packets*, *Reflector Positions* and *error answers* (these may partly occur non-command related, for example *beam broken*).
  With synchronous commands, events or notifications must still be caught - See former paragraph. Any other notification mechanism does not need an event sink, and the *WithEvents* keyword must be removed. In this case, do not implement event handlers; appropriate Windows message handlers must be implemented instead.

- Using both interfaces in the same LTConnect instance – although possible – usually makes

no sense and partly leads to duplicate answers.

- On multi tracker (multi tracker server) systems, create a separate instance of LTConnect for each tracker.
- Do not test explicitly against the VB keyword 'True', if using the *Get<FunctionName>Ex* methods of the LTControl, for those commands returning Boolean data within their result structure. This is because the Boolean member in these structures, if true, is one (1). However, the VB keyword 'True' evaluates to (-1). Always test the variable directly, or against 'Not False'.

Example

```
ObjSync.GetContinuousDistanceModeParamsEx dataout

If (dataout.bUseRegion) Then
    MsgBox "bUseRegion is True"
End If

or

If Not (dataout.bUseRegion = False) Then
    MsgBox "bUseRegion is True"
End If

are both correct. However, the following would evaluate to a
wrong result:

If (dataout.bUseRegion = True) Then
    MsgBox "bUseRegion is True" ' No message even flag true!
End If
```

5.Command Description

Special Functions

Some of the commands/procedures, which have been referred to in this manual are described in detail, with some background information.

Get Reflectors Command

The *GetReflectors* command is often misinterpreted. *GetReflectors* is used to 'ask' the Tracker Server, which reflectors are currently defined, and to get the relation between reflector names and reflector IDs.

Related Commands

- SetReflector
- GetReflector

Comments

GetReflectors causes as many *GetReflectorsRT* data packets to arrive, as reflectors are defined in the tracker database. Each one of these packets contains the following information:

```
struct GetReflectorsRT
{
   struct BasicCommandRT      packetInfo;
   int                        iTotalReflectors;
   int                        iInternalReflectorId;
   enum ES_TargetType         targetType;
   double                     dSurfaceOffset;
   short                      cReflectorName[32];  // Unicode!
};  of Reflector data packets of the following
```

iTotalReflectors

iTotalReflectors is just for programmers' convenience.

- Names the number of reflectors known to the system and has the same value in every packet.

- Provides information, on arrival of the first packet, as to how many packets are still outstanding.
- Counts the incoming packets to know when the last one has arrived.

iInternalReflectorId/cReflectorName

The commands *iInternalReflectorId* and *cReflectorName* provide important information for the user interface/programmer

- The reflector name is a string value (in Unicode), which is see on the user interface of the application software.
- This reflector name is an alias for the reflector ID and cannot be resolved by the system.
- The system can (internally) only deal with reflector IDs, which are integer numbers.
- The commands take/return a reflector ID as a parameter.
- It is crucial to provide the correct reflector ID to *SetReflector.*

   Passing the ID of an unintended (but existing) reflector will cause wrong measurement results.
- Programmers often fill all reflector names in a list box. When the user selects one of the reflectors shown in the list box, a *SetReflector* command is carried out.

   Hence the need for a 'lookup table'.

List index

- It is not correct to use the index of the list box as a reflector ID. This is because the reflector IDs are arbitrary in sequence and may contain gaps.

- The programmer must not assume that the reflector IDs are a sequence of 1….n without any gaps. Although most systems may deliver reflectors with sequential reflector IDs starting from 0 with no gaps

   This may not be presumed. Every system behaves differently.

- *GetReflectors* may deliver for example 3 reflectors with the following Names and IDs:

Name	ID
CCR-75mm	7
CCR-1.5in	2
TBR-0.5in	5

Lookup Table

The list box indices will range from 0 to 2, when the three names are entered in a control list box, in the order shown above. A lookup table is therefore required to match the index values to the reflector IDs. Such a lookup table is shown below:

Index	ID
0	7
1	2
2	5

 The call to *SetReflector* must pass the reflector ID, not the list box index. A frequent source of a programming error.

Reflector Name – Unicode Format

The reflector name is always in Unicode format, irrespective of whether the application is in Unicode or ANSI.

 Names in C/C++ applications may have to be converted accordingly.

See "Sample 7" on page 84, which implements reflector handling with a list box. It uses a MFC Map as a lookup table.

Simpler solutions exist with just an integer array.

See also "Sample 7" on page 84 or "Receiving Data Sample 9" on page 66, on how to interpret reflector names in Unicode format correctly.

Still Image Command

For trackers equipped with an Overview Camera, the *GetStillImage* command takes an image and delivers it as a file image data block.

Related Commands

- GetStillImage
- SetCameraParams
- GetCameraParams
- StillImageGetFile (COM, not in C++)
- WriteDiskFile (COM only)

These commands are available on all TPI levels (C, C++, COM). *Set/GetCameraParameters* is not explained here further.

See Reference Manual for details.

Preconditions

The following preconditions have to be fulfilled:

- Camera mounted on tracker
- System settings: “Has video” flag activated
- Tracker must be in camera view (command ActivateCameraView)

Application of GetStillImage – *C/C++*

The application of GetStillImage is explained below using code fragments.

GetStillImage must be called with the parameter *ES_SI_Bitmap*. The parameter *ES_SI_Jpeg* is not supported yet.

- The answer to a successfully executed *GetStillImage* command results in a *GetStillImageRT* data structure.
- Apart from the common header information, this structure echoes the file type (imageFiletype =ES_SI_Bitmap), the size of the file (lFileSize), and the first Byte of the file (cFileStart).
- The following code accesses the core file data and writes it to a physical disk file:

```
// assume pData contains the data- block received
// to a GetStillImage(ES_SI_Bitmap) command

long lFileSize = ((GetStillImageRT*)pData)->lFileSize;
char cFileStart = ((GetStillImageRT*)pData)->cFileStart;

FILE *pFile = NULL;

if ((pFile = fopen("C:\\Temp\\img.bmp", "wb" )) != NULL )
{
   long lWritten =
        fwrite(&cFileStart, 1, lFileSize, pFile);

   if (lWritten != lFileSize)
      printf("File could not be written(\n");
   else
      printf("wrote %d bytes\n", lWritten);

   fclose(pFile);
 }
```

- The disk- file can be skipped and a memory-mapped file can be used instead. **OR**
- With the file structure of the Bitmap file, the bitmap information can be extracted from the data block and used directly with GDI functions.
- In the code above, it was assumed that pData contained a complete *GetStillImageRT* structure with complete file data padded.

WinSock2 API/MFC CasyncSocket

- Using WinSock2 API or MFC CasyncSocket, to read directly from the socket, must consider the implications of large file data.
  - Since the file data is relatively big (~70 KB), it is very unlikely that it will arrive as one single data block over TCP/IP.
  - Provisions must be made to repeat reading data until the data packet is complete.
  - A technique to achieve this is shown in the *OnMessageReceived* code sample

     See "Queues and Scattered Data" on page 91.

     See also receiver thread in "Receiving Data Sample 9" on page 66.

COM TPI within C/C++

When using the COM TPI (within a C/C++ application), the results of the LTControls *GetStillImage* (synchronous) function can be assumed to be complete. See related code extract below. When receiving StillImage data asynchronously (Event Handler, MessageHandler), the difference is that the data will not be provided directly through a parameter. So *ILTConnect::GetData()* must be used first.

 Note the Variant- type parameter of the fileData.

 See "*Application of GetStillImage* – C/C++" on page 110.

GetStillImage – Synchronous

```
void CCPPClientDlg::OnButtonStillImage()
{
   HRESULT hr = 0;
   long lFileSize;

   VARIANT vt;
   VariantInit(&vt);

   try
   {
      if ((hr = m_pLTCommandSync->GetStillImage(ES_SI_Bitmap,
                                        &lFileSize, &vt)) == S_OK)
      {
         ASSERT(vt.parray->rgsabound[0].cElements ==
                (unsigned long)lFileSize);

         FILE *pFile = NULL;

         // write file to current runtime location
         if ((pFile = fopen("image.bmp", "wb")) != NULL )
         {
           long lWritten = fwrite(vt.parray->pvData, 1,
                                  lFileSize, pFile);

           if (lWritten != lFileSize)
              AfxMessageBox(_T("File could not be written\n"));

           fclose(pFile);

           // Display the image using MSPaint,
           // but first close previous instance
           //
           HWND hWnd = ::FindWindow(_T("MSPaintApp"), NULL);

           if (hWnd) // paint is already running - close first
              ::SendMessage(hWnd, WM_SYSCOMMAND, SC_CLOSE, 0);

           WinExec("mspaint.exe image.bmp", SW_SHOWNOACTIVATE);
         } // if
      } // if
   }
   catch(_com_error &e)
   {
      Beep(4000, 100);
      AfxMessageBox((LPCTSTR)e.Description());
   }

   VariantClear(&vt);  // Avoid memory leak
}
```

GetStillImage – Asynchronous

```
void __stdcall OnStillImageDataReady(ES_StillImageFileType
                  imageFileType, long fileSize, long bytesTotal)
{
   ASSERT(m_bUseAsync);

   VARIANT vt;
   VariantInit(&vt);

   m_pLTConnect->GetData(&vt);

   ASSERT(vt.parray->rgsabound[0].cElements ==
          (unsigned long)bytesTotal);

   GetStillImageRT *pData =
                     (GetStillImageRT *)vt.parray->pvData;

   ASSERT(pData->lFileSize== fileSize);

   // Do something with the file, for example write out
   // to a disk file - like shown in code above

    VariantClear(&vt); // Avoid Memory leak
}
```

COM/VB(A)

Neither type- casts nor writing binary files are common tasks in VisualBasic. In order to achieve the same StillImage features from VB(A), some convenience Functions have been added to the COM TPI: *StillImageGetFile* and *WriteDiskFile*.

This is an extract from an Excel application. The image is displayed in an Image dialog control (named Image1):

```
Private Sub GetStillImage_Click()
   On Error GoTo ErrorHandler

    Dim fileData As Variant
    Dim size As Long

    ObjSync.GetStillImage ES_SI_Bitmap, size, fileData
    ObjConnect.WriteDiskFile fileData, "C:\Temp\img.bmp"

    ' Now load picture into sheet
    Image1.Picture = LoadPicture("C:\Temp\img.bmp")

    Exit Sub
ErrorHandler:
   MsgBox (Err.Description)
End Sub
```

Event handler

Within an event handler, the file data structure must be extracted first, since *GetData* delivers the complete data packet including header information. A similar helper function is required in VB, since no casting to (*GetStillImageRT**) is available.


See "Continuous measurements and VBA" on

page 100 for similar method using *ContinuousDataGetHeaderInfo*.

```
Private Sub ObjAsync_StillImageDataReady(ByVal imageFileType As
LTCONTROLLib.ES_StillImageFileType, ByVal fileSize As Long,
ByVal bytesTotal As Long)

    Dim fsize as Long 'dummy

    ObjConnect.GetData data 'Get whole packet (incl header)

    ' retrieve out size and file data
    ObjConnect.StillImageGetFile data, fsize, file

    ObjConnect.WriteDiskFile file, "img.bmp"

    ' Now load picture into sheet
    Image1.Picture = LoadPicture("img.bmp")

End Sub
```

 Although designed for use with VB, *StillImageGetFile* and *WriteDiskFile* can also be used in LTControl based C++ applications.

Image Click Position

Click positions on the Image are currently written out to Excel cells. These values can be used to calculate relative tracker movement angles, call *MoveRelativeHV* to direct the tracker there and then request a new Image.

```
Private Sub Image1_MouseDown(ByVal Button As Integer, ByVal
Shift As Integer, ByVal X As Single, ByVal Y As Single)
    Beep
    ws.Cells(2, 2).Value = X
    ws.Cells(3, 2).Value = Y
    ws.Cells(5, 2).Value = Shift
End Sub
```

Live Image display

Live Image Control LTVideo2.ocx

The live camera display from the Overview Camera can be implemented into user applications by using an ActiveX control, LTVideo2.ocx. See SDK lib directory, ANSI/Unicode subdirectories.

Registering LTVideo2.ocx

LTVideo2.ocx is an ActiveX type COM object and requires registration on the Application Processor.

From the command line perform the following command:

Regsvr32 <Path>\LTVideo2.ocx, where <path> depends on the location of the file – typically C:\WINNT\System32.

ANSI/Unicode Version

Use the Ansi version for Win98/ME platforms and the Unicode version for WinNT/2000.

 See Version info of LTVideo2.ocx for details, under *File Properties* > *Version TAB*.

Development Platforms

For Visual Basic or Office, the ActiveX controls must be added as a reference.

For VC++, a wrapper class is generated using:

Add to Project/Components > *Controls* > *Controls type library* from Visual Studio.

LTVideo2.tlb

LTVideo2.tlb is the related type library delivered for convenience. LTVideo2.ocx contains all type information required.

Server Address

LTVideo2.ocx has a property server address, which must be set according to your server address.

The port number is 5001. Any changes to the port number must also be done on the server side.

The size must have a width/height proportion of 4:3. The image must be started/stopped by invoking the method Start/StopLiveImage.

 See Microsoft documentation, for further information on how to use ActiveX controls in general.

Events/Methods

The essential methods of the camera OCX are:

- StartLiveImage()
- StopLiveImage()

To alter the default frame rate (15/sec), the following methods are used:

- FrameRateStepUp()
- FrameRateStepDown()

The following event, VideoClick, is used:

```
void VideoClick(double deltaHz,
                double deltaVt,
                long posX,
                long posY,
                long flags);
```

This event occurs when clicking on the image with the mouse. The event parameters are as follows:

- DeltaHz, deltaVt: The angles that can be passed to the MoveHV command, in order to move the tracker to the clicked position.
- PosX, posY: The pixel values of the clicked position within the image coordinate system (top/left = 0, 0).
- The flags parameter can be used to figure out which modifier keys are pressed during the click. The flags parameter is the same as provided by the OnLButtonDown standard message.

   See Microsoft MFC documentation, for details.
- Server address and Port number must be passed as properties.
- An RGB triplet can be passed to alter the color of the crosshair

Orient To Gravity Procedure

This function is used to measure the tilt of the tracker's primary z-axis (standing axis) with respect to the vertical. This can be used to orient the measurement network to gravity. The tilt is specified by two angular components about the tracker's internal x and y-axes.

Related Command

CallOrientToGravity

Comments

- This command is only available in combination with a Nivel20 Inclination Sensor.
- Executing this command drives the tracker head to 4 different positions on the xy plane:
  - Taking Nivel20 measurement samples.
  - In addition, the station inclination parameters Ix and Iy are calculated and returned as result parameters.
- Executing this command does not 'implicitly' apply any orientation values to the system.
- In order to 'activate' the station orientation to gravity, the two result values, Ix and Iy, must be explicitly set with the command *SetStationOrientationParams* (Rotation angles rot1 and rot2).

  See "Mathematics" on page 129 for mathematical description.

Transformation Procedure

This procedure matches a measured set of points to a given set of nominal points by using a least squares, best fit method. The procedure calculates the 7 parameters (x,y,z, omega, phi, kappa, scale), which describe the 'transformation filter' to be applied to the measured points in order to represent these in the coordinate system defined by the nominal points.

Related Commands

- ClearTransformationNominalPointList
- ClearTransformationActualPointList
- AddTransformationNominalPoint
- AddTransformationActualPoint
- SetTransformationInputParams
- GetTransformationInputParams
- CallTransformation

- GetTransformedPoints

Comments

The command CallTransformation displays a transformation carried out with *Set/GetTransformationParams*

Before doing a *CallTransformation,* both point lists, nominal and actual must be prepared. They must contain the same number of elements.

EmScon System Settings

The system settings of emScon (units, coordinate type and coordinate system) must reflect the current input data. Point input values (nominal/actual) are interpreted by emScon based on the current emScon system settings.

- Additional parameters can be set by using the *SetTransformationInputParams* command (For example to fix or weigh certain parameters).
- After a successful calculation, additional results in terms of transformed points and residuals can be retrieved optionally by using *GetTransformedPoints.*

  None of the 7 calculated transformation parameters (received as output from *CallTransformation)* are automatically applied to the system. This must be done explicitly by calling *SetTransformationParams.*

  See "Mathematics" on page 129 for mathematical description.

Automated Intermediate Compensation

The Intermediate Compensation is a simple and fast procedure to perform a fully automated intermediate compensation, where the tracker is in a fixed installation.

Tracker Geometry

Out of a total of 15 parameters, which affect the trueness of the tracker geometry, the most

significant changes are affected by these three parameters:

See emScon manuals, for more information.

- Transit axis tilt, i
- Mirror tilt, c
- Vertical index error, j

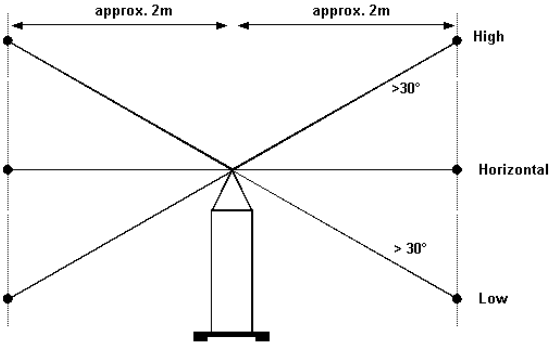
Intermediate Compensation refreshes these three parameters by taking a small number of Two-face measurements. If the result is accepted, it updates only these three parameters and takes over the rest of the overall 15 parameters from the last Full Compensation. It is a simpler and faster procedure than a Full Compensation.

Intermediate vs. Full Compensation

Intermediate Compensations do not replace Full Compensations. Regular intermediate compensations extend the interval at which full compensations need to be carried out.

Setup

A recommended setup is shown below with a network of fixed targets. Based on a given drive library the laser tracker measures the target points automatically and calculates the Intermediate Compensation results.



Two face measurements in the vertical plane

The automated Intermediate Compensation routine requires that all target locations are fitted

with reflectors (recommended 0.5” Tooling Ball or Corner Cube), before the routine is started.

Area Required

Make sure that no one walks around the area during the whole Compensation procedure. Vibration can affect the measurement and walking through the beam causes the signal to break. If a measurement fails, the system automatically repeats the measurement to achieve a successful measurement, a maximum of three times.

Procedure

Requirements

The automated Intermediate Compensation can only be started when the Leica Tracker system is ready to measure.

 See "Integration in Application Software" on page 15.

For the initial setup it is required that the locations of the fixed targets are measured manually. These locations provide the information for the driver points.

- Six Two Face measurements, in two groups of 3 each.
- Each group of 3 points is in an approximate vertical line.
  - Minimum distance from the tracker is 2m.
  - The high and low measurements should be more than 30° from the horizontal.
- The groups should have a horizontal angle separation of about 180°, i.e. all measurements should lie approximately in the same vertical plane.

Minimum Measurements

A minimum of 4 measurements is required (mathematically). More measurements reduce the

influence of errors. In addition, unstable conditions, such as vibrations and rapid temperature changes, make it necessary for more measurements to be taken. The following combinations are examples:

- Eight measurements in 4 pairs (high and low) separated by approx. 90°.
- Twelve measurements in 4 groups of 3 each (high, low, horizontal), separated by approx. 90°.

Related Commands

- ClearDrivePointList
- AddDrivePoint
- CallIntermediateCompensation
- SetCompensation

Comments

Settings

Current emScon system settings, such as units, coordinate system and coordinate type, are taken over when emScon interprets point input (driver point) values. All points in the drive library must be known within ± 2mm (0.0787 in) tolerance, otherwise this will cause an error in the measurements.

The settings, such as units, coordinate system and coordinate system type, must correspond to the input data. Ensure that the settings describe the environment of the driver points before they are uploaded to the server.

One of the first actions of the automated compensation algorithm is to check the geometry of the used driver points. If the target setup fits the requirements (as described above), then the process continues with the measurements, otherwise it will abort.

Compensation Results

A successful Intermediate Compensation procedure returns the following information:

- Total RMS
- Max. Deviation
- Error bit filed with the information of warnings and errors.

Compensation Intervals

An intermediate compensation is recommended when the maximum deviation is ≤ 0.0012 deg (13^{cc}). With the command *SetCompensation* the new calculated compensation can be activated.

Two Face Field-Check

A field check is a control process of the Compensation parameters. It checks the condition of the Leica Tracker, with respect to predefined parameters. It does not, however, provide for compensatory corrections.

Periodicity

If the tracker is used in a stationary position, conduct the field check on a weekly basis. If the field check results show no change, over a period of six weeks, carry out field checks at least once a month.

If the tracker has been moved, always carry out a field check before taking measurements.

 Compensations and field checks must be carried out in normal working conditions, under which the measurements are taken.

Field check two face Measurement

Two face measurements with 4 to 5 reflector positions, distributed over the whole object range, will indicate whether the Tracker compensation is within specifications. To achieve a 2-sigma accuracy, 95 % of the measurements must be within the specification.

Client Routine

The Tracker Server Programming Interface does not have a specific two face measurement mode.

A client routine is required, which can use the basic functionality provided.

See "Procedure - Measurement" on page 126

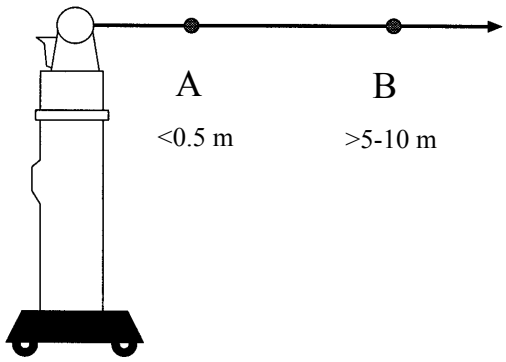
Procedure - Preparation

The procedure requires the following three setups:

1. Two measurements on a straight line.
2. One measurement set on a vertical line.
3. One measurement plus or minus 90° to the vertical line.

Measurements on a Straight Line

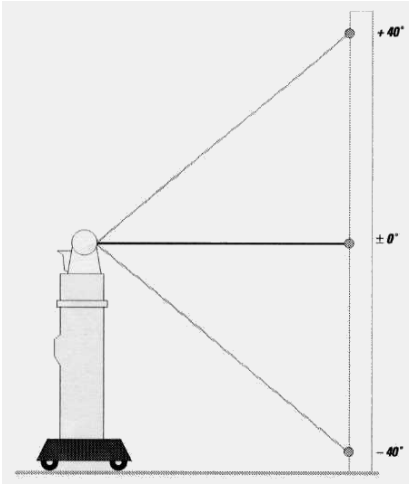
1. The two measurements must be taken on a straight line (ray) at the same level as the as the Tilting mirror of the Tracker. Point A <0.5 m and Point B within 5-l0 m.

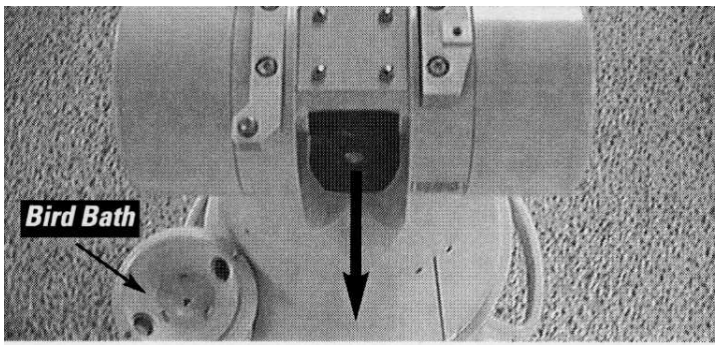


Measurements on a Vertical Line

2. All 3 measurements should be taken in a vertical line.
    1. Mid point 0° at Tracker head height.
    2. Upper measurement at +40° deg.
    3. Lower measurement at -40° deg.

During measurements, the Birdbath should not point in the direction of measurement.





Measurement ± 90° to the Vertical Line.

3. Setup the tripod at 90°, as shown in the graphic below.

The Tracker is setup such that it can turn to the 90° position, without running into stop.

Procedure - Measurement

1. Set up the tracker.
    See "Integration in Application Software" on page 15 for details.
2. Set the coordinate system type to spherical clock wise, SCW,
   TPI command: SetCoordinateSystemType.
3. Set the Stationary Measurement Mode.
   TPI command: SetMeasurementMode
4. Set the Stationary measurement parameter.
   MeasTime to 10000ms
   TPI command: SetStationaryModeParams
5. Attach the reflector to the target location.
6. Point the tracker to the target location.
   TPI command: e.g. GoPostion. This is only possible when the coordinates of the point are known within ± 2mm, otherwise track the reflector manually from the Bird bath.
7. Execute the Stationary Measurement in Face I and save it.
   TPI command: StartMeasurement
8. Execute the command Change Face, which puts the Laser Tracker from Face I to Face II.
    The pointing to a fixed reflector position from a station should be the same in both faces.
   TPI command: ChangeFace
9. Execute the Stationary Measurement in Face II and save it.
   TPI command: StartMeasurement.
10. Execute the command Change face, which puts the Laser Tracker from Face II to Face

I.

TPI command: ChangeFace.

11. Repeat the steps 5 - 10 for all target locations.

Procedure - Calculation

Dev_{vt} = vertical angle Face I – vertical angle Face II

Dev_{h} = horizontal angle Face I – horizontal angle Face II

Both measurements are in Face I representation. Face II measurements are represented in Face I.

Example

Dev_{vt} = 90.7289893– 90.7287338 = 0.0003 Deg
Dev_{h} = 269.9877001– 269.9879985 = -0.0003 Deg

Tolerances

The recommended tolerances of the deviations are:

Vertical angle = ±13cc (0.0012 Deg)

Horizontal angle = ±13cc (0.0012 Deg)

 When the tolerance is exceeded, an Intermediate Compensation is recommended.

6.Mathematics

Transformation

Introduction

It is a mapping of a set of points called actuals or measurements to an equal number of points called nominals or reference. The transformation parameters consists of the following:

- scaling
- rotation
- translation

Computation of a point to point transformation should not be confused with applying a given transformation.

In mathematical terms, the computation of a transformation is a nonlinear, weighted, least squares problem. It is solved through a Newton iteration (linearization) consisting of the following steps.

1. An initial approximation is calculated, ignoring the accuracy (statistics) of the input.
2. The initial approximation is improved iteratively, until a certain accuracy goal is achieved.

Transformation Parameters

A transformation is described numerically by three (3) translation, three (3) rotation and one (1) scale parameter. The scale is typically close to 1, e.g. when describing a temperature dependent

dilation. The accuracy of the nominal and actual points is propagated to the transformation parameters and the transformed points.

The transformation is a similarity map either given in its forward form

$$T(x) = t + Rx/s$$

or as an inverse

$$T(x) = R^{-1}s(x - t)$$

with

- t = 3D translation vector
- R = 3*3 rotation matrix
- s = scale

These seven parameters are determined such that

$$T(actual) = nominal + residual$$

for all points with small residuals in the weighted least squares sense. The transformation can be interpreted as a coordinate system with its origin at *t* and the axes given by the columns of *R*. In terms of the Euler angles.

In terms of the *Euler angles* Ω, Φ, K the rotation matrix assumes the form

$$\begin{pmatrix} \cos(\mathrm{K})\cos(\Phi) & -\sin(\mathrm{K})\cos(\Phi) & \sin(\Phi) \\ \sin(\mathrm{K})\cos(\Omega)+\cos(\mathrm{K})\sin(\Phi)\sin(\Omega) & \cos(\mathrm{K})\cos(\Omega)-\sin(\mathrm{K})\sin(\Phi)\sin(\Omega) & -\cos(\Phi)\sin(\Omega) \\ \sin(\mathrm{K})\sin(\Omega)-\cos(\mathrm{K})\sin(\Phi)\cos(\Omega) & \cos(\mathrm{K})\sin(\Omega)+\sin(\mathrm{K})\sin(\Phi)\cos(\Omega) & \cos(\Phi)\cos(\Omega) \end{pmatrix}$$

The current implementation fails if $\Phi = \pm\pi/2$. As a workaround, an arbitrary pre-rotation can be applied to one of the point sets.

Transformation Types

Usually transformations are calculated for two purposes:

- **Orientation** – (Fig. 1 & Fig. 2)Alignment of a tracker with respect to a world coordinate system (WCS). The world coordinate system is either defined at a principal

station or at an object, as a (CAD) coordinate system. In Orientation, the reference points are given in a world coordinate system, while the actual point coordinates refer to the tracker's coordinate system.

- **Transformation** – (Fig. 3) Creation of an object coordinate system (also called UCS – user coordinate system) to view the actual point coordinates with respect to an object coordinate system. The reference points are given in the object coordinate system and the measurements in the world coordinate system (WCS).

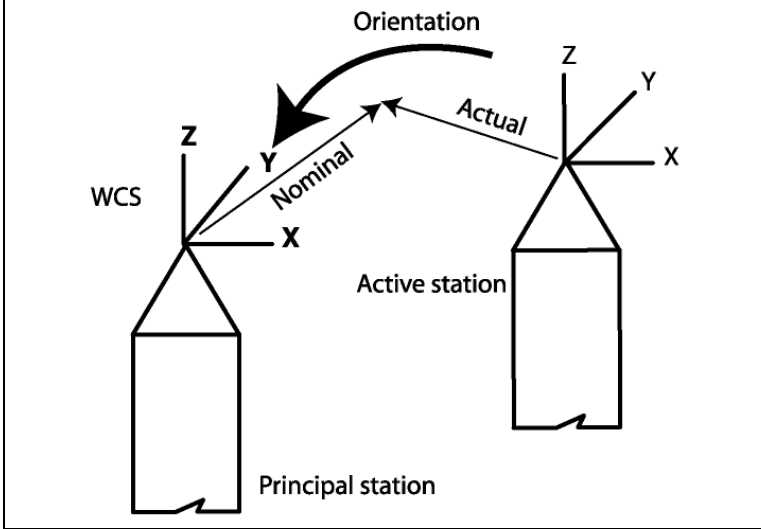


Fig. 1

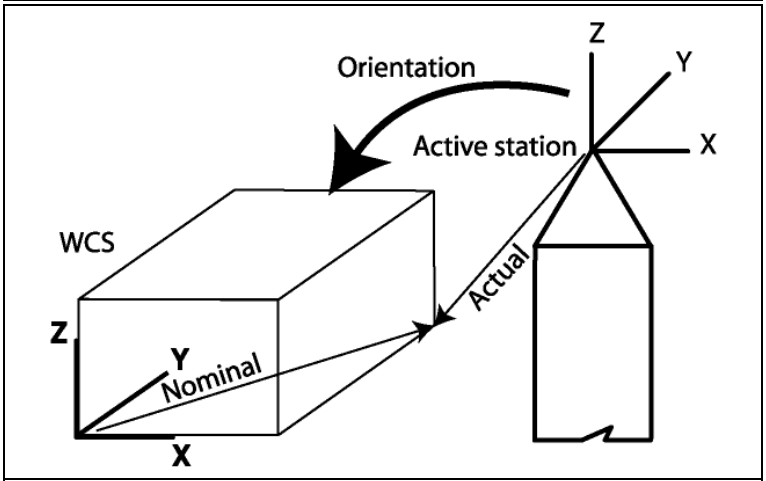


Fig. 2

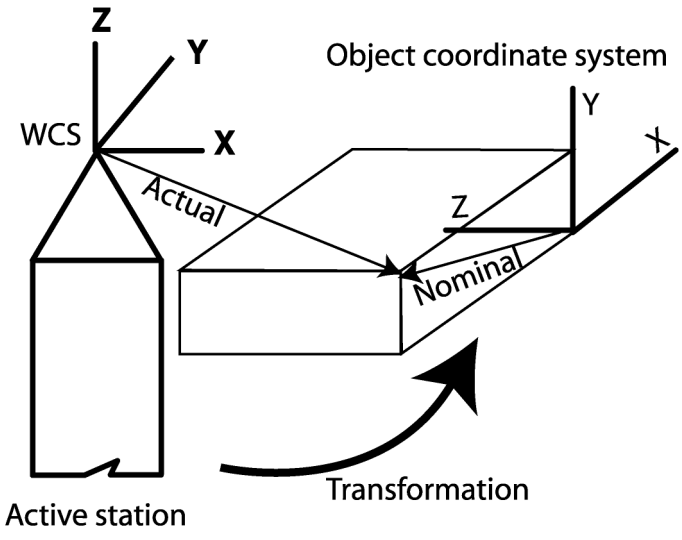


Fig 3

Transformation vs. Orientation

Transformation and orientation are inverse to each other as mappings. For orientation/alignment, the world coordinate system (WCS) is the reference, for the creation of an object coordinate system the object is the reference. The transformation, which is downloaded to emScon, will be a transformation from/to a world coordinate system to/from an object coordinate system.

Orientation

To orient a station use

```
SetTransformationInputParams(ES_TR_AsOrientation, … );
```

The transformation has the form

$$nominal = t + R * actual / s + residual$$

Setting the calculated transformation parameters as orientation parameters and remeasurement of the reference points yields actual coordinates approximately equal to the nominal coordinates.

Transformation

To calculate transformation parameters use

```
SetTransformationInputParams(ES_TR_AsTransformation, … );
```

In this case the transformation has the form

$$nominal = sR^{-1}(actual - t) + residual$$

which is inverse to the one in the orientation case. Setting the calculated transformation parameters and remeasurement also yields actual coordinates approximately equal to the nominal coordinates.

Point Accuracy

Each nominal point, measured point, transformed point or residual vector has accuracy information stored in a symmetric 3 by 3 covariance matrix.

$$covarianceMatrix = \begin{pmatrix} stdDev_1^{\,2} & covar_{12} & covar_{13} \\ covar_{12} & stdDev_2^{\,2} & covar_{23} \\ covar_{13} & covar_{23} & stdDev_3^{\,2} \end{pmatrix}$$

Its eigenvectors and eigenvalues define the error ellipsoid. If the covariance matrix is diagonal, the axes of the error ellipsoid are parallel to the coordinate axes. In correlations

$$\rho_{ij} = \frac{covar_{ij}}{stdDev_i * stdDev_j}$$ satisfy

$$-1 \le \rho_{ij} \le 1$$

All statistical points and 3D vectors can be represented in the following non-redundant form consisting of 9 values:

Coord1, Coord2, Coord3, StdDev1, StdDev2, StdDev3, Covar12, Covar13, Covar23

This format is used throughout the emScon TPI. For continuous measurements the a priori covariance matrix of a point measurement is calculated according to the tracker accuracy. For single point measurements (stationary, sphere center and circle center) the a posteriori or repeatability covariance is calculated from the actual statistical variation of the many measurements.

In the (spherical) tracker coordinate system the a priori covariance matrix is diagonal. Conversion to Cartesian coordinates results in a full matrix.

Transformation to other coordinate systems again transforms the covariance matrix. Maximal consistency is achieved by using the full matrix. However, at any stage the standard deviations, i.e. the square roots of the diagonal entries provide sufficient condensed information on the accuracy of the respective triple.

The covariance matrix of nominal points is diagonal. However, if measured points are used as reference, a full matrix may apply to nominal points as well.

Input of Transformation Computation

Nominal Points

Nominal points are added as shown in the following example:

```
AddNominalPoint(1, 2, 3, ES_FixedStdDev, ES_UnknownStdDev,
ES_ApproxStdDev, 0, 0, 0);
```

The parameters listed are the three coordinates together with their standard deviations and covariances.

We recommend use of the following predefined standard deviations:

Coordinate accuracy	Symbol	Value
Fixed (exactly known)	ES_FixedStdDev	0
Unknown (free)	ES_UnknownStdDev	1E35
Approx. (reasonable)	ES_ApproxStdDev	1E15
Weighted		> 0, < 1E10

Approximately known coordinates are used to calculate an initial approximation. In a minimum configuration, the solution would be ambiguous without this additional information.

See "Examples" on page 137.

Actual Points

Actual points are added in the following form

```
AddActualPoint(-12.487, -5.79687, 5.49683, 0.0001, 0.0001,
0.0001, 0, 0, 0);
```

The number and order of actuals must agree with that of the nominals. The parameters are typically obtained from a single point measurement. The unit settings and transformation parameters must be the same for the measurement and for adding the actuals.

Parameter Constraints

If any of the seven (7) transformation parameters is known, a priori, its value can be fixed, in the same way as for the nominal coordinates. Frequently the scale is fixed to be 1 and the other parameters are free as in the following example:

```
SetInputParams(0, 0, 0, 0, 0, 0, 1, ES_UnknownStdDev,
ES_UnknownStdDev, ES_UnknownStdDev, ES_UnknownStdDev,
ES_UnknownStdDev, ES_UnknownStdDev, ES_FixedStdDev);
```

The values of unknown parameters can be set arbitrarily. In the current implementation constraints are not used to reduce the required number of known nominal coordinates. Constraints are not taken into account for the initial approximation. Erroneous constraints usually result in large point residuals, RMS, Maxdev. and variance factor. To fix some or all components of the translation vector the coordinate type must be Cartesian.

Output of Transformation Computation

Transformation Parameters

The command *CallTransformation()* returns a structure *CallTransformationRT* containing the seven parameters of the transformation (translation, Euler angles and scale) together with their standard deviations. The standard deviation of a fixed parameter is zero (0).

Transformed Points and Residuals

The command *GetTransformedPoints()* returns a list of structures, each containing a transformed point together with its covariance matrix and the three coordinates of the *residual* vector

$$residual = nominal - transformed$$

The covariance matrix of the residual is obtained by adding those of the nominal and the transformed point.

Statistics

The command *CallTransformation()* also returns the

- RMS of residuals
- Maximal deviation
- Variance factor

RMS of Residuals

The RMS of residuals is defined as

$$RMS = \sqrt{\frac{\sum_{i=1}^{n} |residual|_i^2}{noEquations}}$$

where the number of equations is the number of fixed or weighted nominal coordinates.

Only those components of the residual vector corresponding to a fixed or weighted nominal coordinate are taken into account.

Maximum Deviation

The maximum deviation is defined as

$$maxDev = max_{i=1..n} |residual|_i$$

Only fixed and weighted nominals are taken into account.

Weighted Residual Square Sum

The transformation algorithm determines the values of the transformation parameters, to minimize the target function, weighted residual square sum.

$$RSS = \sum_{i=1}^{n} residual_i^T \, weightMatrix_i \, residual_i$$

The weight matrix is the inverse of the covariance matrix of the residual. Both matrices are scalars for constraints. The term, in the weighted least

squares sense, refers to the fact that RSS is the target function of the minimization algorithm.

Variance factor

The variance factor (mean error) takes the accuracy of the input into account:

$$varianceFactor = \frac{RSS}{redundancy}$$

The variance factor is dimensionless, i.e. it does not depend on the length or angle units. The value of the variance factor may vary considerably depending on the accuracy of the input and the model error, i.e. the size of the residuals. If the residuals are systematically bigger than the standard deviations of the actuals, the variance factor exceeds 1. Otherwise it is less than 1.

See "Examples" on page 137.

Redundancy

The redundancy is an integer defined as

$redundancy = noEquations - noParameters$.

In a minimum configuration, the redundancy is zero (0) and the variance factor is set to one (1).

If the redundancy is negative, the solution is non-unique. The error message multiple solutions is returned. In this case more fixed nominals or parameter constraints are needed to pick a unique solution.

Examples

Standard Case with 3 Points

```
AddNominalPoint(1, 2, 3, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(2, 3, 4, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(0, -4, 2, Fixed, Fixed, Fixed, 0, 0, 0);

SetInputParams(0, 0, 0, 0, 0, 0, 1, Unknown, Unknown, Unknown,
Unknown, Unknown, Unknown, Unknown);
```

In this example $redundancy = 3 * noPoints - 7 = 2$.

Pure Dilation

```
AddNominalPoint(1, 1, 0, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(-1, 1, 0, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(1, -1, 0, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(-1, -1, 0, Fixed, Fixed, Fixed, 0, 0, 0);

AddActualPoint(1.1, 1.1, 0, 0.001, 0.001, 0.001, 0, 0, 0);
AddActualPoint(-1.1, 1.1, 0, 0.001, 0.001, 0.001, 0, 0, 0);
AddActualPoint(1.1, -1.1, 0, 0.001, 0.001, 0.001, 0, 0, 0);
AddActualPoint(-1.1, -1.1, 0, 0.001, 0.001, 0.001, 0, 0, 0);

SetInputParams(0, 0, 0, 0, 0, 0, 1, Unknown, Unknown, Unknown,
Unknown, Unknown, Unknown, Fixed);
```

In this example the desired transformation is the identity with parameters 0, 0, 0, 0, 0, 0, 1. The length of all residuals is $0.1\sqrt{2}$. Their covariance matrix is

$$covar = \begin{pmatrix} 10^{-6} & 0 & 0 \\ 0 & 10^{-6} & 0 \\ 0 & 0 & 10^{-6} \end{pmatrix}$$

The weight matrix is

$$weight = \begin{pmatrix} 10^{6} & 0 & 0 \\ 0 & 10^{6} & 0 \\ 0 & 0 & 10^{6} \end{pmatrix}$$

Thus

$$RSS = 4 * 10^{6} * \left(0.1\sqrt{2}\right)^{2} = 80000$$

$$redundancy = 12 - 6 = 6$$

$$varianceFactor = \frac{80000}{6} = 13333.$$

3-2-1 Alignment

```
AddNominalPoint(1, 2, 3, Fixed, Fixed, Approx, 0, 0, 0);
AddNominalPoint(2, 3, 4, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(0, -4, 2, Approx, Fixed, Approx, 0, 0, 0);

SetInputParams(0, 0, 0, 0, 0, 0, 1, Unknown, Unknown, Unknown,
Unknown, Unknown, Unknown, Fixed);
```

This is a minimum configuration since

$$redundancy = 6 - 6 = 0$$

The approximate coordinates are necessary to select a unique solution from the 8 possible solutions. This fact can be easily observed in the following example:

```
AddNominalPoint(0, 0, 0, Fixed, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(1, 0, 0, Unknown, Fixed, Fixed, 0, 0, 0);
AddNominalPoint(1, 1, 0, Unknown, Unknown, Fixed, 0, 0, 0);
```

Here each of the Euler angles can be 0 or π. The scale must be fixed in 3-2-1 situations. Otherwise the solution is undetermined.

Box Corner

The corner of a box is defined by three mutually perpendicular planes. Each plane contains two measured points. Only the nominal coordinate defining the plane is exactly known.

```
AddNominalPoint(0, 1, 1, Fixed, Approx, Approx, 0, 0, 0);
AddNominalPoint(0, 2, 2, Fixed, Approx, Approx, 0, 0, 0);
AddNominalPoint(1, 0, 1, Approx, Fixed, Approx, 0, 0, 0);
AddNominalPoint(1, 0, 2, Approx, Fixed, Approx, 0, 0, 0);
AddNominalPoint(1, 1, 0, Approx, Approx, Fixed, 0, 0, 0);
AddNominalPoint(2, 2, 0, Approx, Approx, Fixed, 0, 0, 0);
```

This is also a minimum configuration with

$$redundancy = 6 - 6 = 0$$

provided the scale is fixed.

Orientation Using Nivel20 measurements

Suppose the horizontal angles ω and φ have been obtained from a Nivel20 measurement. To complete the orientation of the station, use a number of reference points with:

```
SetInputParams(0, 0, 0, omega, phi, 0, 1, Unknown, Unknown,
Unknown, Fixed, Fixed, Unknown, Fixed);
```

7.Appendix

TPI File Listing

Programming Interface Defining Files

The files *ES_C_API_Def.h*, *ES_CPP_API_Def.h*, *LTControl.dll /tlb* as well as all the sample projects are an integral part of the SDK.

- ES_C_API_Def.h
- ES_CPP_API_Def.h
- Enum.h
- LTControl.dll (Unicode version for WinNT/2000/XP)
- LTControl.dll (ANSI version for Win98/ME)
- LTControl.tlb

 The *ES_C_API_Def.h* file may currently be distributed in two parts, that is, with a sub-include file named *enum.h*.
If only one file is being distributed and no #include *enum.h* statement is included in *ES_C_API_Def.h,* it can be assumed that *enum.h* has been directly merged with the API include file.

 Sample files are no longer listed here. See 'Samples' folder on the SDK distribution medium.